PuigFlows

WebSockets a Escala: Backpressure, Heartbeats, Fan-out con Redis y Reconexión sin Pérdida de Mensajes

Una conexión WebSocket es estado, y el estado no escala solo.

Josue Garcia · 2026-09-07 · Edicion ampliada

Una conexión no es una petición

Un endpoint HTTP es cómodo porque olvida: recibe, responde y libera todo lo que estaba usando. Un WebSocket hace justo lo contrario. Cada cliente deja abierto un socket, un descriptor de fichero, un buffer de salida y un trozo de estado dentro de tu proceso durante horas. Multiplica eso por cincuenta mil y los problemas que aparecen no se parecen a los de una API REST.

Lo incómodo es que casi ninguno de esos problemas falla con un error claro. El proceso no lanza una excepción: se queda sin memoria. Las conexiones no se cierran: se quedan colgadas. Los mensajes no se pierden con un log de aviso: simplemente no llegan. Esta guía recorre los cinco puntos donde eso ocurre — backpressure, heartbeats, reparto entre instancias, reconexión y despliegues— con código que puedes copiar.

Los ejemplos usan Node.js con la librería ws y Redis, pero los patrones se traducen tal cual a Go, Elixir o Python.

1. Backpressure: el fallo silencioso que se lleva el proceso

Cuando llamas a ws.send() el mensaje no sale por el cable. Se copia a un buffer en memoria y el sistema operativo lo irá entregando al ritmo que el cliente sea capaz de leer. Si tú produces cien mensajes por segundo y el cliente — un móvil con mala cobertura, una pestaña en segundo plano con los timers estrangulados — consume cinco, la diferencia se queda en tu heap. Nadie te avisa. El proceso crece hasta que el kernel lo mata, normalmente de madrugada.

La señal que necesitas es ws.bufferedAmount: los bytes que ya aceptaste pero que todavía no han salido. Si ese número sube y no vuelve a bajar, ese cliente va más lento que tú.

```
import { WebSocketServer } from 'ws';

const MAX_BUFFER = 1 << 20; // 1 MiB de cola por cliente

const wss = new WebSocketServer({
  port: 8080,
  // Desactivado a propósito: comprimir por mensaje mantiene un contexto
  // de zlib por conexión y se paga en memoria. Ver "errores comunes".
  perMessageDeflate: false,
  maxPayload: 64 * 1024, // rechaza frames gigantes del cliente
});

function safeSend(ws, payload) {
  if (ws.readyState !== ws.OPEN) return false;

  // bufferedAmount = bytes aceptados que el socket aún no ha entregado.
  if (ws.bufferedAmount > MAX_BUFFER) {
    ws.slowStrikes = (ws.slowStrikes ?? 0) + 1;

    if (ws.slowStrikes > 3) {
      // 1013 = Try Again Later. Cerrar es mejor que acumular memoria
      // hasta que el kernel mate el proceso.
      ws.close(1013, 'client too slow');
```

```
    }
    return false; // mensaje descartado
  }

  ws.slowStrikes = 0;
  ws.send(payload);
  return true;
}
```

Ante un cliente lento hay tres respuestas válidas, y elegir entre ellas es una decisión de producto, no de infraestructura:

- **Descartar.** Correcto para datos de estado: una cotización, la posición de un cursor, un contador de presencia. El cliente necesita el valor actual, no la historia.
- **Colapsar.** En lugar de descartar, guarda el último valor y vacíalo a ritmo fijo. Un pico de mil actualizaciones se convierte en un envío.
- **Cerrar.** Si cada mensaje importa —un chat, una traza de logs— no puedes descartar. Cierra con 1013 y deja que el cliente reconecte y pida lo que le falta; la sección 4 explica cómo.

```
// Colapsado: un snapshot por conexión, vaciado a ritmo fijo.
const pending = new Map(); // ws -> último snapshot

export function queueSnapshot(ws, snapshot) {
  pending.set(ws, snapshot);
}

setInterval(() => {
  for (const [ws, snapshot] of pending) {
    pending.delete(ws);
    safeSend(ws, JSON.stringify(snapshot));
  }
}, 50); // 20 envíos por segundo como máximo, pase lo que pase
```

Lo único que nunca es válido es no hacer nada. Sin un límite, un solo cliente lento puede tumbar la instancia entera y llevarse por delante a los otros veinte mil.

2. Heartbeats: la mitad de la conexión que ya no existe

TCP no te avisa cuando el otro extremo desaparece. Si un portátil se suspende, si un móvil salta de Wi-Fi a datos, si un NAT intermedio recicla su tabla, tu servidor nunca recibe un FIN. Se queda con una conexión *medio abierta*: para tu proceso está viva —ocupa memoria y una plaza de tu límite de conexiones— pero no hay nadie al otro lado. En un servicio con rotación alta ese goteo es exactamente lo que hace que la memoria suba en escalera y no baje nunca.

El protocolo WebSocket incluye frames de control *ping* y *pong* justo para esto. Tu aplicación no los ve y no hay que tocar el cliente: el navegador responde al ping por su cuenta.

```
const HEARTBEAT_MS = 30_000;
```

```
wss.on('connection', (ws) => {
  ws.isAlive = true;
  ws.on('pong', () => { ws.isAlive = true; });
});

const heartbeat = setInterval(() => {
  for (const ws of wss.clients) {
    if (ws.isAlive === false) {
      // terminate(), no close(): close() inicia un handshake de cierre
      // y espera una respuesta que nunca va a llegar.
      ws.terminate();
      continue;
    }
    ws.isAlive = false;
    ws.ping();
  }
}, HEARTBEAT_MS);

wss.on('close', () => clearInterval(heartbeat));
```

Hay un detalle que se pasa por alto siempre: **tu intervalo de heartbeat tiene que ser menor que el idle timeout de cualquier proxy que haya en medio.** Un ALB de AWS corta a los 60 segundos sin tráfico por defecto, y nginx hace lo mismo. Si tu ping va cada 90 segundos vas a ver desconexiones "aleatorias" cada minuto en conexiones perfectamente sanas, y vas a perder una tarde buscando el bug en tu código.

```
location /ws {
  proxy_pass http://ws_upstream;
  proxy_http_version 1.1;
  proxy_set_header Upgrade    $http_upgrade;
  proxy_set_header Connection "upgrade";

  # Por defecto nginx cierra a los 60 s sin datos del upstream. Con
  # WebSockets eso mata conexiones sanas pero silenciosas.
  proxy_read_timeout 300s;
  proxy_send_timeout 300s;
}
```

3. Fan-out entre instancias: un índice local y un bus compartido

Con el backpressure controlado y las conexiones muertas fuera, el siguiente problema aparece en cuanto dejas de tener una sola instancia.3. Escalado horizontal: el backplane

Con una sola instancia, difundir un mensaje a una sala es recorrer un Set. Con cuatro instancias detrás de un balanceador, los miembros de esa sala están repartidos y cada proceso solo conoce a los suyos. Necesitas un canal que lleve el mensaje a las instancias que tienen destinatarios: eso es un *backplane*.

El patrón no cambia: cada instancia mantiene un índice local de sala a sockets propios, se suscribe en el backplane únicamente a las salas que tiene abiertas, y publica ahí en vez de difundir directamente.

```
import { createClient } from 'redis';
```

```
// Dos conexiones: en modo suscriptor, Redis no acepta otros comandos.
const sub = createClient({ url: process.env.REDIS_URL });
const pub = sub.duplicate();
await Promise.all([sub.connect(), pub.connect()]);

// Índice local: sala -> sockets conectados A ESTA instancia.
const rooms = new Map();

function join(room, ws) {
  if (!rooms.has(room)) {
    rooms.set(room, new Set());
    // Un canal por sala. Con un único canal global, un mensaje para tres
    // personas viajaría a las cuarenta instancias para ser descartado.
    sub.subscribe('room:' + room, (raw) => fanout(room, raw));
  }
  rooms.get(room).add(ws);
}

function leave(room, ws) {
  const set = rooms.get(room);
  if (!set) return;
  set.delete(ws);
  if (set.size === 0) {
    rooms.delete(room);
    sub.unsubscribe('room:' + room);
  }
}

function fanout(room, raw) {
  for (const ws of rooms.get(room) ?? []) safeSend(ws, raw);
}

// Cualquier instancia, o un worker sin conexiones abiertas, publica aquí.
export function broadcast(room, event) {
  return pub.publish('room:' + room, JSON.stringify(event));
}
```

El límite que tienes que conocer antes de construir encima: Redis Pub/Sub es *fire and forget*. No persiste nada y no reintenta. Si una instancia pierde la conexión con Redis durante dos segundos, esos mensajes no existen para sus clientes y nadie se entera. Para presencia o cursores da igual. Para un chat, no. Ahí necesitas algo que guarde: Redis Streams, NATS JetStream o Kafka. La siguiente sección usa Streams precisamente por eso.

Sobre las *sticky sessions*: son útiles, pero no son una estrategia de escalado. Si tu instancia guarda por conexión un estado que no puede reconstruir, las necesitas. Si ese estado vive en Redis, no, y estarás mejor sin ellas: con afinidad, un pod que se reinicia deja a todos sus clientes apuntando a una cookie que ya no resuelve a ninguna parte.

4. Reconexión: backoff, jitter y reanudación

Un cliente WebSocket se va a desconectar. No es un caso de error, es funcionamiento normal: túneles, despliegues, cambios de red, el metro. La pregunta útil no es cómo evitarlo, sino qué pasa con los mensajes de los tres segundos que el cliente estuvo fuera.

La respuesta es numerar los eventos. Si cada evento de una sala lleva un identificador monótono y guardas una ventana reciente, "he perdido la conexión" se convierte en "dame lo que me falta desde X". Un Redis Stream acotado hace las dos cosas a la vez.

```
export async function publishEvent(room, event) {
  const id = await pub.xAdd(
    'stream:' + room,
    '*',
    { data: JSON.stringify(event) },
    // MAXLEN ~ 1000: recorte aproximado, mucho más barato que el exacto.
    { TRIM: { strategy: 'MAXLEN', strategyModifier: '~', threshold: 1000 } }
  );
  await pub.publish('room:' + room, JSON.stringify({ id, ...event }));
  return id;
}

// Los ids de Stream son "milisegundos-secuencia": compáralos por partes,
// nunca como texto.
function olderThan(a, b) {
  const [aMs, aSeq] = a.split('-').map(Number);
  const [bMs, bSeq] = b.split('-').map(Number);
  return aMs !== bMs ? aMs < bMs : aSeq < bSeq;
}

export async function replaySince(ws, room, lastId) {
  const key = 'stream:' + room;
  const [oldest] = await pub.xRange(key, '-', '+', { COUNT: 1 });

  // Si el cliente pide algo más antiguo de lo que conservamos no podemos
  // reconstruir su historia. Decírselo es mejor que entregarle un estado
  // incoherente sin que lo sepa.
  if (!oldest || olderThan(lastId, oldest.id)) {
    return safeSend(ws, JSON.stringify({ type: 'resync_required' }));
  }

  for (const entry of await pub.xRange(key, '(' + lastId, '+')) {
    safeSend(ws, JSON.stringify({ id: entry.id, ...JSON.parse(entry.message.data) }));
  }
}
```

