WebSockets at Scale: Real-Time Architectures with NestJS and Redis Pub/Sub

Introduction

In today's fast-moving digital world, **real-time interactivity** is no longer optional—it's expected.

- Examples include live chat, financial dashboards, collaborative tools, and multiplayer games.
- Users demand **instant updates** and **seamless communication**.

WebSockets enable this by maintaining a **persistent, full-duplex connection** between client and server, enabling low-latency communication.

However, building real-time features is just the start. The true challenges arise when scaling:

- Supporting **thousands or millions of concurrent users**
- **Distributing** updates across multiple servers
- Ensuring **low latency** globally

To meet these demands, applications need a **robust architecture**. This is where **NestJS** (a progressive Node.js framework) and **Redis Pub/Sub** come in.

As user expectations evolve, applications must:

- Deliver **faster experiences**
- Provide **immediate responses**
- Scale across **distributed environments**

WebSockets are ideal for real-time communication due to:

- Efficiency
- Broad browser support
- Reduced message delivery time via persistent connections

The **Publish/Subscribe (Pub/Sub) pattern**:

- Has long been used in messaging systems
- Enables **scalable message delivery** across services and servers

Redis Pub/Sub offers a practical implementation of this pattern:

- **Messages are published once and delivered to multiple subscribers**
- Ideal for **broadcasting updates** to many connected users simultaneously

In this article, we'll demonstrate:

- How to build a basic real-time service using **WebSockets** and **Redis Pub/Sub** in **NestJS**
- Architectural considerations for **scalability and maintainability**
- How the right patterns can simplify your **infrastructure** and **codebase**

Before diving into implementation, we'll first **define the key components** of the system.

What is pub/sub?

At the heart of the **Pub/Sub pattern** is messaging—small packets of data sent between systems via **channels** (or topics), which act as filters.

Clients can be **publishers**, **subscribers**, or both, and can interact with one or multiple channels as needed.

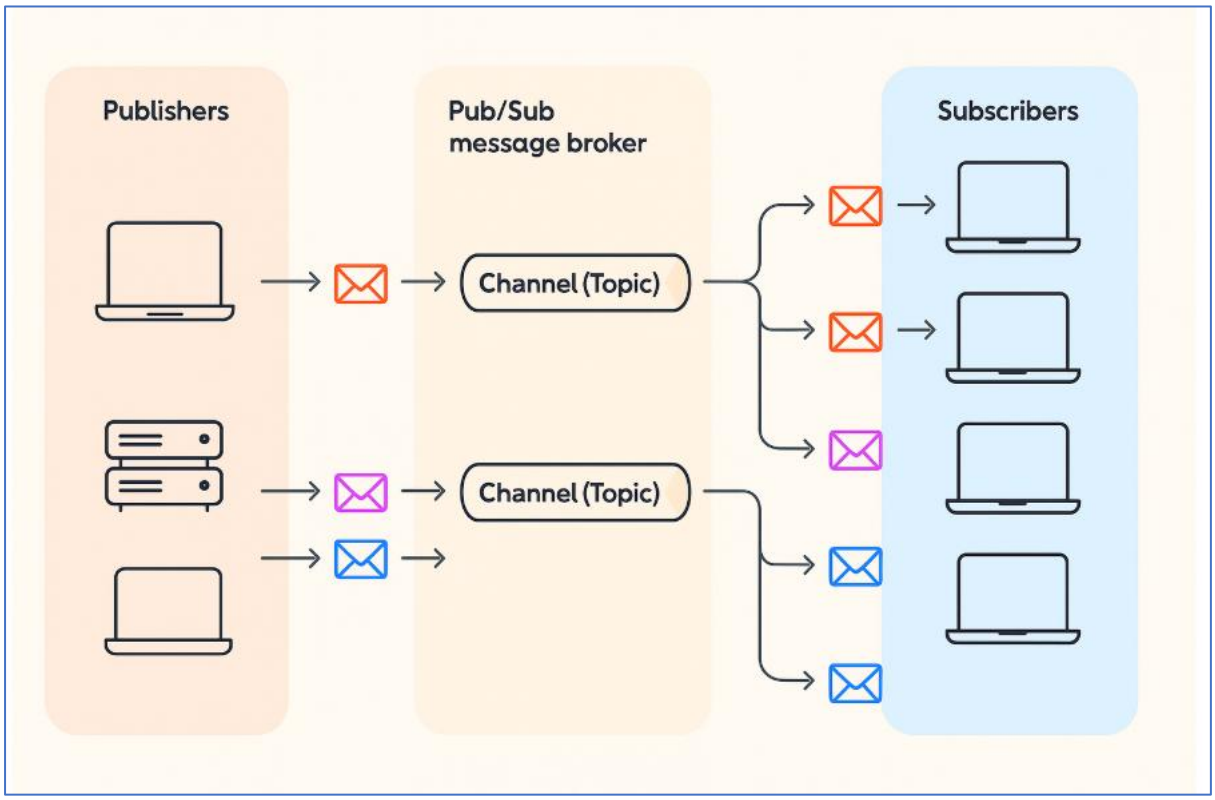
The pattern supports different communication models:

