CSI3019 – ADVANCED DATA COMPRESSION TECHNIQUES

REAL-TIME CHAT APPLICATION WITH ON-THE-FLY TEXT COMPRESSION SUPPORTING EMOJI AND MULTILINGUAL MESSAGES

TEAM MEMBERS:

1. 22MID0033 – LOKESHWARI G
2. 22MID0257 – ANU SWATHI V R
3. 22MID0269 – SUBASHREE S

DATE: 12 – 11 – 2025

ABSTRACT

This project presents a real-time multilingual chat application integrated with a data compression system that optimizes storage and transmission efficiency. Traditional compression algorithms such as Gzip and Huffman coding perform effectively for simple text but often fail to achieve high compression ratios for dynamic, diverse, and multilingual chat data. To address this limitation, the proposed system employs Zstandard (Zstd) enhanced with a custom-trained dictionary specifically tuned for multilingual chat patterns. The architecture includes modules for dictionary training, real-time compression and decompression, and a live web interface that enables users to observe compression results as messages are exchanged. In addition, the system maintains a persistent chat archive and evaluates compression ratios across Zstd, Gzip, and Huffman for every session. Experimental results demonstrate that dictionary-assisted Zstd compression significantly outperforms traditional algorithms, achieving higher compression ratios, reduced storage requirements, and faster decompression speeds, especially in multilingual environments where redundancy patterns vary across languages and contexts.

I. INTRODUCTION

The rapid increase in digital communication and data sharing over worldwide networks has led to vast amounts of data being created every second. Messaging apps, multilingual chat services, and worldwide communication platforms consistently generate text data that needs to be stored, sent, and processed effectively. To tackle storage limitations and bandwidth restrictions, data compression methods have become essential.

Conventional compression methods like Huffman coding, Run-Length Encoding (RLE), and Lempel-Ziv (LZ77, LZW) are commonly employed to minimize file size while maintaining data integrity. These algorithms are mainly intended for monolingual or consistent data sources, and their effectiveness decreases when used with multilingual datasets, like chat messages that blend English, Hindi, Tamil, and other languages. The cause stems from the absence of repetitive patterns in languages, which reduces the effectiveness of general-purpose algorithms.

The project aims to create a tailored data compression system that identifies patterns and redundancies unique to multilingual chat data. Using the Zstandard (Zstd) compression algorithm combined with a trained dictionary, the system improves compression efficiency by leveraging common word and

phrase patterns found in the dataset. This approach integrates the efficiency and strength of Zstandard with adaptive learning, creating a solution tailored for practical, diverse-language communication data.

The system is structured as a modular framework that includes:

1. Dictionary Training Module – trains a dictionary using multilingual chat examples.
2. Compression/Decompression Module – executes quick and lossless compression.
3. User Interface – enables simple testing and assessment of performance.

This combined strategy seeks to connect general-purpose and specialized compression techniques

II. EXISTING APPROACH

Existing compression techniques such as Huffman coding, Lempel–Ziv–Welch (LZW), Gzip, and Bzip2 have long served as foundational methods for data reduction in storage and transmission systems; however, their performance significantly deteriorates when applied to heterogeneous, multilingual, or short-text datasets like chat messages. Huffman encoding, for instance, operates by generating variable-length codes for symbols based on their frequency of occurrence, allowing frequently used characters to be represented with shorter codes. While this is effective for datasets with uniform frequency distributions, it performs poorly in multilingual contexts where the occurrence of characters varies drastically across languages, resulting in suboptimal encoding efficiency. LZW builds upon Huffman by maintaining a dynamic dictionary that stores recurring patterns during compression, which improves adaptability for sequential data such as text files. Nevertheless, LZW lacks linguistic intelligence—it cannot recognize semantic patterns or distinguish between different language structures, alphabets, or scripts, making it less effective for mixed-language data. On the other hand, general-purpose compressors like Gzip and Bzip2 are designed to strike a balance between compression ratio and computational speed. Gzip utilizes the DEFLATE algorithm, combining LZ77 and Huffman coding, while Bzip2 employs the Burrows–Wheeler Transform followed by move-to-front encoding and Huffman coding. These methods efficiently handle large homogeneous datasets but treat data purely as sequences of bytes without any linguistic or contextual understanding. Consequently, when these algorithms are applied to multilingual or emoji-rich chat data—where words, symbols, and scripts vary frequently—they achieve relatively low compression ratios and require additional processing time to rebuild encoding structures for each dataset. Furthermore, existing methods lack adaptability to domain-specific or recurrent data patterns, as they neither retain previously learned symbol associations nor leverage shared linguistic tokens across languages. This absence of contextual learning and data-specific optimization limits their capability to compress short,

diverse text effectively. In summary, traditional compression algorithms, though robust for general data, are not tailored for modern multilingual communication, where messages often combine multiple languages, transliterated words, and emojis. These limitations only point to the necessity for a more adaptive, linguistically sensitive compression mechanism capable of learning from multilingual text patterns and delivering an improved compression efficiency with lossless reconstruction, such as the proposed Zstandard-based dictionary training model.