En el cliente hacen falta dos cosas: recordar el último id procesado y reconectar con **backoff exponencial y jitter**. El jitter no es un adorno. Si mil clientes caen a la vez porque acabas de desplegar y todos esperan exactamente 500 ms, vuelven todos a la vez y tumban la instancia recién arrancada, que vuelve a caer, y así.

```
class ResilientSocket {
  constructor(url, onEvent) {
    this.url = url;
    this.onEvent = onEvent;
    this.attempt = 0;
    this.lastId = null; // último evento procesado
    this.connect();
  }

  connect() {
    const qs = this.lastId ? '?since=' + encodeURIComponent(this.lastId) : '';
```

```
    this.ws = new WebSocket(this.url + qs);

    this.ws.onopen = () => { this.attempt = 0; };

    this.ws.onmessage = (ev) => {
      const msg = JSON.parse(ev.data);
      if (msg.type === 'resync_required') {
        this.lastId = null;      // el servidor ya no cubre nuestro hueco
        return this.onEvent(msg); // recarga el estado completo
      }
      if (msg.id) this.lastId = msg.id;
      this.onEvent(msg);
    };

    this.ws.onclose = () => {
      // Full jitter: espera un valor aleatorio DENTRO del techo, no el
      // techo. Reparte la avalancha en lugar de sincronizarla.
      const ceiling = Math.min(30_000, 500 * 2 ** this.attempt++);
      setTimeout(() => this.connect(), Math.random() * ceiling);
    };
  }

  send(data) {
    // El cliente también tiene backpressure: si el buffer de subida
    // crece, la red del usuario no da para más.
    if (this.ws.readyState !== WebSocket.OPEN) return false;
    if (this.ws.bufferedAmount > 512 * 1024) return false;
    this.ws.send(JSON.stringify(data));
    return true;
  }
}
```

Con eventos numerados y reconexión escalonada, una desconexión deja de ser una pérdida de datos y pasa a ser un hueco que se rellena solo. Falta el momento en que tú provocas todas las desconexiones a la vez.5. Despliegues: drenar en vez de cortar

5. Despliegues: apagar sin tirar a todo el mundo a la vez

Aquí es donde más servicios en tiempo real se rompen, porque el problema solo se manifiesta en producción y con carga. Cuando Kubernetes termina un pod manda SIGTERM. Si tu proceso simplemente sale, cada cliente ve un cierre anormal (código 1006) y, como todos lo ven en el mismo instante, todos reconectan en el mismo instante contra las instancias que quedan.

```
let draining = false;

process.on('SIGTERM', () => {
  draining = true;   // /ready empieza a devolver 503
  server.close();    // deja de aceptar handshakes nuevos

  const clients = [...wss.clients];
  const windowMs = 10_000; // reparte las reconexiones en 10 segundos

  clients.forEach((ws, i) => {
```

```
    setTimeout(() => {
      // 1001 "going away" le dice al cliente que esto es esperado.
      // Sin este cierre explícito vería un 1006 (cierre anormal).
      ws.close(1001, 'server going away');
    }, (i / clients.length) * windowMs);
  });

  setTimeout(() => {
    for (const ws of wss.clients) ws.terminate();
    process.exit(0);
  }, windowMs + 5_000);
});

app.get('/ready', (_req, res) => res.sendStatus(draining ? 503 : 200));
```

```
spec:
  # Tiene que superar la ventana de drenaje (10 s) más el margen (5 s).
  terminationGracePeriodSeconds: 45
  containers:
    - name: ws-gateway
      lifecycle:
        preStop:
          exec:
            # SIGTERM y la retirada del pod del balanceador ocurren en
            # paralelo. Esta pausa evita cerrar conexiones mientras
            # todavía te están llegando handshakes nuevos.
            command: ["sh", "-c", "sleep 5"]
      readinessProbe:
        httpGet: { path: /ready, port: 8080 }
        periodSeconds: 2
```

El preStop con un sleep corto parece un parche sucio, y lo es, pero resuelve una carrera real: la propagación del cambio de Endpoints a los balanceadores no está sincronizada con el SIGTERM. Sin esa pausa vas a estar cerrando conexiones por un lado mientras el balanceador te sigue mandando clientes nuevos por el otro.

Autenticación y Origin: dos trampas del handshake

El handshake de WebSocket es una petición HTTP, pero con dos particularidades que sorprenden. La primera: el navegador no te deja poner cabeceras personalizadas, así que no puedes mandar un Authorization: Bearer. La tentación es meter el token en la query string, donde acaba en los logs de acceso del proxy, del balanceador y de tu APM. Emite un ticket de un solo uso con 30 segundos de vida y cánjealo en el upgrade.

La segunda es más grave: **el handshake no está protegido por CORS.** Cualquier página puede abrir una conexión a tu servidor y el navegador adjuntará las cookies del usuario. Es *Cross-Site WebSocket Hijacking*, y la única defensa es validar Origin tú mismo.

```
const wss = new WebSocketServer({
  noServer: true,
  verifyClient: ({ origin }) => ALLOWED_ORIGINS.has(origin),
});
```

```
server.on('upgrade', async (req, socket, head) => {
  const url = new URL(req.url, 'http://localhost');
  const userId = await redeemTicket(url.searchParams.get('ticket'));

  if (!userId) {
    socket.write('HTTP/1.1 401 Unauthorized\r\n\r\n');
    return socket.destroy();
  }

  wss.handleUpgrade(req, socket, head, (ws) => {
    ws.userId = userId;
    wss.emit('connection', ws, req);
  });
});
```

Presupuesto de memoria: cuántas conexiones caben de verdad en un proceso

Antes de discutir arquitectura conviene poner números al proceso que ya tienes. La primera pared que se toca casi nunca es la CPU: es el límite de descriptores de fichero. Linux reparte 1.024 por proceso por defecto, y cada conexión aceptada consume uno. Con ese valor tu servidor deja de aceptar conexiones alrededor del millar largo y lo hace con un EMFILE que suele aparecer en los logs como un error de accept sin más contexto.

El arreglo es aburrido y hay que hacerlo en las tres capas donde vive el proceso, porque cambiar solo una no sirve de nada:

```
# 1. La sesión, para pruebas locales.
ulimit -n 1048576

# 2. El servicio, si arrancas con systemd. Esto gana a limits.conf.
# /etc/systemd/system/realtime.service
[Service]
LimitNOFILE=1048576

# 3. El kernel, para que la cola de aceptación no descarte SYNs en un pico.
sysctl -w net.core.somaxconn=65535
sysctl -w net.ipv4.tcp_max_syn_backlog=65535
sysctl -w net.ipv4.ip_local_port_range='10240 65535'
```

El tercer bloque importa sobre todo cuando el que abre muchas conexiones eres tú: un balanceador o un proxy que habla con tu backend agota el rango de puertos efímeros mucho antes de lo que la intuición sugiere. En contenedores, además, comprueba el límite dentro del contenedor y no en el host: `cat /proc/1/limits` desde dentro del pod es la única respuesta que cuenta.

Con los descriptores fuera del camino, el siguiente techo es la memoria, y aquí la aritmética es más útil que cualquier consejo genérico. Una conexión inactiva cuesta el socket del kernel, los buffers de recepción y envío, el objeto de JavaScript que la representa y el estado que tú le cuelgues (usuario, salas, cursores). En un servidor bien ajustado eso ronda los 10 KB por conexión: 100.000 conexiones son aproximadamente 1 GB solo en contabilidad, antes de haber enviado un byte útil. Una conexión *activa* con mensajes pendientes es otra historia, porque ahí entra el buffer de salida del que hablábamos en la sección de backpressure, y ese sí puede

escalar a decenas o cientos de kilobytes cada uno.

Hay un detalle de Node.js que despista a mucha gente al medir: los buffers de socket no viven en el heap de V8. Si vigilas solo heapUsed vas a ver una línea plana mientras el proceso se acerca al OOM. Lo que crece es la memoria externa y el RSS.

```
// Muestreo barato que cabe en cualquier servicio.
setInterval(() => {
  const m = process.memoryUsage();
  const n = wss.clients.size || 1;
  console.log(JSON.stringify({
    conns: n,
    rss_mb: (m.rss / 1048576).toFixed(1),
    heap_mb: (m.heapUsed / 1048576).toFixed(1),
    // El sospechoso habitual cuando el heap no explica el RSS.
    external_mb: (m.external / 1048576).toFixed(1),
    bytes_per_conn: Math.round(m.rss / n)
  }));
}, 15000);
```

La métrica útil de ese objeto es la última: bytes por conexión. Si se mantiene estable mientras suben las conexiones, escalas linealmente y puedes proyectar. Si crece con el tiempo a conexiones constantes, tienes una fuga, y el sitio donde buscarla casi siempre es un mapa de salas del que no borras cuando el socket se cierra.

¿Cuántas conexiones aguanta un proceso? Con la librería ws y mensajes moderados, entre 10.000 y 60.000 es el rango en el que la mayoría de servicios se mueve con comodidad; llegar a 100.000 es posible pero deja poco margen para picos. Ese techo no es de Node.js en abstracto, es de tu carga: veinte mensajes por segundo y por cliente no se parecen en nada a uno cada diez segundos. Por eso el número que debes conocer no es el de un artículo, sino el tuyo, y para eso está la sección de pruebas de carga más abajo.

Un apunte sobre --max-old-space-size: subirlo no ayuda con este problema. Solo mueve el límite del heap de V8, que no es donde está el peso. Si tu proceso muere por OOM con el heap tranquilo, el arreglo está en el buffer de salida y en el número de conexiones por proceso, no en una bandera de arranque.

Serializar una vez, no una vez por cliente

Este es el error de rendimiento más caro y más fácil de cometer en un servidor de tiempo real, y no aparece en ninguna traza de error. Difundir un mensaje a una sala se escribe casi siempre así:

```
// Mal: N llamadas a JSON.stringify para N clientes.
for (const ws of room) {
  ws.send(JSON.stringify(event));
}
```

Con cinco clientes da igual. Con cinco mil, acabas de serializar el mismo objeto cinco mil veces en el mismo tick, bloqueando el event loop mientras lo haces. Un evento de 2 KB difundido a 5.000 conexiones son 10 MB de cadenas creadas y tiradas a la basura por cada publicación. El síntoma no es un error: es un p99 de latencia que sube y un event loop lag que nadie relaciona con la difusión.

```
// Bien: una serialización, N envíos.
const payload = JSON.stringify(event);
for (const ws of room) {
  if (ws.readyState !== ws.OPEN) continue;
  if (ws.bufferedAmount > MAX_BUFFER) { handleSlowClient(ws); continue; }
  ws.send(payload);
}
```

Se puede afinar más. Si conviertes la carga a un Buffer una sola vez, evitas además la conversión de UTF-8 por envío; con ws basta pasar el buffer directamente y marcar el frame como texto:

```
const frame = Buffer.from(JSON.stringify(event), 'utf8');
for (const ws of room) ws.send(frame, { binary: false });
```

El sobre del mensaje, y por qué versionarlo desde el día uno

El formato del mensaje es un contrato con clientes que no controlas: navegadores con la pestaña abierta desde hace tres días, aplicaciones móviles que el usuario no ha actualizado, integraciones de terceros. Si publicas { price: 42 } y mañana necesitas añadir la moneda, no tienes forma de saber quién entiende el formato nuevo.

```
// Sobre estable: la versión y el tipo van fuera; los datos, dentro.
{
  v: 1,            // versión del sobre
  t: 'quote.updated', // tipo de evento, en pasado y con espacio de nombres
  id: '1725...-3', // identificador ordenable para reanudar
  ts: 1757246400123,
  d: { symbol: 'ACME', price: 42, currency: 'EUR' }
}
```

Tres reglas que ahorran migraciones: los campos nuevos siempre son opcionales, los campos viejos nunca cambian de significado, y un cliente que recibe un t desconocido lo ignora en silencio en lugar de romperse. Con eso puedes desplegar un evento nuevo sin coordinar con nadie.