- **One-to-one** – Two clients exchange messages on the same channel (e.g., support chat).
- **One-to-many** – One source sends updates to multiple subscribers (e.g., live dashboards).
- **Many-to-one** – Multiple publishers send data to one destination (e.g., centralized logging).
- **Many-to-many** – All clients send and receive in a shared space (e.g., multiplayer games, group chats).

A major benefit is **decoupling** clients don't need to know about each other. Messages are routed through a **broker**, enabling asynchronous and scalable communication.

To make this work in production, a backend that ensures **reliable, low-latency delivery** is essential.

This is where **Redis** comes in: a high-performance data store with built-in **Pub/Sub** support, making it ideal for powering **real-time, distributed applications**.



What is Redis?

- **Redis** is a high-performance in-memory key-value store, widely known for its speed and efficiency.
- It operates primarily in memory for ultra-fast performance but also supports optional data persistence to disk.
- Capable of handling **millions of operations per second**, Redis is well-suited for high-throughput systems.
- One of its key features is built-in **publish/subscribe (Pub/Sub)** messaging, enabling real-time communication by instantly delivering messages to all subscribers of a channel.

What are WebSockets?

- WebSockets are a communication protocol designed for real-time, bidirectional interaction between clients and servers. Unlike traditional HTTP long polling—where the client repeatedly requests updates—WebSockets establish a persistent connection that enables data to flow continuously in both directions as events occur.

- This makes WebSockets ideal for low-latency, real-time applications such as live chat, push notifications, and collaborative editing tools. By keeping an open TCP connection, they eliminate the overhead of repeated HTTP requests, reducing latency and boosting performance—both essential for modern interactive experiences.
- WebSockets are most used to enable fast, direct communication between a web browser and a backend service.

Why WebSockets?

WebSockets provide a persistent, bidirectional connection between the client and server. Unlike HTTP, which is request-response based, WebSockets allow the server to push data to the client at any time.

Use cases:

- Live chat applications
- Real-time notifications
- Collaborative editing (e.g., Google Docs)
- Live dashboards and analytics
- Multiplayer games

Challenges of Scaling WebSockets

1. **Sticky sessions**: WebSocket connections are long-lived. Load balancing them often requires sticky sessions so the connection persists on the same server.
2. **State sharing**: In a multi-instance environment, one instance may receive an event, but the actual socket might be connected to another.
3. **Scalability**: Without centralized communication, instances can't broadcast to all connected clients across servers.
4. **Fault tolerance**: Node failures should not disrupt the global WebSocket state.

Solution: Introduce **Redis Pub/Sub** as a communication bridge between instances.