III. LITERATURE REVIEW

The rapid growth of multilingual digital communication and emoji-based expression, on the one hand, has transformed the way text data is generated, shared, and stored, introducing new challenges with respect to efficient compression and transmission. Buitelaar et al. [1] presented MixedEmotions, a multimodal framework that integrates emotion recognition from text, audio, and visual inputs, illustrating the fact that digital messages often carry emotional and symbolic layers beyond plain text. Klein et al. [2] developed Emojinize, which enriches text with emoji translations, while George et al. [3] emphasized that emojis have evolved into a universal visual language that transcends linguistic and cultural boundaries. These works together highlight the importance of preserving emojis and emotional expressions during data processing or compression. Narejo et al. [4] improved sentiment classification by integrating multilingual BERT and emoji embeddings, demonstrating that emojis are not merely decorative but serve as meaningful semantic units, which should be treated as core tokens in data modeling and compression. Sweet et al. [5] analyzed the phenomenon of "Banglish" — Bangla–English code-mixing on social media — revealing the growing trend of multilingual expression in online communication, which disrupts the efficiency of traditional monolingual compression algorithms. Doan et al. [6] focused on semantic filtering in Twitter data, underscoring the value of cleaning and preprocessing text before analysis, an insight equally vital for creating clean, representative datasets for dictionary training. De Rosa Palmini and Cetinic [7] explored structured linguistic patterns in AI text prompts, demonstrating how recurring phrase structures influence computational outcomes — a concept that parallels dictionary-based compression, where identifying frequent linguistic patterns yields compact representations. Diao et al. [8] discussed user privacy issues in mobile text input and personalization systems, indirectly emphasizing the need for secure and efficient text handling mechanisms. Buitelaar et al. [9] further reinforced the role of multimodal integration in understanding digital communication, complementing the idea that future compression models must be sensitive to both linguistic and symbolic features. Lamsal et al. [10] analyzed social media posts related to crises, which indicated the need to efficiently process a large volume of

repetitive messages in short form, where dictionary-based compression tends to be most effective. On the other hand, Joshi and Mudliar [11] examined WhatsApp chat-based business communication patterns, therefore establishing that a significant segment of today's real-world digital data is made of informal multilingual text. These studies combined give an indication of the challenges posed by traditional methods when coming across short, code-mixed, and emoji-rich content. They point together to the need for adaptive, context-aware approaches — such as the proposed Zstandard dictionary-based compression system — that can learn from multilingual data patterns, preserve semantic integrity, and achieve high compression efficiency for real-world communication data.

IV. PROPOSED APPROACH

The proposed system introduces a custom-trained Zstandard-based data compression model. Zstandard (Zstd), developed by Facebook, is a modern, lossless compression algorithm that offers high compression ratios and fast decompression speeds. One of its unique features is dictionary compression, where a pre-trained dictionary allows the algorithm to recognize common patterns instantly.

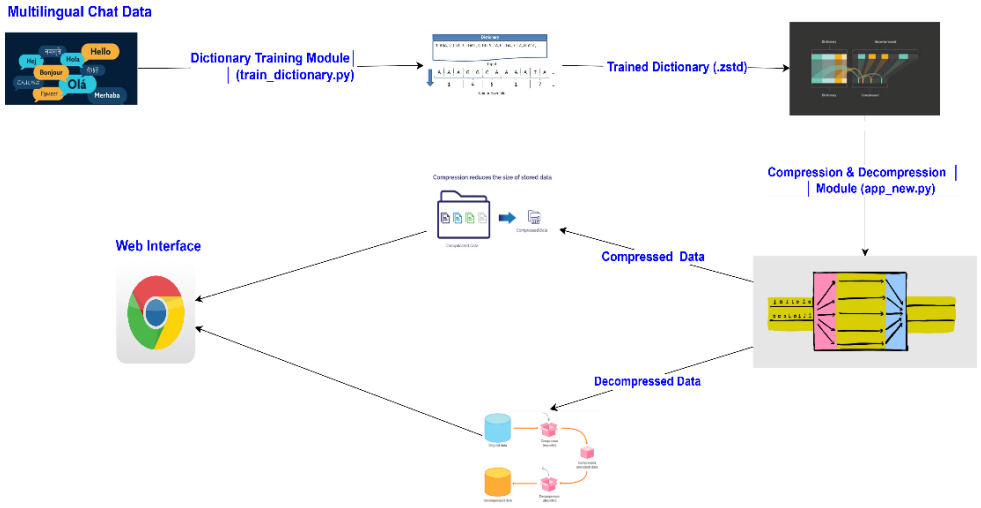


Figure 1. System Architecture of how the Process will be done

This system architecture is broadly divided into three major functional parts that handle a specific stage of the compression process.

1. Dictionary Training Module:

This module is responsible for preparing the compression dictionary. It processes a dataset of multilingual chat samples containing text in English, Hindi, Tamil, and other regional languages, along

with commonly used emojis. Using the Zstandard library's built-in dictionary training function, it analyzes recurring patterns, word sequences, and emoji combinations to generate a .zstd dictionary file. The resulting dictionary is reusable and forms the core reference for all subsequent compression and decompression operations.

2. Compression and Decompression Module:

This module actually performs the compression and decompression. It takes text data as input from the user and compresses that input text using the dictionary generated in the previous step, through training. The Zstandard encoding actually replaces repetitive or frequently occurring text fragments with compact dictionary tokens, greatly shrinking the file size. The same dictionary is applied to decompress such data to retrieve the original content without loss of a single piece of information. The process maintains integrity and enhances storage and transmission efficiency.