Sobre el formato binario: MessagePack o CBOR recortan entre un 15% y un 40% frente a JSON en cargas con muchas claves repetidas, y Protobuf bastante más si el esquema es fijo. Pero JSON tiene una ventaja difícil de superar en producción: se lee en las herramientas de red del navegador sin instalar nada. La recomendación honesta es empezar con JSON, medir el ancho de banda real por conexión y cambiar solo si esa cifra es un problema de coste o de batería, no porque binario suene más rápido.

Agrupar y colapsar: menos frames, misma información

Cuando la frecuencia sube, el coste deja de estar en el tamaño del mensaje y pasa a estar en el número de frames. Cada send() es una cabecera de frame, una escritura al socket y, con suerte, una llamada al sistema. Un cliente que recibe cuarenta actualizaciones por segundo no las va a pintar cuarenta veces: el navegador refresca a 60 Hz y tu interfaz probablemente agrupa cambios antes de renderizar.

La solución es una bandeja de salida por conexión que se vacía a ritmo fijo, con colapsado por clave para los datos de estado:

```
const FLUSH_MS = 25;

class Outbox {
  constructor(ws) {
    this.ws = ws;
    this.latest = new Map(); // clave -> último valor (colapsa)
    this.queue = [];         // eventos que no se pueden perder
    this.timer = null;
  }

  // Datos de estado: solo importa el último.
  set(key, event) {
    this.latest.set(key, event);
    this.schedule();
  }

  // Hechos: todos cuentan.
  push(event) {
    this.queue.push(event);
    this.schedule();
  }

  schedule() {
    if (this.timer) return;
    this.timer = setTimeout(() => this.flush(), FLUSH_MS);
  }

  flush() {
    this.timer = null;
    if (this.ws.readyState !== this.ws.OPEN) return;

    const batch = [...this.queue, ...this.latest.values()];
    this.queue.length = 0;
    this.latest.clear();
    if (batch.length === 0) return;

    // Si el cliente ya va atrasado, no le añadas más peso.
    if (this.ws.bufferedAmount > MAX_BUFFER) {
      metrics.dropped.inc({ reason: 'backpressure' }, batch.length);
      return;
    }
    this.ws.send(JSON.stringify({ v: 1, t: 'batch', d: batch }));
  }
}
```

Dos detalles que marcan la diferencia. El primero es que el temporizador solo se arma cuando hay algo que enviar, así que una conexión ociosa no paga nada. El segundo es la separación entre set y push: una cotización o una posición de cursor se pueden colapsar sin perder información útil, pero un mensaje de chat o una confirmación de pago no. Mezclar ambos en la misma cola es la forma habitual de perder eventos que importaban.

El efecto medido en servicios reales es grande: pasar de cuarenta frames por segundo y por socket a uno cada 25 ms reduce el trabajo del proceso más de lo que reduce el ancho de banda, porque elimina llamadas al sistema y presión sobre el recolector de basura. Es, con diferencia, el cambio con mejor relación entre esfuerzo y resultado de toda esta guía.

Fan-out a escala: Pub/Sub fragmentado y la aritmética que decide tu factura

El patrón de la sección anterior —una suscripción de Redis por instancia y un índice local de salas— funciona hasta que deja de hacerlo, y conviene saber dónde está esa frontera antes de llegar.

En Redis Pub/Sub clásico, un mensaje publicado se entrega a todos los suscriptores del canal. En Redis Cluster eso significa que el mensaje viaja a *todos* los nodos del cluster, porque cualquiera de ellos podría tener un suscriptor interesado. Con seis nodos y diez mil publicaciones por segundo, tu bus interno mueve sesenta mil mensajes por segundo aunque cada mensaje solo interese a una instancia. Es tráfico que no hace nada y que se paga en CPU de Redis y en red.

El Pub/Sub fragmentado, disponible desde Redis 7.0 con SSUBSCRIBE y SPUBLISH, ata cada canal a la ranura hash que le corresponde y entrega solo en ese shard. El cambio en el código es pequeño; el efecto en el gasto de Redis, no:

```
import { createClient } from 'redis';

const pub = createClient({ url: REDIS_URL });
const sub = pub.duplicate();
await Promise.all([pub.connect(), sub.connect()]);

// Antes: publish/subscribe. Con cluster, esto habla con todos los nodos.
// Ahora: sPublish/sSubscribe. El mensaje no sale del shard del canal.
async function joinRoom(room, ws) {
  const channel = 'room:' + room;
  if (!local.has(channel)) {
    local.set(channel, new Set());
    await sub.sSubscribe(channel, (raw) => deliverLocal(channel, raw));
  }
  local.get(channel).add(ws);
}

async function leaveRoom(room, ws) {
  const channel = 'room:' + room;
  const set = local.get(channel);
  if (!set) return;
  set.delete(ws);
  // Sin esta línea acumulas suscripciones muertas hasta que Redis se queja.
  if (set.size === 0) {
    local.delete(channel);
    await sub.sUnsubscribe(channel);
  }
}
```

La otra decisión de diseño con impacto real es la granularidad del canal. Hay dos extremos y un punto sensato entre ellos. Un canal global por el que pasa todo obliga a cada instancia a recibir y descartar el 90% del tráfico; es simple y funciona bien hasta unos pocos miles de mensajes por segundo. Un canal por sala hace que cada instancia reciba solo lo suyo, pero con cien mil salas activas tienes cien mil suscripciones repartidas por el cluster, y las operaciones de suscripción y desuscripción se convierten en el cuello de botella cuando los usuarios entran y salen a ritmo alto.

El punto intermedio que aguanta mejor es agrupar salas en un número fijo de canales por hash: room:' +

(hash(room) % 256). Cada instancia mantiene 256 suscripciones estables, entrar en una sala no toca Redis en absoluto, y el tráfico desperdiciado se divide por 256 respecto al canal único. Es la solución aburrida y suele ser la correcta.

```
const SHARDS = 256;
const shardOf = (room) => 'fan:' + (xxhash32(room) % SHARDS);

// Al arrancar, una vez. No hay suscripciones dinámicas en el camino caliente.
for (let i = 0; i < SHARDS; i++) {
  await sub.sSubscribe('fan:' + i, onShardMessage);
}

function onShardMessage(raw) {
  const { room, payload } = JSON.parse(raw);
  const set = localRooms.get(room);
  if (!set) return; // no es para esta instancia: descartar es barato
  for (const ws of set) trySend(ws, payload);
}
```

Hay una trampa aritmética que conviene ver escrita. Si una sala tiene 50.000 miembros repartidos entre 10 instancias y publicas 20 mensajes por segundo, Redis mueve 200 mensajes por segundo (20 × 10) pero tus procesos ejecutan un millón de send() por segundo entre todos. El bus rara vez es el cuello de botella: lo es el último tramo. Cuando alguien propone cambiar Redis por Kafka o NATS para “aguantar el fan-out”, la pregunta correcta es cuál de los dos números está en rojo, porque cambiar el bus no toca el que suele estarlo.

Entrega fiable: Pub/Sub no es una cola

Merece la pena insistir en esto porque es la confusión que más datos hace desaparecer sin dejar rastro. Redis Pub/Sub entrega a quien esté escuchando *en ese instante*. Si tu instancia estaba reiniciándose, si el suscriptor perdió la conexión medio segundo, si el proceso estaba ocupado: el mensaje no se reintenta ni se guarda. No hay error, no hay métrica, no hay nada. Simplemente no llegó.

Para todo lo que el usuario pueda echar en falta —un mensaje de chat, una notificación, un cambio de estado de un pedido— hace falta un log persistente. Redis Streams cubre ese papel sin añadir una pieza nueva a la arquitectura, y encaja de forma natural con la reanudación por identificador que ya viste en la sección de reconexión.

```
// Publicar: el stream es a la vez bus y log de reanudación.
await pub.xAdd('room:' + room, '*',
  { type: event.t, data: JSON.stringify(event.d) },
  // Retención por número de entradas, aproximada (mucho más barata que exacta).
  { TRIM: { strategy: 'MAXLEN', strategyModifier: '~', threshold: 10000 } }
);
```

La retención se decide con una cuenta sencilla: cuánto tiempo quieres poder reanudar, multiplicado por tu ritmo de eventos. Si publicas 5 eventos por segundo en una sala y quieres cubrir una desconexión de treinta minutos, necesitas 9.000 entradas; con MAXLEN ~ 10000 vas holgado. El modificador ~ permite a Redis recortar por nodos completos del árbol en lugar de entrada a entrada, y la diferencia de coste es notable en streams con mucho tráfico.

Cuando además el trabajo derivado del evento no se puede perder —enviar un correo, cobrar, actualizar un índice—, entra en juego el grupo de consumidores. Cada mensaje entregado queda en la lista de pendientes del consumidor hasta que lo confirma con XACK. Si el proceso muere entre la entrega y la confirmación, el mensaje sigue ahí, invisible para el resto del grupo, hasta que alguien lo reclame:

```
const GROUP = 'fanout';
const ME = process.env.POD_NAME;

// Camino normal: leer lo que nadie ha visto todavía.
async function consume() {
  const res = await redis.xReadGroup(GROUP, ME,
    [{ key: STREAM, id: '>' }], { COUNT: 100, BLOCK: 5000 });
  for (const msg of res?.[0]?.messages ?? []) {
    await handle(msg);
    await redis.xAck(STREAM, GROUP, msg.id);
  }
}

// Camino de recuperación: reclamar lo que un compañero muerto dejó a medias.
// Reclamar y procesar en el mismo paso; si solo reclamas, el problema se mueve.
async function reapIdlePel() {
  let cursor = '0-0';
  do {
    const { nextCursor, messages } = await redis.xAutoClaim(
      STREAM, GROUP, ME, 60000 /* ms ocioso */, cursor, { COUNT: 50 });
    for (const msg of messages) {
      await handle(msg);           // handle DEBE ser idempotente
      await redis.xAck(STREAM, GROUP, msg.id);
    }
    cursor = nextCursor;
  } while (cursor !== '0-0');
}

setInterval(reapIdlePel, 15000);
```

Los dos detalles que deciden si esto funciona: xAutoClaim solo reclama entradas cuyo tiempo ocioso supera el umbral, así que un consumidor lento pero vivo no se ve robado; y reclamar y procesar tienen que ir juntos, porque si solo reclamas, las entradas pasan a tu lista de pendientes y el problema se ha movido de sitio sin resolverse. Como el mismo mensaje puede procesarse dos veces —el proceso puede morir justo después de handle y antes de xAck—, el manejador tiene que ser idempotente. Aquí la idempotencia no es una buena práctica opcional: es el precio de entrada.

Confirmaciones de aplicación para lo que sube el cliente

El sentido contrario tiene el mismo problema y se resuelve igual. Cuando el cliente envía algo que no puede perderse, el send() del navegador no garantiza nada: entrega al buffer local, y si la conexión cae en ese momento el mensaje se evapora sin error.

```
// Cliente: identificador propio, reintento hasta confirmación.
const pending = new Map();

function sendReliable(type, data) {
```

```
  const id = crypto.randomUUID();
  const msg = { v: 1, t: type, id, d: data };
  pending.set(id, msg);
  if (ws.readyState === WebSocket.OPEN) ws.send(JSON.stringify(msg));
  return id;
}

ws.addEventListener('message', (e) => {
  const m = JSON.parse(e.data);
  if (m.t === 'ack') pending.delete(m.d.id);
});

// Al reconectar, reenviar lo no confirmado. El servidor deduplica por id.
ws.addEventListener('open', () => {
  for (const msg of pending.values()) ws.send(JSON.stringify(msg));
});
```

El servidor guarda los identificadores vistos en una clave de Redis con caducidad (unas horas basta) y responde con la misma confirmación si el mensaje ya se procesó. Es el mismo patrón de clave de idempotencia que se usa en APIs de pago, aplicado a un canal donde los reintentos no son la excepción sino lo normal.

