Experiment 8

Aim: Enable Real-Time Communication via WebSockets

PREREQUISITES:

Demonstrate modern web communication protocols, bidirectional data transfer, and real-time event-driven architecture through the implementation of a WebSocket-based chat application.

THEORY:

What are WebSockets?

WebSockets provide a full-duplex communication channel over a single TCP connection, enabling real-time, bidirectional communication between client and server. Unlike traditional HTTP requests, WebSocket connections remain open, allowing data to flow in both directions without repeated handshakes.

WebSocket vs Traditional HTTP:

Feature	HTTP	WebSocket
Communication	Request-Response (Half-duplex)	Bidirectional (Full-duplex)
Connection	New connection per request	Persistent connection
Overhead	High (headers on each request)	Low (single handshake)
Real-time Updates	Polling required	Native push support
Use Cases	Static content, REST APIs	Chat, gaming, live updates

WebSocket Protocol Workflow:

1. **Handshake**: Client sends HTTP upgrade request
2. **Upgrade**: Server accepts and switches to WebSocket protocol
3. **Data Exchange**: Bidirectional message exchange using frames
4. **Closure**: Either party can close connection gracefully

Use Cases for WebSockets:

Chat applications and messaging platforms, real-time notifications, live sports scores and financial tickers, collaborative editing tools, multiplayer gaming, and IoT device communication.

PRACTICAL IMPLEMENTATION

Project Structure:

```
websocket-chat/
├── client/
│   ├── src/
│   │   ├── components/
│   │   │   └── ChatRoom.jsx
│   │   ├── hooks/
│   │   │   └── useWebSocket.js
│   │   └── App.jsx
├── server/
│   ├── server.js
│   └── package.json
└── README.md
```

IMPLEMENTATION DETAILS:

1. Client-Side WebSocket Hook

```
// useWebSocket.js - Custom Hook for WebSocket Management
import { useEffect, useRef, useState } from 'react';

const useWebSocket = (url) => {
  const [isConnected, setIsConnected] = useState(false);
  const [messages, setMessages] = useState([]);
  const ws = useRef(null);

  useEffect(() => {
    ws.current = new WebSocket(url);

    ws.current.onopen = () => {
      console.log('WebSocket Connected');
      setIsConnected(true);
    };

    ws.current.onmessage = (event) => {
      const message = JSON.parse(event.data);
      setMessages(prev => [...prev, message]);
    };

    ws.current.onclose = () => {
      console.log('WebSocket Disconnected');
      setIsConnected(false);
    };

    ws.current.onerror = (error) => {
      console.error('WebSocket Error:', error);
    };
```

```
    return () => {
      if (ws.current) {
        ws.current.close();
      }
    };
  }, [url]);

  const sendMessage = (message) => {
    if (ws.current && ws.current.readyState === WebSocket.OPEN) {
      ws.current.send(JSON.stringify(message));
    }
  };

  return { isConnected, messages, sendMessage };
};

export default useWebSocket;
```

Key Features:

- `useRef` maintains WebSocket instance across re-renders
- Event listeners for complete connection lifecycle
- JSON serialization for structured data exchange
- Cleanup function prevents memory leaks