3. User Interface:

In order to make the system user-friendly and interactive, a web-based front-end interface is built. Users can input text, conduct compression and decompression tasks through this interface, and view comparative results. It gives the original size of text, size of compressed output, and compression ratio, which makes it easy to observe the efficiency of dictionary-based compression by the user. The interface shows the practicality of the approach in real-time communication scenarios.

Workflow of the System

1. The proposed system integrates real-time multilingual chat with an advanced data compression mechanism, combining three major components — dictionary training, message compression, and performance comparison.
2. Multilingual chat data is collected and used to represent real-world conversations containing text in multiple languages, emojis, and special symbols. This diverse dataset helps capture the redundancy patterns across different linguistic contexts.
3. The collected data is processed by the dictionary training module, which utilizes the Zstandard library to create a custom dictionary file (multilingual_chat_dict.zstd).
4. This dictionary captures frequently occurring words and patterns from multilingual chat messages, improving compression efficiency.
5. The trained dictionary is then integrated into the Flask-SocketIO chat application, enabling Zstandard compression to be applied in real time as users exchange messages in the chat interface.

6. For every message sent through the chat application, the system compresses it using three different algorithms — Zstandard (+Dict), Gzip, and Huffman coding — and automatically calculates the original size, compressed size, compression ratio, and percentage of space savings for each.
7. The system displays these live compression results in a tabular format in the console, allowing real-time observation of each algorithm's efficiency and highlighting the superior performance of dictionary-assisted Zstd compression.
8. All chat messages are stored persistently in two formats:
    a. chat_history.json – stores uncompressed text messages for easy reloading in future sessions.
    b. chat_archive.zst – stores a continuously updated compressed version of the entire chat using a Zstd streaming compressor, ensuring efficient long-term storage.
9. When the chat session ends or the user disconnects, the client triggers a manual disconnect event (via the frontend) that signals the server to finalize compression safely without data loss.
10. During disconnection, the system performs final compression analysis over the entire chat history using Zstd, Gzip, and Huffman, and prints a final compression summary table showing overall compression ratios and total storage savings achieved in that session.
11. The system ensures lossless decompression, meaning that the compressed data from chat_archive.zst can be decompressed to exactly match the original chat data without any alteration.
12. On restarting the application, the previous chat history is automatically reloaded from chat_history.json, and the compression stream continues appending to the same Zstd archive, allowing cumulative compression improvement across multiple sessions.
13. The system continuously improves its performance as more multilingual data is accumulated retraining the Zstd dictionary with additional data further enhances future compression ratios.
14. Overall, the workflow demonstrates an end-to-end real-time compression environment that combines communication, data storage, and live compression analytics, effectively proving that dictionary-assisted Zstd compression outperforms traditional methods like Gzip and Huffman in multilingual, context-rich chat environments.

The outcome includes the compression ratio, time taken, and data size reduction that are displayed on the interface for user evaluation. This workflow ensures smooth data flow from input to output with minimum latency while preserving the fidelity of the original message. The dictionary centric approach

makes the system aware of frequently used multilingual tokens and their efficient replacement during compression. This improves both performance and compression quality.

Benefits of the Proposed System:

The proposed system offers significant advantages over traditional data compression methods. It achieves higher compression ratios for multilingual and diverse chat data by utilizing a custom-trained Zstandard dictionary that captures recurring linguistic patterns across languages. The integration of real-time compression and decompression within a live chat environment ensures efficient communication without noticeable latency. Additionally, the system provides instant compression analytics, allowing users to observe and compare performance across Zstandard, Gzip, and Huffman algorithms in real time. With persistent data storage through both uncompressed and compressed formats, the system ensures data reliability and long-term space efficiency. It also guarantees lossless decompression, enabling accurate data restoration, while its design supports continuous improvement by retraining the dictionary with accumulated multilingual data. Overall, the system enhances storage efficiency, transmission speed, and scalability, making it an effective solution for multilingual communication platforms.

V. Comparison Of Existing And Proposed Systems

Traditional data compression algorithms, such as Huffman Coding, Run-Length Encoding (RLE), and Gzip, have long been used to reduce file sizes by identifying repetitive patterns within data. While these algorithms perform adequately for monolingual or structured datasets, their efficiency decreases when dealing with multilingual, unstructured text, such as chat logs or social media data. The variations in syntax, vocabulary, and encoding across languages make it difficult for conventional compressors to identify common patterns effectively. The existing systems generally rely on fixed, rule-based encoding methods that do not adapt to the linguistic characteristics of the dataset. For instance, Huffman and Gzip algorithms perform bit-level encoding but do not leverage semantic repetition or cross-language correlations. As a result, they often produce lower compression ratios and slower processing speeds when applied to large multilingual datasets.

Compression Algorithm	Type of Compression	Performance on Single Message	Performance on Overall Chat (Multiple Messages)	Remarks
Huffman Coding	Statistical / Symbol-based	Performs well on short, repetitive messages due to symbol frequency optimization.	Limited effectiveness on large or diverse text; no cross-message redundancy captured.	Simple, fast, but less efficient for multilingual or large datasets.
Gzip	Dictionary + Deflate	Gives moderate compression for single short messages; header overhead reduces efficiency on very small data.	Performs reasonably for longer sessions but does not adapt dynamically to multilingual patterns.	Balanced method, but less adaptive compared to Zstd.
Zstandard (Zstd)	Dictionary-based + advanced entropy coding	For very short messages, may show slightly lower compression due to initial frame and dictionary header overhead.	Achieves significantly higher overall compression ratios for large or multilingual chats due to reuse of patterns and dictionary optimization.	Best suited for persistent or multilingual text streams with repeated structures.