Autorización por sala: el cliente nunca decide a qué se suscribe

El handshake te dice *quién* es el usuario. No te dice a qué tiene derecho. Es una distinción que se pierde con facilidad porque el código de suscripción suele nacer en una demo, donde el servidor hace lo que el cliente le pide:

```
// El fallo de autorización más común en servicios de tiempo real.
ws.on('message', (raw) => {
  const msg = JSON.parse(raw);
  if (msg.t === 'subscribe') rooms.join(msg.room, ws); // cualquiera, a cualquier sala
});
```

Con eso, un usuario autenticado del tenant A escribe {"t":"subscribe","room":"org:B:alerts"} en la consola del navegador y empieza a recibir el tráfico del tenant B. No hay exploit, no hay cadena de vulnerabilidades: es la funcionalidad tal como está escrita. Y como el canal no pasa por tu middleware HTTP, ninguna de las protecciones que sí tienes en la API REST se aplica aquí.

La regla es sencilla: cada suscripción es una decisión de autorización, igual que cada endpoint. Y como se ejecuta en el camino caliente, conviene que tenga caché con vida corta:

```
const ROOM_RE = /^(chat|alerts|board):[a-z0-9-]{1,64}$/;
const authzCache = new Map(); // 'userId|room' -> { ok, exp }
const AUTHZ_TTL_MS = 30000;

async function canSubscribe(user, room) {
  if (!ROOM_RE.test(room)) return false;

  const key = user.id + '|' + room;
  const hit = authzCache.get(key);
```

```
  if (hit && hit.exp > Date.now()) return hit.ok;

  const ok = await db.userCanRead(user.id, room); // tu lógica real
  authzCache.set(key, { ok, exp: Date.now() + AUTHZ_TTL_MS });
  return ok;
}

ws.on('message', async (raw) => {
  const msg = safeParse(raw);
  if (!msg) return ws.close(1003, 'bad payload');

  if (msg.t === 'subscribe') {
    if (ws.rooms.size >= MAX_ROOMS_PER_CONN) {
      return send(ws, { t: 'error', d: { code: 'too_many_rooms' } });
    }
    if (!(await canSubscribe(ws.user, msg.room))) {
      // No distingas "no existe" de "no tienes permiso": filtra información.
      return send(ws, { t: 'error', d: { code: 'forbidden' } });
    }
    rooms.join(msg.room, ws);
    send(ws, { t: 'subscribed', d: { room: msg.room } });
  }
});
```

La expresión regular no es decorativa. Sin ella, un nombre de sala es una cadena arbitraria que acabará concatenada en una clave de Redis y, en algunos diseños, en una consulta. Validar la forma antes de usarla cierra esa puerta y además evita que un cliente cree un millón de salas vacías con nombres aleatorios y te llene el mapa de salas.

Revocación en caliente: el permiso que caducó a mitad de la conexión

Una conexión WebSocket puede vivir horas. Si expulsas a alguien de un proyecto, su sesión HTTP deja de funcionar en el siguiente ratón; su WebSocket sigue recibiendo eventos hasta que se desconecte por su cuenta. Es el equivalente en tiempo real del token que no se puede revocar.

La solución que menos código nuevo introduce es un canal de control por el que viajan las revocaciones, consumido por todas las instancias:

```
// Publicador: al cambiar permisos, avisa a todo el cluster.
await pub.publish('control', JSON.stringify({
  t: 'revoke', userId, room
}));

// Cada instancia, en su suscriptor de control:
sub.subscribe('control', (raw) => {
  const m = JSON.parse(raw);
  if (m.t !== 'revoke') return;

  for (const ws of rooms.membersOf(m.room)) {
    if (ws.user.id !== m.userId) continue;
    rooms.leave(m.room, ws);
    authzCache.delete(m.userId + '|' + m.room);
    send(ws, { t: 'unsubscribed', d: { room: m.room, reason: 'revoked' } });
  }
```

```
});
```

Fíjate en que la revocación borra también la entrada de la caché de autorización. Sin esa línea, el cliente vuelve a suscribirse un segundo después y la caché le dice que sí durante los treinta segundos que le quedaban de vida. Es un fallo que no se ve en pruebas manuales porque la ventana es corta, y que en producción significa que la revocación no funciona la mitad de las veces.

Para el caso multi-tenant, la disciplina que evita casi todos los incidentes es que el identificador de tenant nunca venga del mensaje: se toma del contexto autenticado de la conexión y se antepone al nombre de sala en el servidor. Si el cliente puede escribir el tenant, el aislamiento depende de que nadie se equivoque nunca al validar.

Límites de uso en el canal: el abuso no entra por la API

Todos los servicios tienen rate limiting en la API HTTP. Muy pocos lo tienen en el canal WebSocket, y sin embargo ahí un atacante no paga el coste de un handshake por petición: abre una conexión y envía cien mil mensajes por el mismo túnel. Hacen falta límites en tres puntos distintos, porque protegen de cosas distintas.

En el handshake, para que nadie abra diez mil conexiones desde una IP y agote tus descriptores. **Por conexión**, para que un cliente no sature el event loop con mensajes entrantes. **Por carga útil**, para que un solo frame no reserve cien megabytes antes de que puedas rechazarlo.

```
import { WebSocketServer } from 'ws';

const wss = new WebSocketServer({
  noServer: true,
  // Un frame más grande que esto cierra la conexión con 1009.
  maxPayload: 64 * 1024,
  perMessageDeflate: false
});

// Cubo de fichas por conexión: 20 mensajes/s de media, ráfagas de 50.
class TokenBucket {
  constructor(rate, burst) {
    this.rate = rate; this.burst = burst;
    this.tokens = burst; this.last = Date.now();
  }
  take() {
    const now = Date.now();
    this.tokens = Math.min(this.burst, this.tokens + ((now - this.last) / 1000) * this.rate);
    this.last = now;
    if (this.tokens < 1) return false;
    this.tokens -= 1;
    return true;
  }
}

wss.on('connection', (ws) => {
  ws.bucket = new TokenBucket(20, 50);
  ws.strikes = 0;

  ws.on('message', (raw) => {
```

```
    if (!ws.bucket.take()) {
      metrics.throttled.inc();
      // Tres avisos y fuera: cerrar al primero castiga a redes malas.
      if (++ws.strikes > 3) return ws.close(1008, 'rate limit');
      return send(ws, { t: 'error', d: { code: 'rate_limited' } });
    }
    handle(ws, raw);
  });
});
```

El detalle de los tres avisos no es indulgencia: un móvil que sale de un túnel puede vaciar de golpe su cola de mensajes pendientes y disparar el límite sin mala intención. Cerrar a la primera convierte una mala cobertura en un bucle de reconexión.

En el handshake, el límite por IP se aplica antes de aceptar el upgrade, y conviene contar conexiones vivas, no intentos:

```
const perIp = new Map();
const MAX_PER_IP = 50;

server.on('upgrade', (req, socket, head) => {
  const ip = req.headers['x-forwarded-for']?.split(',')[0].trim() || req.socket.remoteAddress;
  const n = perIp.get(ip) || 0;
  if (n >= MAX_PER_IP) {
    socket.write('HTTP/1.1 429 Too Many Requests\r\n\r\n');
    return socket.destroy();
  }
  // Un socket a medio negociar también consume recursos.
  socket.setTimeout(10000, () => socket.destroy());

  authenticate(req).then((user) => {
    if (!user) { socket.write('HTTP/1.1 401 Unauthorized\r\n\r\n'); return socket.destroy(); }
    wss.handleUpgrade(req, socket, head, (ws) => {
      perIp.set(ip, n + 1);
      ws.on('close', () => {
        const c = (perIp.get(ip) || 1) - 1;
        c <= 0 ? perIp.delete(ip) : perIp.set(ip, c);
      });
      ws.user = user;
      wss.emit('connection', ws, req);
    });
  });
});
```

El setTimeout sobre el socket cubre un ataque que casi nadie prueba: abrir conexiones TCP, empezar el upgrade y no terminarlo nunca. Sin ese temporizador, cada intento a medias se queda ocupando un descriptor de forma indefinida. Y el borrado del contador en el evento close es igual de importante que el límite: sin él, el mapa perIp es una fuga de memoria que además acaba bloqueando a usuarios legítimos.

Observabilidad: seis métricas y por qué esas

Un servicio de tiempo real mal instrumentado se diagnostica por quejas de usuarios, y las quejas llegan tarde y

mal descritas (“a veces no me llegan los mensajes”). Con seis series puedes explicar casi cualquier incidente sin abrir un depurador.

- **Conexiones activas** (gauge, por instancia). La forma de la curva dice más que el número: un escalón hacia abajo es un despliegue o una caída; una pendiente sostenida hacia arriba sin que suban los usuarios es una fuga de conexiones zombis.
- **Cierres por causa** (counter con etiqueta reason: normal, heartbeat, backpressure, rate_limit, shutdown). Es la métrica que convierte “se desconectan” en un problema concreto.
- **Bytes en el buffer de salida** (histograma, muestreado). Te da la distribución de clientes lentos. El p50 es siempre cero; lo que importa es el p999.
- **Mensajes descartados** (counter con etiqueta reason). Si descartas por política, tienes que saber cuánto.
- **Latencia de publicación a entrega** (histograma). La única métrica que mide lo que el usuario percibe.
- **Lag del event loop** (histograma). En Node.js es el indicador temprano de casi todo: serialización por cliente, JSON gigantes, trabajo síncrono en el camino de difusión.

```
import client from 'prom-client';
import { monitorEventLoopDelay } from 'perf_hooks';

const connections = new client.Gauge({
  name: 'ws_connections_active', help: 'Conexiones abiertas'
});
const closes = new client.Counter({
  name: 'ws_closes_total', help: 'Cierres por causa', labelNames: ['reason']
});
const outbuf = new client.Histogram({
  name: 'ws_send_buffer_bytes', help: 'bufferedAmount muestreado',
  buckets: [0, 4096, 65536, 262144, 1048576, 8388608, 33554432]
});
const delivery = new client.Histogram({
  name: 'ws_publish_to_deliver_seconds', help: 'Publicación a entrega',
  buckets: [0.005, 0.01, 0.05, 0.1, 0.25, 0.5, 1, 2, 5]
});

// Lag del event loop, sin coste apreciable.
const h = monitorEventLoopDelay({ resolution: 20 });
h.enable();
setInterval(() => {
  eventLoopP99.set(h.percentile(99) / 1e6); // ms
  h.reset();
}, 10000);

// Muestrear el buffer: recorrer 100k sockets cada segundo no sale gratis.
setInterval(() => {
  let i = 0;
  for (const ws of wss.clients) {
    if (i++ % 50 !== 0) continue; // una de cada cincuenta
    outbuf.observe(ws.bufferedAmount);
  }
}, 10000);
```

La regla de cardinalidad es innegociable: **nunca** etiquetes por identificador de usuario, de sala o de conexión.

Una etiqueta room con diez mil salas multiplica por diez mil cada serie, y el que se cae no es tu servicio sino Prometheus. Si necesitas ese detalle, va en un log estructurado o en una traza, no en una métrica.