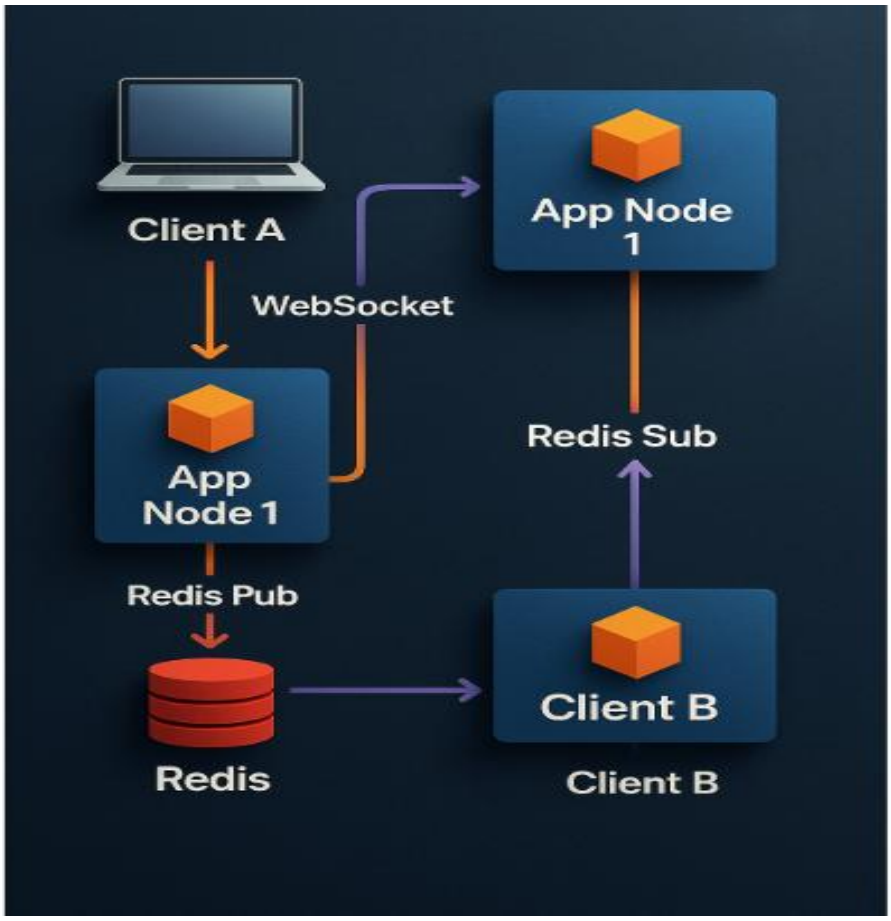
A tutorial: A simple pub/sub service

Now, let’s see how a simple real-time pub/sub service comes together with Redis and WebSockets, where multiple clients can subscribe to a channel and receive channel messages.

A typical architecture consists of a **WebSocket server** for handling client connections, backed by **Redis** as the Pub/Sub layer for distributing new messages. A **load balancer** like NGINX, or AWS ALB is used to handle incoming WebSocket connections and route them across multiple server instances; this is key to distributing load on our service.

Autoscaling can be used to dynamically adjust our servers to match demand, maintaining performance while reducing cost during quiet periods. To do this, we would need our WebSocket servers to remain stateless.

Conceptual Architecture Chart



Step-by-Step Implementation with NestJS

1. **Install Dependencies**

npm install @nestjs/websockets
@nestjs/platform-socket.io socket.io ioredis @nestjs/redis

2. **Set Up Redis Adapter for Socket.IO**

Create a custom adapter to integrate Redis Pub/Sub with WebSockets.

```ts
// redis.adapter.ts
import { IoAdapter } from '@nestjs/platform-socket.io';
import { ServerOptions } from 'socket.io';
import { createAdapter } from '@socket.io/redis-adapter';
import { createClient } from 'redis';

export class RedisIoAdapter extends IoAdapter {
  async createIOServer(port: number, options?: ServerOptions): Promise<any> {
    const server = super.createIOServer(port, options);

    const pubClient = createClient({ url: 'redis://localhost:6379' });
    const subClient = pubClient.duplicate();

    await pubClient.connect();
    await subClient.connect();

    server.adapter(createAdapter(pubClient, subClient));
    return server;
  }
}
```

3. **Apply the Adapter in main.ts**

```ts
...
async function bootstrap() {
    const app = await NestFactory.create(AppModule);
    const adapter = new RedisIoAdapter(app);
    app.useWebSocketAdapter(adapter);
    await app.listen(3000);
}
bootstrap();
```

4. **Create WebSocket Gateway**

```ts
import {
  SubscribeMessage,
  WebSocketGateway,
  WebSocketServer,
  MessageBody,
} from '@nestjs/websockets';
import { Server } from 'socket.io';

...
@WebSocketGateway()
export class EventsGateway {
  @WebSocketServer()
  server: Server;

  @SubscribeMessage('message')
  handleMessage(@MessageBody() data: string): void {
    this.server.emit('message', data); // Broadcast to all instances
  }
}
```