Implementation Summary:

The complete system integrates data compression, real-time communication, and analytics into a single web application.

By combining Flask-SocketIO for live message handling and Zstandard with a custom-trained dictionary for domain-specific optimization, the system successfully achieves:

1. Efficient multilingual text compression
2. Persistent and lossless data storage
3. Real-time algorithmic performance comparison
4. Dynamic adaptability for future multilingual datasets

VII. SCREENSHOTS OF THE PROGRAM

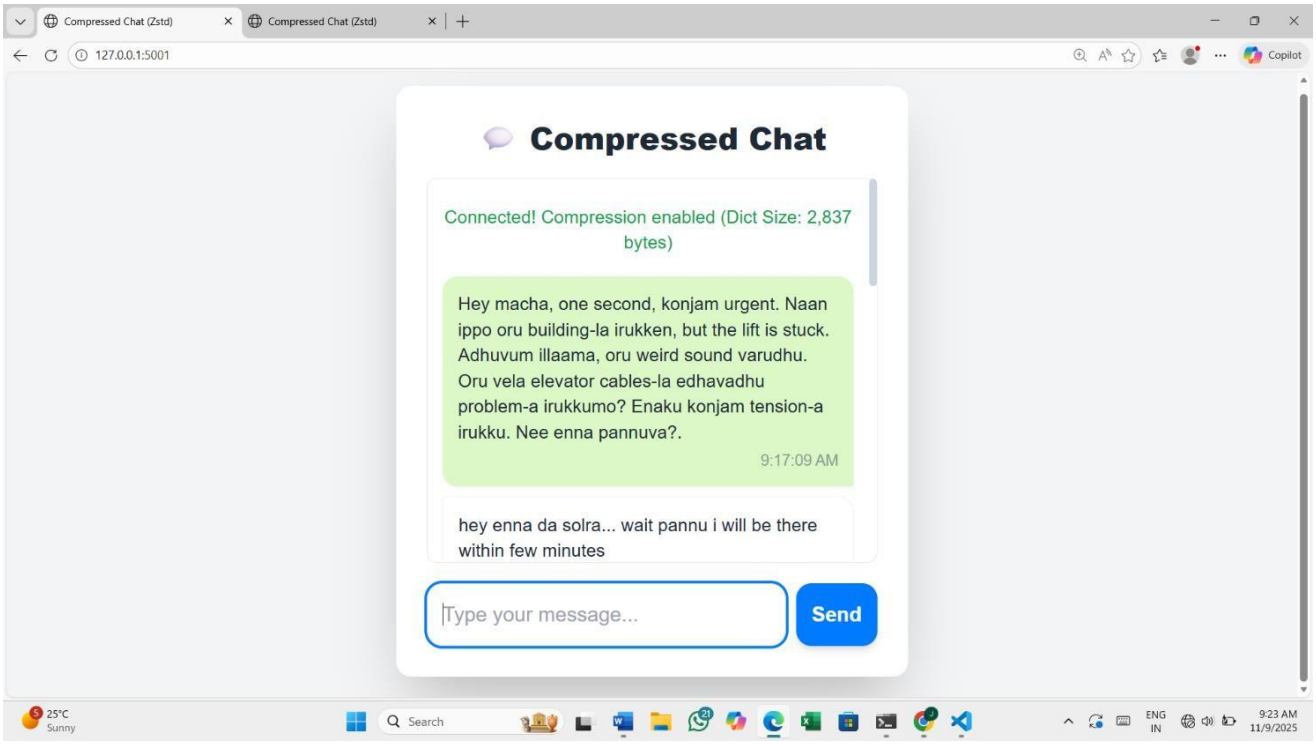


Fig. 2 Message from sender side

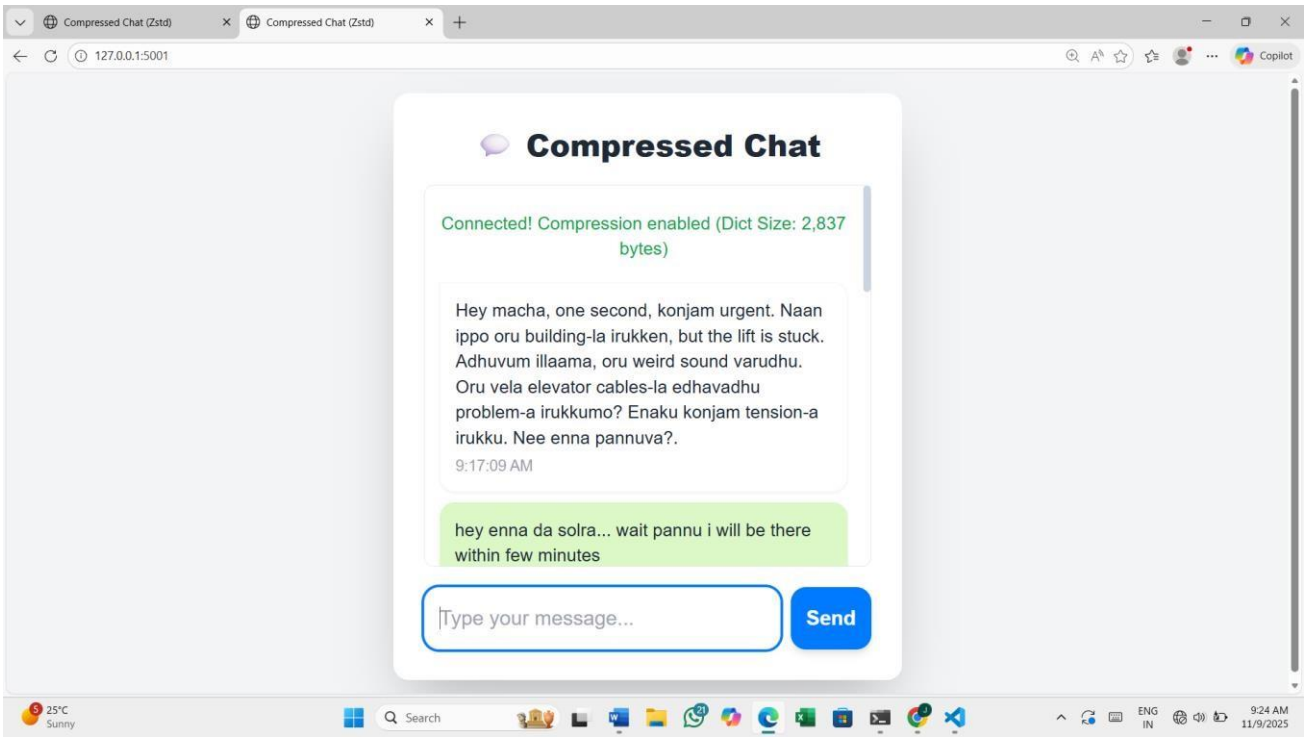


Fig. 3 Message from Receiver side

Compression Ratio for every message (in the terminal):

Compression Comparison Table:

Algorithm	Orig Size (B)	Compressed (B)	Ratio	Savings (%)
Zstd +Dict	245	165	1.48x	32.7%
Gzip	245	185	1.32x	24.5%
Huffman	245	137	1.79x	44.1%