El canario: medir la entrega de verdad, no la publicación

La latencia que se suele medir es la del servidor: cuánto tarda en publicar. Esa cifra puede estar perfecta mientras los usuarios no reciben nada, porque el problema está en el fan-out, en la red o en un cliente atascado. La forma barata de medir la cadena completa es un canario: un cliente sintético, desplegado como cualquier otro servicio, que se conecta como un usuario real y mide el viaje de ida y vuelta.

```
// Canario: publica por HTTP, espera por WebSocket, exporta la diferencia.
setInterval(async () => {
  const nonce = crypto.randomUUID();
  const t0 = performance.now();

  const done = new Promise((resolve, reject) => {
    const timer = setTimeout(() => reject(new Error('timeout')), 5000);
    canaryWs.once('message', (raw) => {
      if (JSON.parse(raw).d?.nonce !== nonce) return;
      clearTimeout(timer); resolve();
    });
  });

  await fetch(API + '/canary', { method: 'POST', body: JSON.stringify({ nonce }) });

  try {
    await done;
    delivery.observe((performance.now() - t0) / 1000);
  } catch {
    canaryFailures.inc();
  }
}, 10000);
```

Un canario por región y por instancia detecta el caso que ninguna otra métrica ve: una instancia sana según su propio criterio, con conexiones abiertas y CPU baja, que ha dejado de recibir del bus de mensajes porque su suscriptor de Redis se cayó y nadie lo reconectó. Desde dentro todo va bien; desde fuera, el 25% de tus usuarios está mirando datos congelados.

Balanceadores y proxies: los timeouts que cortan conexiones sanas

Muchos problemas de estabilidad que se atribuyen al código son en realidad un temporizador de un componente intermedio que nadie miró. Una conexión WebSocket, para un proxy, es una conexión HTTP que lleva mucho rato sin bytes. Si su timeout de inactividad es menor que tu intervalo de heartbeat, el proxy la cierra y tu servidor ni se entera hasta que intenta escribir.

Los valores por defecto que más incidentes causan, a septiembre de 2026:

- **AWS Application Load Balancer**: 60 segundos de inactividad por defecto, configurable hasta 4.000. En Kubernetes se ajusta con la anotación alb.ingress.kubernetes.io/ load- balancer- attributes: idle_timeout.timeout_seconds=600.
- **nginx**: proxy_read_timeout y proxy_send_timeout valen 60 segundos. Es el motivo número uno de

desconexiones a los 60 segundos exactos en despliegues con ingress-nginx.

- **Cloudflare**: cierra conexiones WebSocket inactivas alrededor de los 100 segundos. No es configurable en los planes habituales, así que tu heartbeat tiene que ser más rápido, punto.
- **Envoy / Istio**: stream_idle_timeout de 5 minutos y route.timeout de 15 segundos; este último hay que ponerlo a cero explícitamente para rutas de WebSocket o cortará la conexión a los quince segundos.

De ahí sale una regla que conviene escribir en el repositorio: **el intervalo de heartbeat debe ser menor que el timeout de inactividad más pequeño de todo el camino, con margen para una pérdida de paquete.** Si el eslabón más corto es Cloudflare con 100 segundos, un ping cada 30 deja sitio a que uno se pierda sin consecuencias. Un ping cada 90 no.

```
# Ingress con ALB: los tres valores tienen que contarse juntos.
metadata:
  annotations:
    alb.ingress.kubernetes.io/load-balancer-attributes: idle_timeout.timeout_seconds=600
    alb.ingress.kubernetes.io/target-group-attributes: |
      stickiness.enabled=true,
      stickiness.type=lb_cookie,
      deregistration_delay.timeout_seconds=120
```

El deregistration_delay es el que casi nadie ajusta y el que decide si un despliegue corta conexiones vivas. Por defecto son 300 segundos, lo que suena generoso, pero si tu terminationGracePeriodSeconds es de 30, el pod muere mucho antes de que el balanceador termine de drenarlo. Los dos números tienen que contarse en el mismo sentido: primero drena el balanceador, luego muere el proceso.

Sobre las sticky sessions conviene ser preciso, porque generan confusión. Una vez completado el upgrade, la conexión TCP ya está fijada a un destino: no puede migrar. La afinidad importa para el *handshake*, no para el tráfico posterior, y es útil sobre todo cuando el cliente hace una petición HTTP previa cuyo estado necesita encontrar en la misma instancia. No es, y nunca ha sido, una estrategia de escalado: si tu diseño necesita que un usuario vuelva siempre a la misma instancia para funcionar, lo que tienes es estado local sin replicar, y eso se rompe en el primer reinicio.

Pruebas de carga: conocer tu techo antes que producción

Todos los números de esta guía son orientativos. El único que sirve para tomar decisiones es el de tu servicio con tu carga, y conseguirlo cuesta una tarde. Lo que distingue una prueba útil de un número inventado es la forma de la rampa y las métricas que miras mientras tanto.

Subir de golpe a la carga máxima no mide tu capacidad: mide tu comportamiento ante un ataque. Node.js además tarda unos segundos en calentar el JIT, así que los primeros datos siempre mienten. Una rampa por etapas con una meseta larga es lo que refleja producción.

```
// k6: rampa por etapas, ping/pong de aplicación y medida del viaje completo.
import ws from 'k6/ws';
import { check } from 'k6';
import { Trend, Rate } from 'k6/metrics';

const rtt = new Trend('app_rtt_ms');
```

```
const lost = new Rate('lost_messages');

export const options = {
  stages: [
    { duration: '2m', target: 5000 },   // subida suave
    { duration: '10m', target: 5000 },  // meseta: aquí se mide
    { duration: '2m', target: 20000 },  // segundo escalón
    { duration: '10m', target: 20000 },
    { duration: '2m', target: 0 }       // bajada: evita conexiones colgadas
  ],
  thresholds: {
    'app_rtt_ms': ['p(99)<500'],
    'lost_messages': ['rate<0.001']
  }
};

export default function () {
  ws.connect(__ENV.WS_URL, { headers: { Authorization: 'Bearer ' + __ENV.TOKEN } }, (socket) => {
    socket.on('open', () => {
      socket.send(JSON.stringify({ v: 1, t: 'subscribe', room: 'load:' + (__VU % 200) }));
      socket.setInterval(() => {
        socket.send(JSON.stringify({ v: 1, t: 'echo', id: String(Date.now()) }));
      }, 5000);
    });

    socket.on('message', (raw) => {
      const m = JSON.parse(raw);
      if (m.t !== 'echo') return;
      rtt.add(Date.now() - Number(m.id));
      lost.add(false);
    });

    socket.setTimeout(() => socket.close(), 300000);
  });
}
```

La bajada final no es decorativa: sin ella terminas la prueba dejando decenas de miles de conexiones a medio cerrar y el informe queda contaminado.

Mientras corre la prueba, lo que se mira no es si aguantó, sino cuatro cosas a la vez: el **p99 del viaje de ida y vuelta**, la **distribución de** bufferedAmount (si el p999 empieza a subir, tu techo está más cerca de lo que parece), el **RSS a carga constante** (si crece, hay fuga y el resultado no es proyectable) y el **lag del event loop**. Un servicio que aguanta 50.000 conexiones con el event loop a 300 ms de p99 no aguanta 50.000 conexiones: está a un pico de distancia de dejar de responder.

La prueba que casi nadie hace: la tormenta de reconexión

El escenario que tumba servicios reales no es la carga estable, es la recuperación. Con la prueba en su meseta, mata una instancia que sostiene el 25% de las conexiones y observa. Todos esos clientes van a reconectar, y si tu backoff no lleva jitter lo van a hacer a la vez, contra tres instancias que ya estaban ocupadas.

Lo que hay que verificar en ese momento: que el pico de conexiones nuevas por segundo se reparte en el tiempo en lugar de concentrarse, que las instancias supervivientes no superan su presupuesto de memoria al absorber el reparto, y que la reanudación por identificador devuelve los eventos perdidos en lugar de dejar

huecos silenciosos. Es una prueba de veinte minutos que ahorra un incidente de madrugada.

Elegir el servidor: ws, uWebSockets.js o Socket.IO

La elección de librería importa menos de lo que parece al principio y más de lo que parece cuando ya tienes cien mil conexiones. Conviene entender qué compra cada una.

ws es la opción por defecto sensata. Implementa el protocolo y poco más, se integra con el servidor HTTP de Node, y su API es la que verás en la mayoría de ejemplos y respuestas. Su límite práctico está en las decenas de miles de conexiones por proceso con mensajes moderados. Que sea la más lenta de las tres no la hace mala elección: la mayoría de servicios nunca llega a notarlo, y la previsibilidad vale mucho.

uWebSockets.js es una implementación en C++ con enlaces para Node. Los números que se publican van de 3 a 10 veces más conexiones y caudal según la prueba, y en escenarios exigentes se han medido del orden de 120.000 conexiones recibiendo un mensaje de 1 KB por segundo cada una con la CPU del proceso al 70%. El precio es real: la API es distinta, no se integra con Express ni Fastify de la forma habitual, la comunidad es más pequeña y hay menos ejemplos que copiar cuando algo falla a las tres de la mañana. Es una migración que se justifica con una medición, no con un artículo.

Socket.IO no es WebSocket: es un protocolo propio que *usa* WebSocket como transporte cuando puede y cae a long polling cuando no. A cambio te da salas, confirmaciones, reconexión automática y adaptadores para escalar horizontalmente sin escribir el fan-out a mano. Si estás construyendo algo donde esas piezas son el 80% del trabajo, ahorra semanas. Los costes a tener presentes son que el cliente tiene que ser el suyo (nada de new WebSocket() desde otro lenguaje sin una librería compatible), que el overhead por mensaje es mayor, y que la capa de compatibilidad complica el diagnóstico cuando el problema es de red.

El criterio que funciona: empieza con ws si controlas cliente y servidor y el protocolo es simple; empieza con Socket.IO si necesitas sus primitivas ya y tu escala es moderada; migra a uWebSockets.js cuando tengas una gráfica que demuestre que el proceso es el cuello de botella y no el diseño. Cambiar de librería para arreglar un problema de arquitectura es la forma más cara de no arreglarlo.

Cuándo no usar WebSockets

La mitad del coste operativo de esta guía desaparece si tu caso de uso no necesita un canal bidireccional. Merece la pena hacerse la pregunta antes, no después.

Si los datos solo van del servidor al cliente —notificaciones, un panel que se actualiza, el progreso de un trabajo, tokens de un modelo de lenguaje—, los Server-Sent Events son casi siempre mejor opción. Van sobre HTTP normal, atraviesan proxies sin configuración especial, el navegador reconecta solo y reanuda con la cabecera Last-Event-ID sin que escribas una línea, y sobre HTTP/2 o HTTP/3 no sufren el viejo límite de seis conexiones por origen. Renuncias al canal de subida, que en estos casos no necesitabas: para eso ya tienes tu API.

Si la frecuencia es baja y la latencia no es crítica, una petición cada pocos segundos es más barata de operar que cualquier conexión persistente. No tiene heartbeats, ni reconexión, ni estado que drenar en cada despliegue. Suena poco elegante y es la respuesta correcta con más frecuencia de la que se admite.

WebSockets gana cuando el tráfico es bidireccional y frecuente: colaboración en vivo, edición compartida,

juegos, mensajería, trading.

WebTransport es la novedad con más recorrido. Va sobre HTTP/3 y QUIC, ofrece múltiples flujos independientes sin bloqueo de cabecera de línea y, sobre todo, datagramas no fiables: para posiciones de cursor o telemetría, donde un paquete perdido no importa porque el siguiente lo corrige, es exactamente la primitiva que faltaba. Alcanzó estado Baseline a mediados de 2026, cuando Safari 26.4 lo incorporó el 24 de marzo. El obstáculo hoy es el servidor: Node.js no trae cliente ni servidor de WebTransport, y la mayor parte de lo que hay en producción se apoya en Go con quic-go y webtransport-go. Si tu backend es Node, WebTransport todavía es un proyecto, no una elección.