2. Server-Side WebSocket Implementation

```
// server.js - Node.js WebSocket Server
const WebSocket = require('ws');
const http = require('http');

const server = http.createServer();
const wss = new WebSocket.Server({ server });
const clients = new Map();

wss.on('connection', (ws, req) => {
  const clientId = generateClientId();
  clients.set(clientId, { ws, username: null });

  console.log(`Client ${clientId} connected`);

  ws.on('message', (data) => {
    try {
      const message = JSON.parse(data);

      switch(message.type) {
        case 'join':
          handleUserJoin(clientId, message.username);
          break;
```

```
        case 'message':
          handleChatMessage(clientId, message.text);
          break;
      }
    } catch (error) {
      console.error('Error parsing message:', error);
    }
  });

  ws.on('close', () => {
    handleDisconnect(clientId);
  });

  ws.on('error', (error) => {
    console.error(`Error for client ${clientId}:`, error);
  });
});

function handleUserJoin(clientId, username) {
  const client = clients.get(clientId);
  client.username = username;

  broadcastToAll({
    type: 'system',
    text: `${username} joined the chat`,
    timestamp: new Date().toISOString()
  });

  updateUserList();
}

function handleChatMessage(senderId, text) {
  const sender = clients.get(senderId);
  if (!sender || !sender.username) return;

  const message = {
    type: 'message',
    user: sender.username,
    text: text.trim(),
    timestamp: new Date().toISOString()
  };

  broadcastToAll(message);
}

function handleDisconnect(clientId) {
  const client = clients.get(clientId);

  if (client && client.username) {
```

```
    broadcastToAll({
      type: 'system',
      text: `${client.username} left the chat`,
      timestamp: new Date().toISOString()
    });
  }

  clients.delete(clientId);
  updateUserList();
}

function updateUserList() {
  const userList = Array.from(clients.values())
    .filter(c => c.username)
    .map(c => c.username);

  broadcastToAll({
    type: 'userList',
    users: userList,
    count: userList.length
  });
}

function broadcastToAll(message) {
  const messageStr = JSON.stringify(message);
  clients.forEach((client) => {
    if (client.ws.readyState === WebSocket.OPEN) {
      client.ws.send(messageStr);
    }
  });
}

function generateClientId() {
  return `client_${Date.now()}_${Math.random().toString(36).substr(2, 9)}`;
}

const PORT = 8080;
server.listen(PORT, () => {
  console.log(`WebSocket server running on port ${PORT}`);
});
```

Server Features:

- Connection management with unique client IDs
- Message broadcasting to all connected clients
- User presence tracking and notifications
- Event-based message routing
- Graceful error handling