5. **Redis Pub/Sub Usage (Optional Logic Layer)**
You can also add Redis Pub/Sub manually in services for decoupled logic or custom broadcasting.

```
@Injectable()
export class NotificationService {
  private pubClient = new Redis();

  notifyAll(event: string, data: any) {
    this.pubClient.publish(event, JSON.stringify(data));
  }
}
```

Benefits of Combining Redis Pub/Sub with WebSockets

Using Redis Pub/Sub alongside WebSockets brings several powerful advantages to real-time systems. Here's how each feature translates into tangible benefits:

- **Decoupled Architecture**
  Redis allows server instances to communicate without being directly connected. This removes the need for tight coupling and makes your system easier to scale and maintain.
- **Seamless Broadcasting**
  Messages published to a Redis channel can be broadcast to all connected WebSocket clients, no matter which server instance they're connected to. This ensures consistent communication across the board.
- **Scalability Without Session Stickiness**
  Redis enables horizontal scaling, meaning you can spin up multiple WebSocket servers without worrying about which client is connected to which instance. No sticky sessions required.
- **Fault Tolerance and Isolation**
  If one WebSocket server goes down, Redis ensures that the rest of the system continues functioning. Communication between other clients and servers isn't affected.
- **Ultra-Low Latency**
  Redis is designed for speed, often delivering messages between publisher and subscribers in under 10 milliseconds—making it ideal for applications where every millisecond counts.

This combination lays the foundation for a responsive, resilient, and highly scalable real-time infrastructure.

Best Practices for Building Real-Time Systems with Redis and WebSockets

To build a robust and scalable real-time architecture, consider following these key best practices:

- **Keep WebSocket Logic Stateless**
  Avoid storing session data in memory on individual WebSocket servers. Instead, use Redis or a shared database to store user sessions and state. This ensures that any server can handle any client connection, enabling easier scaling and fault tolerance.
- **Use Namespaces and Rooms**
  Organize your communication channels by using namespaces and rooms. This helps separate different types of interactions (e.g., chat rooms, notifications, or live updates), making it easier to manage and route messages efficiently.
- **Implement Authentication and Rate Limiting**
  Always authenticate clients before establishing a WebSocket connection, and apply rate limiting to prevent abuse or overload. This adds a layer of security and protects your infrastructure from unnecessary strain.
- **Monitor Redis and Tune for Performance**
  Keep an eye on Redis memory usage and configure the maxmemory-policy to handle eviction gracefully under load. Proper monitoring helps avoid performance bottlenecks and ensures stable pub/sub behavior.
- **Use Redis Cluster in Production**
  In production environments, use Redis Cluster to distribute data and avoid a single point of failure. This improves resilience, availability, and scalability across your system.

Following these practices will help ensure your real-time system remains fast, secure, and resilient under heavy usage.

Real-World Use Cases of WebSockets and Redis Pub/Sub

Here are some practical examples where the combination of WebSockets and Redis Pub/Sub powers real-time communication at scale:

- **Chat Applications**
  Think of a WhatsApp-like experience where users can send and receive messages instantly. Using rooms, messages can be broadcasted to everyone in

a group chat in real time, ensuring seamless communication across multiple devices.

- **Live Stock Tickers**
  Financial platforms stream market data—such as stock prices, crypto values, or forex rates—to thousands of connected users. Redis ensures that updates are pushed with minimal delay, giving users up-to-the-second visibility.
- **IoT Dashboards**
  In IoT ecosystems, sensor data from multiple devices is sent live to dashboards used by administrators or engineers. Real-time streams help monitor performance, trigger alerts, and make decisions instantly.
- **Online Multiplayer Games**
  Games with real-time action—like player movement, actions, and game state changes—rely on low-latency, synchronized communication. WebSockets maintain continuous connectivity, while Redis ensures updates reach all players efficiently.

These use cases highlight how this architecture supports fast, scalable, and interactive experiences across a wide range of industries.

Conclusion

Building real-time applications with WebSockets unlocks a world of interactive experiences—but making them scale across multiple servers isn't always straightforward. That’s were combining **NestJS** with **Redis Pub/Sub** really shines. Redis handles the event distribution behind the scenes, allowing WebSocket messages to propagate instantly across all server instances, no matter how many users are connected.

Whether you're creating a chat platform, a live notification system, or a collaborative workspace, this setup gives you the reliability, scalability, and performance you need to grow with confidence.