Una aclaración que ahorra tiempo: WebSocket sobre HTTP/2 está especificado (RFC 8441) y sobre HTTP/3 también (RFC 9220), pero a día de hoy ningún navegador ni servidor importante lo ha llevado a producción; Chrome sigue en fase de prototipo y Firefox no ha anunciado implementación. Si alguien propone "WebSockets sobre HTTP/3" como solución a un problema tuyo, la respuesta corta es que todavía no existe donde importa.

Caso práctico: de 8.000 conexiones inestables a 120.000 estables

Lo que sigue es un caso compuesto, con la forma y el orden en que estos problemas aparecen de verdad. Los números son representativos de un servicio de paneles en vivo sobre cuatro instancias de Node detrás de un ALB, no una medición de laboratorio.

Punto de partida. 8.000 conexiones concurrentes en hora punta. Tres síntomas: el RSS de cada instancia sube durante el día hasta que el orquestador la mata por memoria, alrededor de una vez al día; el p99 de entrega ronda los 4 segundos en el pico; y cada despliegue genera una ventana de unos 90 segundos de errores de conexión en el cliente.

Semana 1 — Instrumentar antes de tocar nada. Se añaden las seis métricas y un canario. El histograma de bufferedAmount revela lo que nadie sospechaba: el 0,4% de las conexiones acumula más de 30 MB pendientes. Con 8.000 conexiones eso son 32 sockets sosteniendo casi 1 GB entre todos. El origen: un cliente corporativo con una VPN lenta y un panel abierto en una pantalla de televisión.

Semana 2 — Serializar una vez y aplicar política de descarte. Se saca JSON.stringify del bucle y se añade el límite de buffer con colapsado por clave. El RSS pasa de subir 200 MB por hora a mantenerse plano; la memoria en pico baja de 6 GB a 900 MB por instancia. La latencia mejora de rebote, porque el event loop deja de pasarse cientos de milisegundos serializando en cada difusión.

Semana 2 — Heartbeats alineados con el balanceador. Auditando el camino se descubre que el ALB tenía el timeout de inactividad por defecto de 60 segundos y el heartbeat de aplicación estaba en 45, pero el ping solo se enviaba si no había habido tráfico en ese periodo, lo que en la práctica dejaba huecos por encima del minuto. Se fija el ping a 30 segundos incondicional y el timeout del ALB a 600. Las conexiones fantasma —abiertas en el servidor, muertas en el cliente— caen del 12% al 0,3% del total.

Semana 3 — Agrupar frames. Se introduce la bandeja de salida con vaciado cada 25 ms. Los sockets pasan de recibir unos 40 frames por segundo a 6 o 7, con la misma información. La CPU media del proceso baja del 78% al 31%. Este es el cambio que más capacidad libera de todos.

Semana 4 — Despliegues sin ventana de errores. Se añade preStop con 15 segundos de drenaje, terminationGracePeriodSeconds a 45, deregistration_delay a 120, un cierre con código de aplicación que le dice

al cliente que reconecte, y jitter en el backoff. La ventana de 90 segundos de errores se reduce a reconexiones escalonadas que el usuario no percibe.

Semana 5 — Fan-out por shards. Redis estaba al 65% de CPU con un canal por sala y 30.000 salas activas, casi todo en operaciones de suscripción y desuscripción al entrar y salir de paneles. Se sustituye por 256 canales fijos con sSubscribe y enrutado local. La CPU de Redis baja al 11% y el ruido en las gráficas desaparece.

Resultado. Las mismas cuatro instancias sostienen 120.000 conexiones en una prueba de carga con p99 de entrega de 180 ms, y el servicio lleva meses sin un reinicio por memoria. Ninguno de los cambios fue difícil por separado. Lo difícil fue el orden: sin las métricas de la primera semana, el equipo llevaba meses discutiendo si el problema era Redis, y no lo era.

Errores comunes

- **Activar** permessage-deflate **"porque ahorra ancho de banda".** En ws viene desactivado por una razón: mantiene un contexto de zlib por conexión y, con concurrencia alta en Linux, provoca fragmentación de memoria severa. Actívalo solo si has medido tu carga real y compensa.
- **Autoescalar por CPU.** Un gateway de WebSockets se satura de memoria, descriptores de fichero y ancho de banda mucho antes que de CPU. Escala por número de conexiones activas; si escalas por CPU, tu HPA no se enterará de nada hasta que los pods empiecen a morir por OOM.
- **No poner** maxPayload**.** Sin límite, un cliente puede obligarte a reservar cientos de megas con un único frame.
- **Tratar Redis Pub/Sub como si fuera una cola.** No lo es: no hay persistencia, ni acks, ni reintentos. Si el mensaje importa, usa Streams, JetStream o Kafka.
- **Etiquetar métricas por usuario o por conexión.** Te revienta la cardinalidad en Prometheus. Mide agregados: conexiones activas, histograma de bufferedAmount, RTT del ping, contador por código de cierre y ratio de resync_required —esta última es tu mejor alarma temprana de que la ventana de replay se te ha quedado corta.
- **Probar solo en local.** En localhost no hay latencia, ni proxies con idle timeout, ni redes móviles que cambian de IP. Casi todo lo de esta guía es invisible hasta que hay un balanceador de por medio.

Y ocho más que solo aparecen a escala

- **Serializar dentro del bucle de difusión.** Un JSON.stringify por cliente convierte una publicación en trabajo cuadrático encubierto. Es el error de rendimiento más caro de esta lista.
- **No borrar del mapa de salas al cerrar.** El socket muere, la referencia se queda. La fuga es lenta y constante, y se manifiesta como bytes por conexión que crecen sin explicación.
- **Etiquetar métricas por sala o por usuario.** Funciona en local con tres salas. Con diez mil, el que se cae es Prometheus.
- **Confiar en el nombre de sala que envía el cliente.** Sin validación de formato ni comprobación de permiso, la suscripción es una puerta abierta entre tenants.
- **Medir solo la latencia de publicación.** El servidor puede publicar en 2 ms mientras el usuario no recibe nada. Sin un canario, esa diferencia es invisible.
- **Olvidar el** deregistration_delay **del balanceador.** Un drenaje perfecto en la aplicación no sirve de nada si el

balanceador sigue enviando conexiones nuevas al pod que se está muriendo.

- **Reconectar sin límite de intentos ni tope de espera.** Un backoff exponencial sin techo deja al usuario esperando ocho minutos tras un corte de red de diez segundos.
- **Probar solo la carga estable.** El escenario que rompe servicios es la recuperación tras perder una instancia, y no aparece en ninguna prueba de rampa suave.

Checklist antes de producción

- maxPayload configurado y perMessageDeflate desactivado salvo medición en contra.
- Todos los envíos pasan por una función que comprueba bufferedAmount, con política explícita de descarte, colapsado o cierre.
- Heartbeat con intervalo *menor* que el idle timeout del proxy o balanceador, y terminate() para las conexiones que no responden.
- Backplane suscrito por sala, no un canal global. Si los mensajes no pueden perderse, con persistencia.
- Eventos numerados y ventana de replay acotada, con una respuesta explícita de resync_required cuando el hueco es demasiado grande.
- Cliente con backoff exponencial y jitter completo, y reanudación desde el último id.
- SIGTERM que drena escalonando los cierres con código 1001, preStop y terminationGracePeriodSeconds holgado.
- Origin validado en el handshake y autenticación por ticket de un solo uso, nunca el token de sesión en la URL.
- Autoescalado por conexiones activas, no por CPU.

Checklist ampliada

- LimitNOFILE ajustado en el servicio y verificado *dentro* del contenedor, no en el host.
- Una sola serialización por publicación, y buffer reutilizado si el volumen lo justifica.
- Bandeja de salida con vaciado periódico y colapsado por clave para datos de estado.
- Cada suscripción pasa por una comprobación de permiso, con caché de vida corta y borrado en la revocación.
- Nombre de sala validado contra una expresión regular antes de tocar Redis.
- Límite de mensajes por segundo y por conexión, límite de salas por conexión y límite de conexiones por IP.
- Timeout en el socket a medio negociar, para que un upgrade sin terminar no ocupe un descriptor para siempre.
- Las seis métricas exportadas, sin etiquetas de alta cardinalidad, y un canario por región.
- Intervalo de heartbeat menor que el timeout de inactividad más pequeño de todo el camino, con margen para una pérdida.
- deregistration_delay del balanceador y terminationGracePeriodSeconds contados juntos y en el orden correcto.

- Eventos que no se pueden perder en un stream persistente, no en Pub/Sub, con retención calculada sobre tu ritmo real.
- Manejadores idempotentes en todo lo que se reprocesa tras un xAutoClaim.
- Prueba de carga con rampa por etapas y prueba explícita de tormenta de reconexión, ambas en CI o al menos antes de cada cambio grande.

Preguntas frecuentes

¿Cuántas conexiones aguanta un proceso de Node.js?

Depende del tráfico, no de Node. Con ws, mensajes moderados y estado pequeño por conexión, entre 10.000 y 60.000 es el rango cómodo; 100.000 es alcanzable pero sin margen. Con uWebSockets.js el techo sube varias veces. El número que necesitas es el tuyo, medido con la rampa por etapas de la sección de pruebas de carga.

¿Hace falta sticky sessions para escalar WebSockets?

No. Una vez completado el upgrade, la conexión ya está fijada a una instancia por el propio TCP. La afinidad solo importa para el handshake y para flujos donde una petición HTTP previa dejó estado local. Si tu servicio *necesita* que el usuario vuelva siempre a la misma instancia, el problema es el estado local sin replicar.

¿Puedo poner el token en la URL de conexión?

Es lo que hace mucha gente porque el navegador no deja añadir cabeceras a new WebSocket(), pero deja el token en logs de acceso, en historiales de proxy y en herramientas de monitorización. La alternativa razonable es una cookie HttpOnly con SameSite, o un ticket de un solo uso y vida muy corta que se pide por HTTP y se canjea en el handshake.

¿Por qué se desconectan todos a los 60 segundos exactos?

Casi siempre es proxy_read_timeout de nginx o el timeout de inactividad del ALB, ambos con 60 segundos por defecto. La pista es la precisión del número: los problemas de red no ocurren a intervalos exactos.

¿Compresión sí o no?

permessage-deflate viene desactivado en ws por un motivo: cada conexión reserva su propio contexto de zlib, y el coste en memoria y CPU es considerable a escala. Si tus mensajes son grandes y repetitivos puede compensar, pero es una decisión que se toma con una medición representativa de tu carga, nunca por defecto.

¿Redis Pub/Sub o Redis Streams?

Pub/Sub para lo efímero, donde perder un mensaje durante un reinicio no cambia nada: presencia, cursores, cotizaciones. Streams para todo lo que el usuario pueda echar en falta, y como log de reanudación al reconectar. Muchos servicios usan los dos, y está bien.

¿Cómo pruebo esto en local si el problema solo pasa a escala?

No lo pruebas a escala en local, pero sí puedes reproducir las condiciones. Un cliente que no lee del socket

reproduce el backpressure con una sola conexión. Un tc netem con 300 ms de retardo y 2% de pérdida reproduce el móvil en el metro. Y matar un contenedor a mano reproduce la tormenta de reconexión en pequeño.

¿Merece la pena migrar a WebTransport ahora?

Si tu backend es Go y tu caso tiene datagramas naturales —posiciones, telemetría, audio—, sí, ya es una opción con soporte de navegador completo desde marzo de 2026. Si tu backend es Node.js, todavía no: no hay implementación de primera clase, y estarás manteniendo dos rutas de transporte durante meses a cambio de una mejora que probablemente no sea tu cuello de botella.

¿Qué hago con los clientes que van lentos de forma crónica?