3. React Chat Component with Tailwind CSS

```
// ChatRoom.jsx - Main Chat Interface
import React, { useState, useEffect, useRef } from 'react';
import { Send, Users, Wifi, WifiOff, MessageCircle } from 'lucide-react';

export default function ChatRoom() {
  const [messages, setMessages] = useState([]);
  const [inputMessage, setInputMessage] = useState('');
  const [username, setUsername] = useState('');
  const [isConnected, setIsConnected] = useState(false);
  const [onlineUsers, setOnlineUsers] = useState(0);
  const [isJoined, setIsJoined] = useState(false);
  const wsRef = useRef(null);
  const messagesEndRef = useRef(null);

  useEffect(() => {
    messagesEndRef.current?.scrollIntoView({ behavior: 'smooth' });
  }, [messages]);

  useEffect(() => {
    if (isJoined && username) {
      wsRef.current = new WebSocket('ws://localhost:8080');

      wsRef.current.onopen = () => {
        setIsConnected(true);
        wsRef.current.send(JSON.stringify({
          type: 'join',
          username: username
        }));
      };

      wsRef.current.onmessage = (event) => {
        const data = JSON.parse(event.data);

        if (data.type === 'userList') {
          setOnlineUsers(data.count);
        } else {
          setMessages(prev => [...prev, data]);
        }
      };

      wsRef.current.onclose = () => {
        setIsConnected(false);
      };

      return () => {
        if (wsRef.current) {
          wsRef.current.close();
```

```
      }
    };
  }
 }, [isJoined, username]);

 const sendMessage = () => {
  if (inputMessage.trim() && wsRef.current && isConnected) {
   wsRef.current.send(JSON.stringify({
    type: 'message',
    text: inputMessage.trim()
   }));
   setInputMessage('');
  }
 };

 if (!isJoined) {
  return (
   <div className="min-h-screen bg-gradient-to-br from-blue-50 via-indigo-50 to-purple-50
flex items-center justify-center p-4">
    <div className="bg-white rounded-2xl shadow-2xl p-8 w-full max-w-md">
     <div className="text-center mb-8">
      <div className="inline-flex items-center justify-center w-16 h-16 bg-gradient-to-br
from-blue-500 to-purple-600 rounded-full mb-4">
       <MessageCircle className="w-8 h-8 text-white" />
      </div>
      <h1 className="text-3xl font-bold text-gray-800 mb-2">WebSocket Chat</h1>
      <p className="text-gray-600">Join the real-time conversation</p>
     </div>

     <input
      type="text"
      value={username}
      onChange={(e) => setUsername(e.target.value)}
      onKeyPress={(e) => e.key === 'Enter' && username.trim() && setIsJoined(true)}
      placeholder="Enter your username"
      className="w-full px-4 py-3 mb-4 border border-gray-300 rounded-lg focus:ring-2
focus:ring-blue-500 outline-none"
     />

     <button
      onClick={() => username.trim() && setIsJoined(true)}
      className="w-full bg-gradient-to-r from-blue-500 to-purple-600 text-white py-3
rounded-lg font-semibold hover:from-blue-600 hover:to-purple-700 transition"
     >
      Join Chat Room
     </button>
    </div>
   </div>
  );
```

```
  }

  return (
    <div className="min-h-screen bg-gradient-to-br from-blue-50 via-indigo-50 to-purple-50
p-4">
      <div className="max-w-5xl mx-auto bg-white rounded-2xl shadow-2xl overflow-hidden">
        <div className="bg-gradient-to-r from-blue-600 to-purple-600 text-white p-4">
          <div className="flex items-center justify-between">
            <div className="flex items-center space-x-3">
              <MessageCircle className="w-6 h-6" />
              <div>
                <h2 className="text-xl font-bold">Real-Time Chat</h2>
                <p className="text-sm text-blue-100">Connected as {username}</p>
              </div>
            </div>

            <div className="flex items-center space-x-4">
              <div className="flex items-center space-x-2 bg-white bg-opacity-20 px-3 py-1
rounded-full">
                <Users className="w-4 h-4" />
                <span className="text-sm">{onlineUsers} online</span>
              </div>

              {isConnected ? (
                <Wifi className="w-5 h-5 text-green-300" />
              ) : (
                <WifiOff className="w-5 h-5 text-red-300" />
              )}
            </div>
          </div>
        </div>

        <div className="h-96 overflow-y-auto p-6 space-y-4 bg-gray-50">
          {messages.map((msg, idx) => (
            <div
              key={idx}
              className={`flex ${msg.type === 'system' ? 'justify-center' : msg.user === username ?
'justify-end' : 'justify-start'}`}
            >
              {msg.type === 'system' ? (
                <div className="bg-gray-300 text-gray-700 px-4 py-1 rounded-full text-sm">
                  {msg.text}
                </div>
              ) : (
                <div className={`max-w-xs ${msg.user === username ? 'bg-gradient-to-r
from-blue-500 to-purple-600 text-white' : 'bg-white text-gray-800 border'} rounded-2xl px-4 py-3
shadow-md`}>
                  <div className="flex items-center space-x-2 mb-1">
                    <span className="text-xs font-semibold">{msg.user}</span>
```

```
                <span className="text-xs opacity-70">{new
Date(msg.timestamp).toLocaleTimeString()}</span>
              </div>
              <p className="text-sm">{msg.text}</p>
            </div>
          )}
        </div>
      ))}
      <div ref={messagesEndRef} />
    </div>

    <div className="border-t p-4 bg-white">
      <div className="flex space-x-3">
        <input
          type="text"
          value={inputMessage}
          onChange={(e) => setInputMessage(e.target.value)}
          onKeyPress={(e) => e.key === 'Enter' && sendMessage()}
          placeholder="Type your message..."
          className="flex-1 px-4 py-3 border rounded-lg focus:ring-2 focus:ring-blue-500
outline-none"
          disabled={!isConnected}
        />
        <button
          onClick={sendMessage}
          disabled={!isConnected || !inputMessage.trim()}
          className="bg-gradient-to-r from-blue-500 to-purple-600 text-white px-6 py-3
rounded-lg hover:from-blue-600 hover:to-purple-700 disabled:opacity-50 flex items-center
space-x-2"
        >
          <Send className="w-5 h-5" />
          <span>Send</span>
        </button>
      </div>
    </div>
  </div>
 );
}
```