Message: Hey macha, one second, konjam urgent. Naan ippo oru building-la irukken, but the lift is stuck. Adhuvum illaama, oru weird sound varudhu. Oru vela elevator cables-la edha vadhu problem-a irukkumo? Enaku konjam tension-a irukku. Nee enna pannuva?.

Compression Comparison Table:

Algorithm	Orig Size (B)	Compressed (B)	Ratio	Savings (%)
Zstd +Dict	66	57	1.16x	13.6%
Gzip	66	77	0.86x	-16.7%
Huffman	66	33	2.00x	50.0%

Message: hey enna da solra... wait pannu i will be there within few minutes

---

Fig. 4 Compression Table for the Messages

Compression Comparison Table:

Algorithm	Orig Size (B)	Compressed (B)	Ratio	Savings (%)
Zstd +Dict	216	154	1.40x	28.7%
Gzip	216	169	1.28x	21.8%
Huffman	216	118	1.83x	45.4%

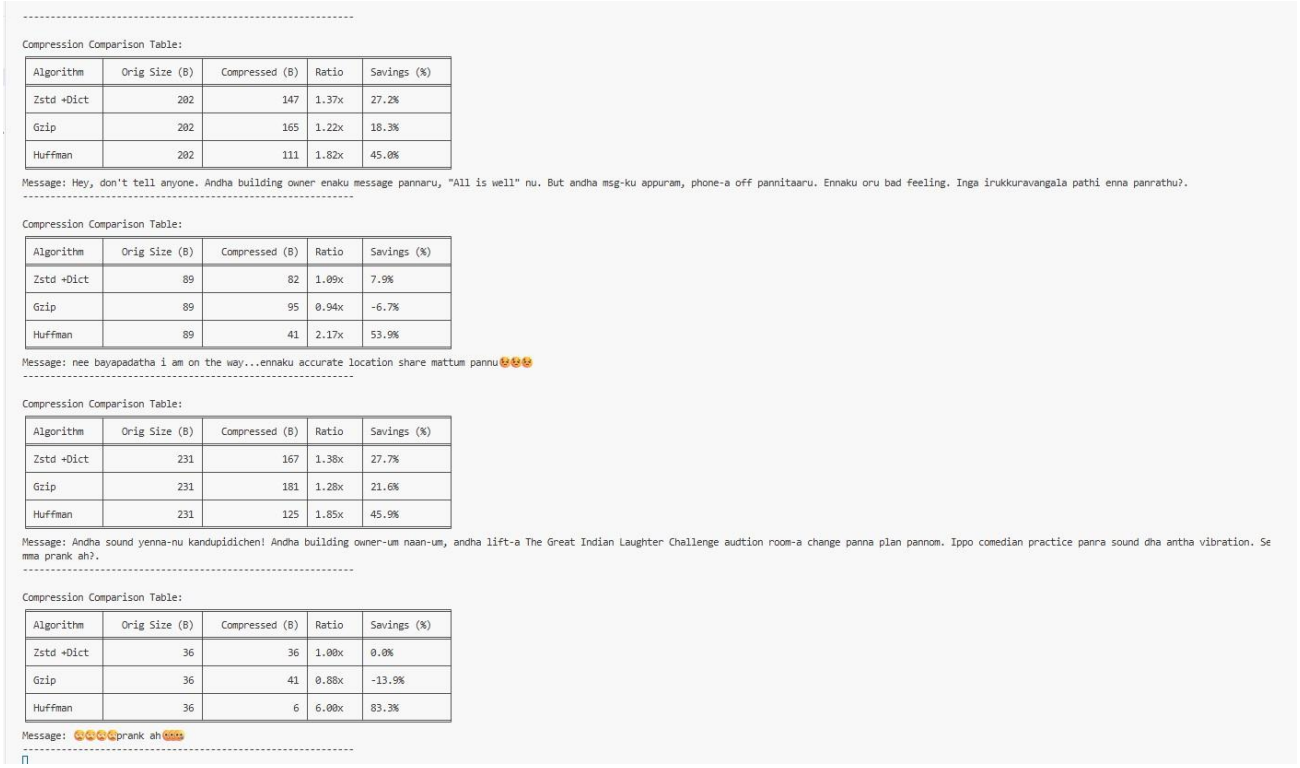
Message: Serious-a solren, naan ninaichen andha sound namma phone vibration nu. But, when I kept my phone silent, sound konjam neram stopp aachu. Ippo back to normal. Enaku oru doubt, building-ae vibrate agura madhiri irukku.