Decide una política y aplícala igual para todos: descartar, colapsar o desconectar. Lo que no puedes hacer es no decidir, porque entonces la política es "acumular en memoria hasta que el proceso muera", y castiga a todos los usuarios de esa instancia por culpa de uno.

Glosario

- **Backpressure.** La situación en la que produces datos más rápido de lo que el consumidor puede recibirlos. En WebSockets se manifiesta como bufferedAmount creciendo sin bajar.
- **bufferedAmount.** Bytes ya aceptados por send() pero todavía no entregados a la red. Es la señal más honesta del estado de un cliente.
- **Frames de control ping y pong.** Mensajes del propio protocolo WebSocket, gestionados por la librería, que sirven para comprobar que el otro extremo sigue vivo.
- **Conexión fantasma.** Un socket que el servidor considera abierto y que en el cliente ya no existe. Consume memoria y descriptores hasta que un heartbeat lo detecta.
- **Fan-out.** La entrega de un mensaje publicado una vez a muchos destinatarios. El coste real está en el último tramo, no en el bus.
- **Pub/Sub fragmentado.** Modo de Redis 7.0 y posteriores (SSUBSCRIBE, SPUBLISH) que limita la entrega al shard responsable del canal en lugar de difundir a todo el cluster.
- **PEL (pending entries list).** En un grupo de consumidores de Redis Streams, la lista de mensajes entregados a un consumidor y todavía sin confirmar con XACK.
- **Colapsado (conflation).** Sustituir varios valores pendientes de la misma clave por el último. Válido para estado, nunca para hechos.
- **Jitter.** Aleatoriedad añadida al tiempo de espera entre reintentos para que miles de clientes no reconecten en el mismo instante.
- **Tormenta de reconexión.** El pico de conexiones nuevas que sigue a la caída de una instancia. Sin jitter, es capaz de tumbar a las instancias que quedaban sanas.
- **Lag del event loop.** El retraso entre cuando una tarea debería ejecutarse y cuando lo hace. En Node.js es el indicador temprano de casi cualquier problema de saturación.
- **Canario.** Un cliente sintético que ejerce el camino completo del usuario y exporta el resultado como métrica. Detecta lo que la instrumentación desde dentro no puede ver.

Ninguna de estas piezas es complicada por separado. Lo que las hace difíciles es que todas fallan en silencio y ninguna se nota en desarrollo: se notan a las tres de la mañana, con tráfico real y un despliegue reciente. Merece la pena dejarlas puestas antes.

PuigFlows

WebSockets at Scale: Backpressure, Heartbeats, Redis Fan-out and Reconnection Without Losing Messages

A WebSocket connection is state, and state does not scale on its own.

Josue Garcia · 2026-09-07 · Expanded edition

ENGLISH

A connection is not a request

An HTTP endpoint is comfortable because it forgets: it receives, responds, and releases everything it was holding. A WebSocket does the opposite. Every client leaves an open socket, a file descriptor, an outbound buffer and a slice of state inside your process for hours. Multiply that by fifty thousand and the problems that show up look nothing like the ones you get from a REST API.

The awkward part is that almost none of them fail with a clear error. The process doesn't throw an exception: it runs out of memory. Connections don't close: they hang. Messages aren't lost with a warning in the logs: they simply never arrive. This guide walks through the five places where that happens —backpressure, heartbeats, fan-out across instances, reconnection and deploys— with code you can copy.

The examples use Node.js with the ws library and Redis, but the patterns translate directly to Go, Elixir or Python.

1. Backpressure: the silent failure that takes down the process

When you call ws.send(), the message does not go out on the wire. It gets copied into an in-memory buffer and the operating system delivers it at whatever rate the client can read. If you produce a hundred messages per second and the client —a phone on bad signal, a background tab with throttled timers— consumes five, the difference stays on your heap. Nobody warns you. The process grows until the kernel kills it, usually at 3am.

The signal you need is ws.bufferedAmount: the bytes you have already accepted that haven't gone out yet. If that number climbs and never comes back down, that client is slower than you are.

```
import { WebSocketServer } from 'ws';

const MAX_BUFFER = 1 << 20; // 1 MiB of queue per client

const wss = new WebSocketServer({
  port: 8080,
  // Off on purpose: per-message compression keeps a zlib context per
  // connection and you pay for it in memory. See "common mistakes".
  perMessageDeflate: false,
  maxPayload: 64 * 1024, // reject oversized frames from the client
});

function safeSend(ws, payload) {
  if (ws.readyState !== ws.OPEN) return false;

  // bufferedAmount = accepted bytes the socket hasn't delivered yet.
  if (ws.bufferedAmount > MAX_BUFFER) {
    ws.slowStrikes = (ws.slowStrikes ?? 0) + 1;

    if (ws.slowStrikes > 3) {
      // 1013 = Try Again Later. Closing beats piling up memory until
      // the kernel kills the process.
      ws.close(1013, 'client too slow');
    }
```

```
    return false; // message dropped
  }

  ws.slowStrikes = 0;
  ws.send(payload);
  return true;
}
```

There are three valid answers to a slow client, and picking one is a product decision, not an infrastructure decision:

- **Drop.** Correct for state data: a price quote, a cursor position, a presence counter. The client needs the current value, not the history.
- **Coalesce.** Instead of dropping, keep the latest value and flush at a fixed rate. A burst of a thousand updates becomes one send.
- **Close.** If every message matters —a chat, a log stream— you can't drop. Close with 1013 and let the client reconnect and ask for what it missed; section 4 shows how.

```
// Coalescing: one snapshot per connection, flushed at a fixed rate.
const pending = new Map(); // ws -> latest snapshot

export function queueSnapshot(ws, snapshot) {
  pending.set(ws, snapshot);
}

setInterval(() => {
  for (const [ws, snapshot] of pending) {
    pending.delete(ws);
    safeSend(ws, JSON.stringify(snapshot));
  }
}, 50); // 20 sends per second, tops, no matter what happens upstream
```

The only thing that is never valid is doing nothing. With no ceiling, a single slow client can bring down the whole instance and take the other twenty thousand with it.

2. Heartbeats: the half of the connection that no longer exists

TCP does not tell you when the other end disappears. If a laptop suspends, if a phone jumps from Wi-Fi to cellular, if a NAT box in the middle recycles its table, your server never receives a FIN. You are left with a *half-open* connection: as far as your process is concerned it's alive —it holds memory and a slot against your connection limit— but there is nobody on the other side. In a service with high churn, that slow drip is exactly why memory climbs like a staircase and never comes back down.

The WebSocket protocol includes *ping* and *pong* control frames for precisely this. Your application never sees them and you don't have to touch the client: the browser answers pings on its own.

```
const HEARTBEAT_MS = 30_000;

wss.on('connection', (ws) => {
```

```
  ws.isAlive = true;
  ws.on('pong', () => { ws.isAlive = true; });
});

const heartbeat = setInterval(() => {
  for (const ws of wss.clients) {
    if (ws.isAlive === false) {
      // terminate(), not close(): close() starts a closing handshake
      // and waits for a reply that is never coming.
      ws.terminate();
      continue;
    }
    ws.isAlive = false;
    ws.ping();
  }
}, HEARTBEAT_MS);

wss.on('close', () => clearInterval(heartbeat));
```

One detail gets missed every single time: **your heartbeat interval must be shorter than the idle timeout of every proxy in the path.** An AWS ALB cuts connections after 60 seconds without traffic by default, and nginx does the same. If your ping fires every 90 seconds you'll see "random" disconnects once a minute on perfectly healthy connections, and you'll burn an afternoon looking for the bug in your own code.

```
location /ws {
  proxy_pass http://ws_upstream;
  proxy_http_version 1.1;
  proxy_set_header Upgrade    $http_upgrade;
  proxy_set_header Connection "upgrade";

  # By default nginx closes after 60s with no data from the upstream.
  # With WebSockets that kills healthy but quiet connections.
  proxy_read_timeout 300s;
  proxy_send_timeout 300s;
}
```

3. Fan-out across instances: a local index and a shared bus

With backpressure under control and dead connections cleared out, the next problem shows up the moment you stop running a single instance.3. Horizontal scaling: the backplane

On a single instance, broadcasting to a room means iterating a Set. With four instances behind a load balancer, that room's members are spread out and each process only knows its own. You need a channel that carries the message to the instances that actually have recipients: that's a *backplane*.

The pattern doesn't change: each instance keeps a local index of room to its own sockets, subscribes on the backplane only to the rooms it currently holds, and publishes there instead of broadcasting directly.

```
import { createClient } from 'redis';

// Two connections: in subscriber mode Redis accepts no other commands.
```

```
const sub = createClient({ url: process.env.REDIS_URL });
const pub = sub.duplicate();
await Promise.all([sub.connect(), pub.connect()]);

// Local index: room -> sockets connected TO THIS instance.
const rooms = new Map();

function join(room, ws) {
  if (!rooms.has(room)) {
    rooms.set(room, new Set());
    // One channel per room. With a single global channel, a message for
    // three people would travel to all forty instances to be discarded.
    sub.subscribe('room:' + room, (raw) => fanout(room, raw));
  }
  rooms.get(room).add(ws);
}

function leave(room, ws) {
  const set = rooms.get(room);
  if (!set) return;
  set.delete(ws);
  if (set.size === 0) {
    rooms.delete(room);
    sub.unsubscribe('room:' + room);
  }
}

function fanout(room, raw) {
  for (const ws of rooms.get(room) ?? []) safeSend(ws, raw);
}

// Any instance, or a worker with no open connections, publishes here.
export function broadcast(room, event) {
  return pub.publish('room:' + room, JSON.stringify(event));
}
```

The limit you need to know before building on top of this: Redis Pub/Sub is fire and forget. It persists nothing and retries nothing. If one instance loses its Redis connection for two seconds, those messages never existed as far as its clients are concerned, and nobody finds out. For presence or cursors that's fine. For a chat it isn't. There you need something durable: Redis Streams, NATS JetStream or Kafka. The next section uses Streams for exactly that reason.

On *sticky sessions*: they're useful, but they are not a scaling strategy. If your instance holds per-connection state it cannot rebuild, you need them. If that state lives in Redis, you don't, and you're better off without: with affinity, a restarting pod leaves all its clients holding a cookie that no longer resolves anywhere.

4. Reconnection: backoff, jitter and resume

A WebSocket client will disconnect. That's not an error case, it's normal operation: tunnels, deploys, network changes, the subway. The useful question isn't how to prevent it, but what happens to the messages from the three seconds the client was gone.