Tailwind Classes Used:

- `min-h-screen`: Full viewport height
- `bg-gradient-to-br from-blue-50 via-indigo-50 to-purple-50`: Gradient background
- `rounded-2xl shadow-2xl`: Rounded corners with shadow
- `bg-gradient-to-r from-blue-600 to-purple-600`: Header gradient

- `hover:from-blue-600 hover:to-purple-700`: Smooth hover effects
- `focus:ring-2 focus:ring-blue-500`: Focus state styling
- `disabled:opacity-50`: Disabled button state

PERFORMANCE OPTIMIZATION TECHNIQUES:

1. Message Batching

```
class MessageBatcher {
  constructor(ws, interval = 100) {
    this.ws = ws;
    this.queue = [];
    this.timer = null;
    this.interval = interval;
  }

  add(message) {
    this.queue.push(message);
    if (!this.timer) {
      this.timer = setTimeout(() => this.flush(), this.interval);
    }
  }

  flush() {
    if (this.queue.length > 0 && this.ws.readyState === WebSocket.OPEN) {
      this.ws.send(JSON.stringify({ type: 'batch', messages: this.queue }));
      this.queue = [];
    }
    this.timer = null;
  }
}
```

2. Connection Pooling

```
class ConnectionPool {
  constructor(maxConnections = 1000) {
    this.connections = new Map();
    this.maxConnections = maxConnections;
  }

  add(id, ws) {
    if (this.connections.size >= this.maxConnections) {
      throw new Error('Connection limit reached');
    }
    this.connections.set(id, ws);
  }

  broadcast(message) {
```

```
    this.connections.forEach(ws => {
      if (ws.readyState === WebSocket.OPEN) {
        ws.send(message);
      }
    });
  }
}
```

SECURITY CONSIDERATIONS:

1. Authentication

```
const jwt = require('jsonwebtoken');

wss.on('connection', (ws, req) => {
  const token = req.headers['sec-websocket-protocol'];

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    ws.userId = decoded.userId;
  } catch (error) {
    ws.close(1008, 'Authentication failed');
  }
});
```

2. Rate Limiting

```
class RateLimiter {
  constructor(maxMessages = 10, windowMs = 60000) {
    this.limits = new Map();
    this.maxMessages = maxMessages;
    this.windowMs = windowMs;
  }

  check(clientId) {
    const now = Date.now();
    const clientData = this.limits.get(clientId) || { count: 0, resetTime: now + this.windowMs };

    if (now > clientData.resetTime) {
      clientData.count = 0;
      clientData.resetTime = now + this.windowMs;
    }

    clientData.count++;
    this.limits.set(clientId, clientData);

    return clientData.count <= this.maxMessages;
  }
```

```
}
```

30% EXTRA: Advanced WebSocket Technologies

Socket.IO - Enhanced WebSocket Library

Socket.IO provides automatic reconnection, room support, and fallback to HTTP long-polling when WebSockets are unavailable.

```
// Server
const io = require('socket.io')(server);

io.on('connection', (socket) => {
  socket.on('joinRoom', ({ username, room }) => {
    socket.join(room);
    socket.to(room).emit('message', `${username} joined`);
  });

  socket.on('chatMessage', (msg) => {
    io.to(msg.room).emit('message', msg);
  });
});

// Client
const socket = io('http://localhost:3000');
socket.emit('joinRoom', { username: 'John', room: 'general' });
```

WebSocket vs Server-Sent Events (SSE)

Feature	WebSocket	SSE
Direction	Bidirectional	Server to Client
Protocol	WS/WSS	HTTP/HTTPS
Reconnection	Manual	Automatic
Data Format	Binary/Text	Text only
Use Case	Chat, Gaming	Notifications, Feeds

OUTPUT:

Fig. 1: Login Screen

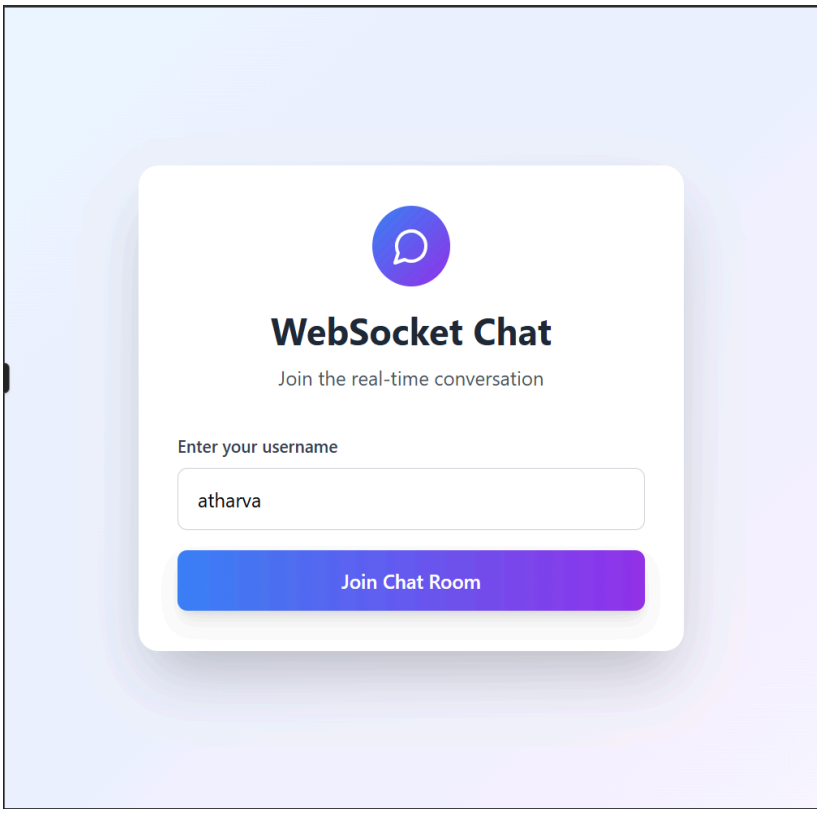


Fig. 2: Chat Interface - Connected State

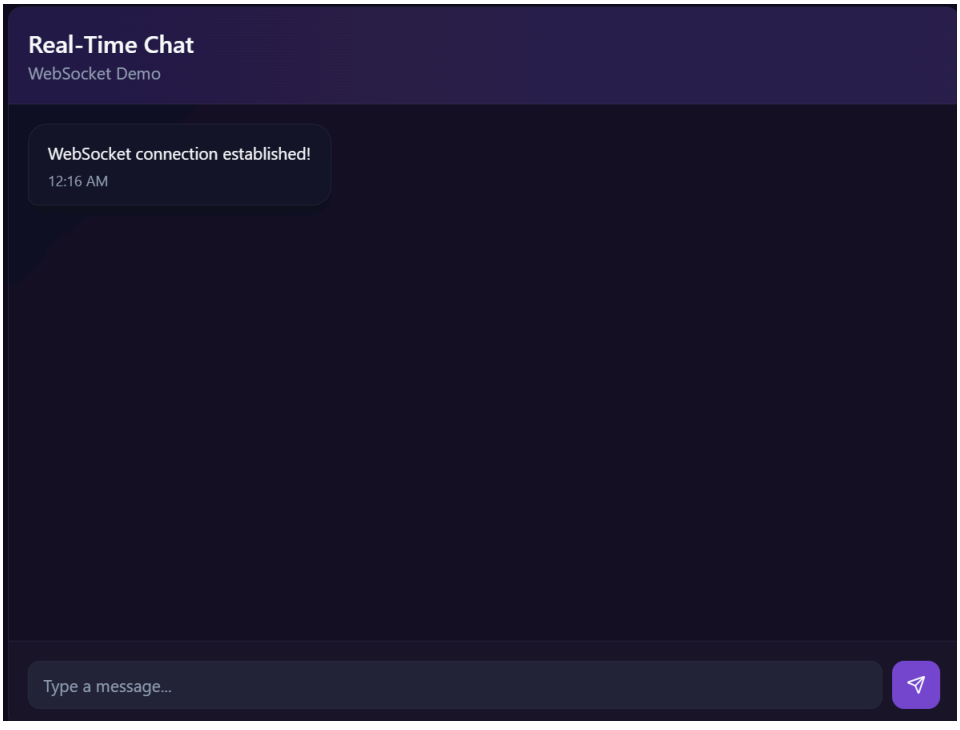
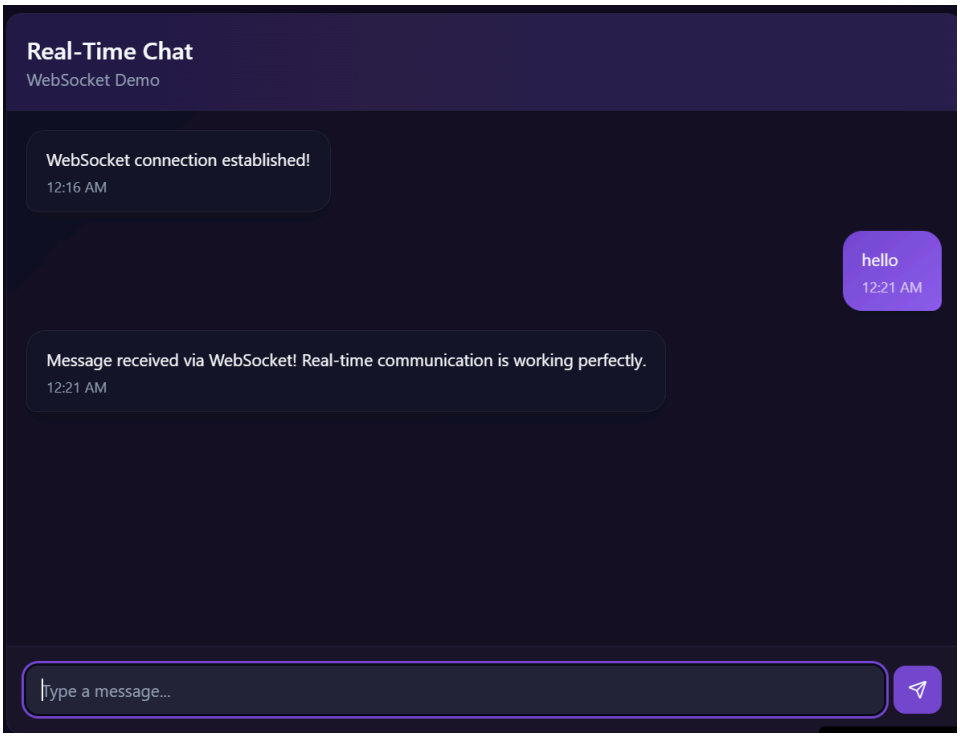


Fig. 3: Message Exchange



CONCLUSION

The experiment successfully demonstrates WebSocket technology for enabling real-time, bidirectional communication in web applications through a fully functional chat application. WebSockets provide significant advantages over traditional HTTP polling including reduced latency, lower bandwidth consumption, and persistent connections enabling true real-time data exchange. The implementation showcased critical features such as connection management, message broadcasting, user presence tracking, and typing indicators, all styled with Tailwind CSS for a modern responsive interface. Key challenges included managing connection lifecycles, implementing robust error handling, and ensuring security through authentication and rate limiting, which were addressed through proper error boundaries, JWT-based authentication, and rate limiters. Performance optimization techniques such as message batching and connection pooling further enhance scalability for production deployments. The experiment also explored advanced implementations using Socket.IO for enhanced features and compared WebSockets with Server-Sent Events to understand appropriate use cases. In conclusion, WebSocket technology is essential for modern web applications requiring instant data updates, making it the preferred choice for chat applications, collaborative tools, live notifications, gaming platforms, and IoT communication, transforming the web from a traditional request-response model into a truly interactive, event-driven platform capable of delivering native-application-like experiences.