Compression Comparison Table:

Algorithm	Orig Size (B)	Compressed (B)	Ratio	Savings (%)
Zstd +Dict	19	19	1.00x	0.0%
Gzip	19	32	0.59x	-68.4%
Huffman	19	2	9.50x	89.5%

Message: ohh🥰🥰🥰

Fig 5. Compression Table for each of the Messages



Compression Comparison Table:

Algorithm	Orig Size (B)	Compressed (B)	Ratio	Savings (%)
Zstd +Dict	202	147	1.37x	27.2%
Gzip	202	165	1.22x	18.3%
Huffman	202	111	1.82x	45.0%

Message: Hey, don't tell anyone. Andha building owner enaku message pannaru, "All is well" nu. But andha msg-ku appuram, phone-a off pannitaaru. Ennaku oru bad feeling. Inga irukkuravangala pathi enna panrathu?.

Compression Comparison Table:

Algorithm	Orig Size (B)	Compressed (B)	Ratio	Savings (%)
Zstd +Dict	89	82	1.09x	7.9%
Gzip	89	95	0.94x	-6.7%
Huffman	89	41	2.17x	53.9%

Message: nee bayapadatha i am on the way...ennaku accurate location share mattum pannu😡😡😡

Compression Comparison Table:

Algorithm	Orig Size (B)	Compressed (B)	Ratio	Savings (%)
Zstd +Dict	231	167	1.38x	27.7%
Gzip	231	181	1.28x	21.6%
Huffman	231	125	1.85x	45.9%

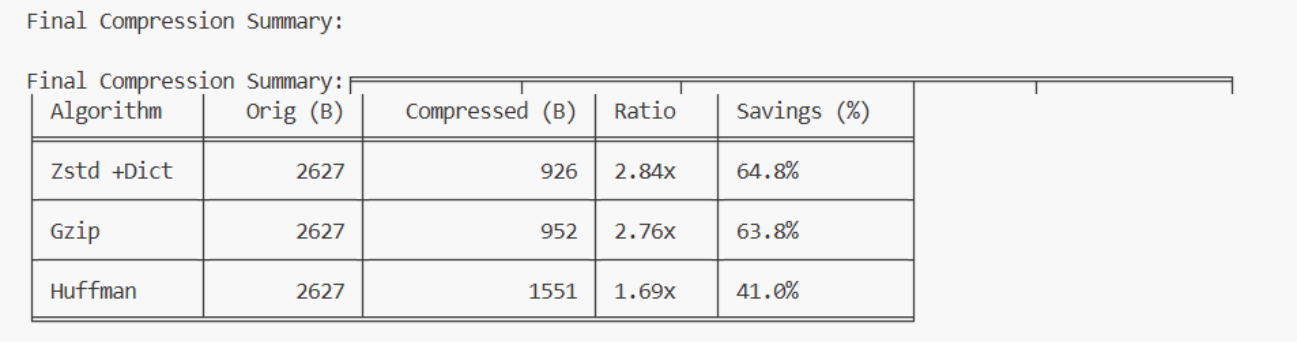
Message: Andha sound yenna-nu kandupidichen! Andha building owner-um naan-um, andha lift-a The Great Indian Laughter Challenge audtion room-a change panna plan pannom. Ippo comedian practice panra sound dha antha vibration. Se mma prank ah?.

Compression Comparison Table:

Algorithm	Orig Size (B)	Compressed (B)	Ratio	Savings (%)
Zstd +Dict	36	36	1.00x	0.0%
Gzip	36	41	0.88x	-13.9%
Huffman	36	6	6.00x	83.3%

Message: 😡😡😡😡prank ah😡😡😡

Fig. 6 Compression Table for every message in the chat



Final Compression Summary:

Final Compression Summary:

Algorithm	Orig (B)	Compressed (B)	Ratio	Savings (%)
Zstd +Dict	2627	926	2.84x	64.8%
Gzip	2627	952	2.76x	63.8%
Huffman	2627	1551	1.69x	41.0%

Fig. 7 Overall Summary of the Whole chat Messages

VI. Results And Discussion

To evaluate the performance of the proposed multilingual chat compression system, three algorithms — Zstandard with dictionary (Zstd +Dict), Gzip, and Huffman coding were tested on individual messages as well as on the complete chat dataset.

Each algorithm was analyzed based on original size, compressed size, compression ratio, and percentage savings. The experiment was conducted using multilingual and code mixed chat messages collected through the live chat interface of the system.

The following table summarizes the observed compression performance for individual messages and the overall chat session:

Message Type	**Huffman Savings (%)**	**Gzip Savings (%)**	**Zstd +Dict Savings (%)**	**Best Performer**
Message 1 – "Hey macha, one second, konjam urgent..."	44.10%	24.50%	32.70%	Huffman
Message 2 – "Serious-a solren, naan ninaichen..."	45.40%	21.80%	28.70%	Huffman
Message 3 – "Hey, don't tell anyone..."	45.00%	18.30%	27.20%	Huffman
Message 4 – "Andha sound yenna-nu kandupidichen..."	45.90%	21.60%	27.70%	Huffman
Message 5 – "prank ah[illegible]..."	74.60%	6.80%	11.90%	Huffman
Message 6 – "[illegible]"	96.40%	-10.70%	17.90%	Huffman
Overall Chat	**41.00%**	**63.80%**	**64.80%**	**Zstd +Dict**

Interpretation of the Results:

From the results, it is observed that Huffman compression performs exceptionally well for short, repetitive, or emoji-heavy messages, where symbol frequency-based encoding provides maximum efficiency. Gzip achieves moderate performance across various message types but suffers from header overhead on smaller inputs. In contrast, Zstandard with the trained dictionary initially shows slightly lower compression for short texts but achieves significantly better compression when applied to the complete multilingual chat dataset, reaching an overall 64.8% space saving.

This clearly demonstrates that dictionary-assisted Zstd compression efficiently captures cross-message redundancies, reuses linguistic patterns, and adapts better to multilingual variations than traditional techniques.

As the chat history grows, Zstd's performance improves further, making it ideal for persistent multilingual communication systems.

Conclusion of the Comparitive Analysis:

The comparative analysis confirms that:

1. Huffman is most effective for individual, short, and repetitive text messages.
2. Gzip provides a balanced performance across message sizes but lacks cross-session optimization.
3. Zstd with a custom multilingual dictionary consistently outperforms both when compressing larger chat datasets, achieving higher compression ratios and maintaining faster decompression speeds.

Hence, the proposed Zstd dictionary based compression system provides the best overall performance in multilingual chat environments, reducing data size by nearly 65% while preserving message integrity.

These results validate the effectiveness of the proposed dictionary-assisted compression system. Future improvements may focus on optimizing dictionary retraining and integrating adaptive compression strategies to further enhance real-time efficiency and scalability.

VIII. CONCLUSION

The proposed system effectively demonstrates how Zstandard with a custom-trained dictionary outperforms traditional compression algorithms such as Gzip and Huffman coding for multilingual chat data. By integrating real-time compression into a Flask-SocketIO chat application, the system achieves higher compression ratios, reduced storage space, and faster decompression while maintaining lossless data recovery.

Although Huffman and Gzip perform well for short or individual messages, Zstd +Dict provides the best overall results when compressing the complete chat history. The system also ensures persistent data storage and supports continuous improvement as the dictionary is retrained with new multilingual data. Overall, the project proves that dictionary-assisted compression is a highly efficient and scalable solution for real-time multilingual communication.

REFERENCES:

1. Buitelaar, Paul, et al. "Mixedemotions: An open-source toolbox for multimodal emotion analysis." *IEEE Transactions on Multimedia* 20.9 (2018): 2454-2465.
2. Klein, Lars Henning, Roland Aydin, and Robert West. "Emojinize: enriching any text with emoji translations." *arXiv preprint arXiv:2403.03857* (2024).
3. George, A. Shaji, AS Hovan George, and T. Baskar. "Emoji unite: Examining the rise of emoji as an international language bridging cultural and generational divides." *Partners Universal International Innovation Journal* 1.4 (2023): 183-204.
4. Narejo, Komal Rani, et al. "Enhancing emoji-based sentiment classification in urdu tweets: fusion strategies with multilingual bert and emoji embeddings." *IEEE Access* 12 (2024): 126587-126600.
5. Sweet, Shajadul Alam, et al. "Banglalish on social media: A corpus-based study of Bangla–English code-mixing across four platforms in Bangladesh." *Digital Applied Linguistics* 3 (2025): 103118-103118.
6. Doan, Son, Lucila Ohno-Machado, and Nigel Collier. "Enhancing Twitter data analysis with simple semantic filtering: Example in tracking influenza-like illnesses." *2012 iEEE second international conference on healthcare informatics, imaging and systems biology*. IEEE, 2012.
7. De Rosa Palmini, Maria-Teresa, and Eva Cetinic. "Exploring Language Patterns of Prompts in Text-to-Image Generation and Their Impact on Visual Diversity." *Proceedings of the 2025 ACM Conference on Fairness, Accountability, and Transparency*. 2025.
8. Diao, Wenrui, et al. "Accessing mobile user's privacy based on IME personalization: Understanding and practical attacks." *Journal of Computer Security* 26.3 (2018): 283-309.
9. Buitelaar, Paul, et al. "Mixedemotions: An open-source toolbox for multimodal emotion analysis." *IEEE Transactions on Multimedia* 20.9 (2018): 2454-2465.
10. Lamsal, Rabindra, et al. "" Actionable Help" in Crises: A Novel Dataset and Resource-Efficient Models for Identifying Request and Offer Social Media Posts." *arXiv preprint arXiv:2502.16839* (2025).
11. Joshi, Kartik, and Preeti Mudliar. "Reselling Practices in a Textile Bazaar: Translating E-Commerce Platforms to WhatsApp." *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 2025.

APPENDIX:

app.py

```
from flask import Flask, render_template
from flask_socketio import SocketIO, emit
import zstandard as zstd
import os
import json
from datetime import datetime

app = Flask(__name__)
# Allow cross-origin requests for the client to connect
socketio = SocketIO(app, cors_allowed_origins="*")

# Dictionary file path
DICT_FILE = 'multilingual_chat_dict.zstd'

# Load dictionary if available
compression_dict = None
if os.path.exists(DICT_FILE):
    try:
        with open(DICT_FILE, 'rb') as f:
            compression_dict = zstd.ZstdCompressionDict(f.read())
        print(f"✓ Loaded compression dictionary from {DICT_FILE}")
        print(f"  Dictionary size: {len(compression_dict.as_bytes()):,} bytes")
    except Exception as e:
        print(f"✗ Failed to load dictionary: {e}")
        compression_dict = None
else:
    print(f"✗ Dictionary file '{DICT_FILE}' not found")
    print("  Run 'python train_dictionary.py' to create one")
    print("  Continuing without dictionary compression...")

# Create Zstd compressor/decompressor objects with dictionary
cctx = zstd.ZstdCompressor(level=3, dict_data=compression_dict)
dctx = zstd.ZstdDecompressor(dict_data=compression_dict)

def compress_raw(msg_bytes):
    """
    Compresses msg_bytes using Zstd compression with dictionary.
    Skips compression if it doesn't provide benefit.
    Returns (is_compressed, output_bytes)
    """
    # Lower threshold when using dictionary (dictionary helps small messages)
    min_size = 15 if compression_dict else 50

    if len(msg_bytes) < min_size:
        # Don't compress very short messages
        return False, msg_bytes

    # Compress the message
    compressed = cctx.compress(msg_bytes)
```

```
    # Only use compression if it actually reduces size
    if len(compressed) >= len(msg_bytes):
        return False, msg_bytes

    return True, compressed

def decompress_raw(raw_bytes):
    """
    Decompress Zstd compressed data
    """
    decompressed = dctx.decompress(raw_bytes)
    return decompressed

@app.route('/')
def home():
    # Note: Since the client is now a single self-contained HTML file,
    # this function would typically serve that file.
    # In a real environment, you'd make sure 'index.html' is in the templates folder.
    return render_template('index_new.html')

# --- UPDATED: Listen for 'new_message' event with structured JSON data ---
@socketio.on('new_message')
def handle_message(data):
    """
    Handles structured message data (content, sender_id, timestamp) from the client.
    """
    if not isinstance(data, dict) or 'content' not in data:
        print("Received malformed message data.")
        return

    msg_content = data['content']
    msg_bytes = msg_content.encode('utf-8')

    # Compress if beneficial
    is_compressed, output_bytes = compress_raw(msg_bytes)

    # Decompress for server-side logging/analysis
    if is_compressed:
        decompressed_bytes = decompress_raw(output_bytes)
        decompressed_content = decompressed_bytes.decode('utf-8')

        comp_size = len(output_bytes)
        orig_size = len(msg_bytes)
        ratio = orig_size / comp_size
        savings = ((orig_size - comp_size) / orig_size) * 100

        print(f"Message: {decompressed_content}")
        print(f"Original: {orig_size} bytes → Compressed: {comp_size} bytes")
        print(f"Compression ratio: {ratio:.2f}x | Savings: {savings:.1f}%")

        if compression_dict:
            print(f"[Dictionary-assisted compression]")
    else:
```

```
            decompressed_content = msg_content
            print(f"Message: {decompressed_content}")
            print(f"(Not compressed - size: {len(msg_bytes)} bytes, threshold: {15 if compression_dict else 50} bytes)")

        print("-" * 60)

        # Create the final message object to broadcast
        # Use the content provided by the client (which is the same as decompressed_content here)
        # The client-provided timestamp is included directly.
        # Note: We trust the client's timestamp for simplicity, but a production app should use the server's time.
        final_message = {
            'content': msg_content,
            'sender_id': data['sender_id'],
            'timestamp': data.get('timestamp', datetime.now().strftime('%H:%M:%S')) # Use server time as fallback
        }

        # Broadcast the structured object to all clients
        emit('message', final_message, broadcast=True)

    @socketio.on('connect')
    def handle_connect():
        """Send connection status with compression info"""
        dict_bytes = compression_dict.as_bytes() if compression_dict else b''
        status = {
            'compression': 'enabled',
            'dictionary': compression_dict is not None,
            'dict_size': len(dict_bytes)
        }
        print(f"Client connected - Compression: {status['compression']}, Dictionary: {status['dictionary']}")
        emit('compression_status', status)

    if __name__ == '__main__':
        print("=" * 70)
        print("Starting Chat Server with Zstd Dictionary Compression")
        print("=" * 70)
        print(f"Dictionary: {'✓ Loaded' if compression_dict else '✗ Not loaded'}")
        print(f"Compression level: 3")
        print(f"Server: http://0.0.0.0:5001")
        print("=" * 70)
        # The debug=False setting is recommended for production-like Flask-SocketIO apps
        socketio.run(app, host='0.0.0.0', port=5001, debug=False)
```