The answer is to number your events. If every event in a room carries a monotonic id and you keep a recent

window of them, "I lost the connection" turns into "give me everything after X". A capped Redis Stream does both at once.

```
export async function publishEvent(room, event) {
  const id = await pub.xAdd(
    'stream:' + room,
    '*',
    { data: JSON.stringify(event) },
    // MAXLEN ~ 1000: approximate trimming, far cheaper than exact.
    { TRIM: { strategy: 'MAXLEN', strategyModifier: '~', threshold: 1000 } }
  );
  await pub.publish('room:' + room, JSON.stringify({ id, ...event }));
  return id;
}

// Stream ids are "milliseconds-sequence": compare them part by part,
// never as strings.
function olderThan(a, b) {
  const [aMs, aSeq] = a.split('-').map(Number);
  const [bMs, bSeq] = b.split('-').map(Number);
  return aMs !== bMs ? aMs < bMs : aSeq < bSeq;
}

export async function replaySince(ws, room, lastId) {
  const key = 'stream:' + room;
  const [oldest] = await pub.xRange(key, '-', '+', { COUNT: 1 });

  // If the client asks for something older than what we kept, we cannot
  // rebuild its history. Saying so beats handing it an inconsistent
  // state it has no way of detecting.
  if (!oldest || olderThan(lastId, oldest.id)) {
    return safeSend(ws, JSON.stringify({ type: 'resync_required' }));
  }

  for (const entry of await pub.xRange(key, '(' + lastId, '+')) {
    safeSend(ws, JSON.stringify({ id: entry.id, ...JSON.parse(entry.message.data) }));
  }
}
```

The client needs two things: to remember the last id it processed, and to reconnect with **exponential backoff and jitter**. Jitter is not decoration. If a thousand clients drop at once because you just deployed and they all wait exactly 500 ms, they all come back at once and knock over the instance that just started, which drops them again, and so on.

```
class ResilientSocket {
  constructor(url, onEvent) {
    this.url = url;
    this.onEvent = onEvent;
    this.attempt = 0;
    this.lastId = null; // last event processed
    this.connect();
  }

  connect() {
    const qs = this.lastId ? '?since=' + encodeURIComponent(this.lastId) : '';
```

```
    this.ws = new WebSocket(this.url + qs);

    this.ws.onopen = () => { this.attempt = 0; };

    this.ws.onmessage = (ev) => {
      const msg = JSON.parse(ev.data);
      if (msg.type === 'resync_required') {
        this.lastId = null;       // the server no longer covers our gap
        return this.onEvent(msg); // reload the full state
      }
      if (msg.id) this.lastId = msg.id;
      this.onEvent(msg);
    };

    this.ws.onclose = () => {
      // Full jitter: wait a random value WITHIN the ceiling, not the
      // ceiling itself. Spreads the stampede instead of syncing it.
      const ceiling = Math.min(30_000, 500 * 2 ** this.attempt++);
      setTimeout(() => this.connect(), Math.random() * ceiling);
    };
  }

  send(data) {
    // The client has backpressure too: if the outbound buffer grows,
    // the user's network can't keep up.
    if (this.ws.readyState !== WebSocket.OPEN) return false;
    if (this.ws.bufferedAmount > 512 * 1024) return false;
    this.ws.send(JSON.stringify(data));
    return true;
  }
}
```

With numbered events and staggered reconnects, a disconnect stops being data loss and becomes a gap that fills itself. What's left is the moment you cause every disconnect at once.5. Deploys: drain instead of cut

5. Deploys: shutting down without dropping everyone at once

This is where most real-time services break, because the problem only shows up in production and under load. When Kubernetes terminates a pod it sends SIGTERM. If your process just exits, every client sees an abnormal close (code 1006) and, because they all see it in the same instant, they all reconnect in the same instant against whatever instances are left.

```
let draining = false;

process.on('SIGTERM', () => {
  draining = true;   // /ready starts returning 503
  server.close();    // stop accepting new handshakes

  const clients = [...wss.clients];
  const windowMs = 10_000; // spread reconnects over 10 seconds

  clients.forEach((ws, i) => {
    setTimeout(() => {
```

```
      // 1001 "going away" tells the client this was expected.
      // Without this explicit close it would see a 1006 instead.
      ws.close(1001, 'server going away');
    }, (i / clients.length) * windowMs);
  });

  setTimeout(() => {
    for (const ws of wss.clients) ws.terminate();
    process.exit(0);
  }, windowMs + 5_000);
});

app.get('/ready', (_req, res) => res.sendStatus(draining ? 503 : 200));
```

```
spec:
  # Must exceed the drain window (10s) plus the margin (5s).
  terminationGracePeriodSeconds: 45
  containers:
    - name: ws-gateway
      lifecycle:
        preStop:
          exec:
            # SIGTERM and the pod's removal from the load balancer happen
            # in parallel. This pause avoids closing connections while
            # new handshakes are still arriving.
            command: ["sh", "-c", "sleep 5"]
      readinessProbe:
        httpGet: { path: /ready, port: 8080 }
        periodSeconds: 2
```

A preStop hook with a short sleep looks like a dirty patch, and it is, but it solves a real race: Endpoint changes propagating to load balancers is not synchronised with SIGTERM. Without that pause you'll be closing connections on one side while the balancer keeps sending you new ones on the other.

Authentication and Origin: two handshake traps

The WebSocket handshake is an HTTP request, but with two quirks that catch people out. First: the browser won't let you set custom headers, so you can't send an Authorization: Bearer. The temptation is to put the token in the query string, where it ends up in the access logs of your proxy, your load balancer and your APM. Issue a single-use ticket with a 30 second lifetime and redeem it during the upgrade.

The second one is worse: **the handshake is not protected by CORS.** Any page can open a connection to your server and the browser will attach the user's cookies. That's *Cross-Site WebSocket Hijacking*, and the only defence is validating Origin yourself.

```
const wss = new WebSocketServer({
  noServer: true,
  verifyClient: ({ origin }) => ALLOWED_ORIGINS.has(origin),
});

server.on('upgrade', async (req, socket, head) => {
```

```
  const url = new URL(req.url, 'http://localhost');
  const userId = await redeemTicket(url.searchParams.get('ticket'));

  if (!userId) {
    socket.write('HTTP/1.1 401 Unauthorized\r\n\r\n');
    return socket.destroy();
  }

  wss.handleUpgrade(req, socket, head, (ws) => {
    ws.userId = userId;
    wss.emit('connection', ws, req);
  });
});
```

Memory budget: how many connections actually fit in one process

Before arguing about architecture, put numbers on the process you already have. The first wall you hit is almost never CPU: it is the file descriptor limit. Linux hands out 1,024 per process by default, and every accepted connection consumes one. With that value your server stops accepting somewhere past a thousand connections, and it does it with an EMFILE that usually shows up in the logs as a bare accept error with no context.

The fix is boring, and you have to do it in all three layers where the process lives, because changing only one of them buys you nothing:

```
# 1. The shell, for local testing.
ulimit -n 1048576

# 2. The service, if you start under systemd. This wins over limits.conf.
# /etc/systemd/system/realtime.service
[Service]
LimitNOFILE=1048576

# 3. The kernel, so the accept queue does not drop SYNs during a spike.
sysctl -w net.core.somaxconn=65535
sysctl -w net.ipv4.tcp_max_syn_backlog=65535
sysctl -w net.ipv4.ip_local_port_range='10240 65535'
```

That third block matters most when *you* are the one opening many connections: a load balancer or proxy talking to your backend exhausts the ephemeral port range far sooner than intuition suggests. In containers, check the limit inside the container and not on the host: `cat /proc/1/limits` from inside the pod is the only answer that counts.

With descriptors out of the way, the next ceiling is memory, and here arithmetic beats generic advice. An idle connection costs the kernel socket, its receive and send buffers, the JavaScript object representing it, and whatever state you hang off it (user, rooms, cursors). On a tuned server that lands around 10 KB per connection: 100,000 connections is roughly 1 GB of pure bookkeeping before you have sent a single useful byte. An *active* connection with pending messages is a different story, because that is where the outgoing buffer from the backpressure section lives, and that one can grow to tens or hundreds of kilobytes each.

One Node.js detail trips up almost everyone who tries to measure this: socket buffers do not live in the V8 heap.

If you watch only heapUsed you will see a flat line while the process walks into an OOM kill. What grows is external memory and RSS.

```
// Cheap sampling that fits in any service.
setInterval(() => {
  const m = process.memoryUsage();
  const n = wss.clients.size || 1;
  console.log(JSON.stringify({
    conns: n,
    rss_mb: (m.rss / 1048576).toFixed(1),
    heap_mb: (m.heapUsed / 1048576).toFixed(1),
    // The usual suspect when the heap does not explain the RSS.
    external_mb: (m.external / 1048576).toFixed(1),
    bytes_per_conn: Math.round(m.rss / n)
  }));
}, 15000);
```

The useful field is the last one: bytes per connection. If it stays flat as connections climb, you scale linearly and you can project. If it grows over time at a constant connection count, you have a leak, and the place to look is almost always a room map you never delete from when a socket closes.

So how many connections fit in one process? With the ws library and moderate message rates, 10,000 to 60,000 is where most services live comfortably; reaching 100,000 is possible but leaves little headroom for spikes. That ceiling is not a property of Node.js in the abstract, it is a property of your workload: twenty messages per second per client looks nothing like one every ten seconds. The number you need is not the one in an article, it is yours, and the load testing section below is how you get it.

A note on --max-old-space-size: raising it does not help here. It only moves the V8 heap limit, which is not where the weight is. If your process dies of OOM while the heap looks calm, the fix is in the outgoing buffer and in connections per process, not in a startup flag.

Serialize once, not once per client

This is the most expensive and most easily made performance mistake in a real-time server, and it never shows up in a stack trace. Broadcasting to a room is almost always written like this:

```
// Wrong: N calls to JSON.stringify for N clients.
for (const ws of room) {
  ws.send(JSON.stringify(event));
}
```

With five clients it does not matter. With five thousand, you have just serialized the same object five thousand times in the same tick, blocking the event loop while you do it. A 2 KB event broadcast to 5,000 connections is 10 MB of strings created and thrown away per publish. The symptom is not an error: it is a p99 latency that creeps up and an event loop lag nobody connects to the broadcast path.

```
// Right: one serialization, N sends.
const payload = JSON.stringify(event);
for (const ws of room) {
```

```
  if (ws.readyState !== ws.OPEN) continue;
  if (ws.bufferedAmount > MAX_BUFFER) { handleSlowClient(ws); continue; }
  ws.send(payload);
}
```

You can go one step further. Converting the payload to a Buffer once also skips the UTF-8 conversion on every send; with ws you just pass the buffer and mark the frame as text:

```
const frame = Buffer.from(JSON.stringify(event), 'utf8');
for (const ws of room) ws.send(frame, { binary: false });
```

The message envelope, and why you version it on day one

The message format is a contract with clients you do not control: browsers with a tab open for three days, mobile apps the user has not updated, third-party integrations. If you publish { price: 42 } and tomorrow you need to add the currency, you have no way of knowing who understands the new shape.

```
// Stable envelope: version and type on the outside, data inside.
{
  v: 1,               // envelope version
  t: 'quote.updated', // event type, past tense and namespaced
  id: '1725...-3',    // sortable identifier for resuming
  ts: 1757246400123,
  d: { symbol: 'ACME', price: 42, currency: 'EUR' }
}
```

Three rules that save you migrations: new fields are always optional, old fields never change meaning, and a client that receives an unknown t ignores it silently instead of breaking. With that in place you can ship a new event without coordinating with anyone.

On binary formats: MessagePack or CBOR cut 15% to 40% off JSON on payloads with repeated keys, and Protobuf considerably more when the schema is fixed. But JSON has one advantage that is hard to beat in production: you can read it in the browser network tab without installing anything. The honest recommendation is to start with JSON, measure real bandwidth per connection, and switch only if that figure is a cost or battery problem, not because binary sounds faster.

Batching and conflation: fewer frames, same information

As frequency rises, the cost stops being message size and becomes frame count. Every send() is a frame header, a socket write and, if you are lucky, a syscall. A client receiving forty updates per second is not going to paint forty times: the browser refreshes at 60 Hz and your UI probably coalesces changes before rendering anyway.

The answer is a per-connection outbox flushed on a fixed cadence, with key-based conflation for state data:

```
const FLUSH_MS = 25;

class Outbox {
  constructor(ws) {
```

```
    this.ws = ws;
    this.latest = new Map(); // key -> latest value (conflates)
    this.queue = [];         // events that must not be lost
    this.timer = null;
  }

  // State data: only the latest matters.
  set(key, event) {
    this.latest.set(key, event);
    this.schedule();
  }

  // Facts: every one counts.
  push(event) {
    this.queue.push(event);
    this.schedule();
  }

  schedule() {
    if (this.timer) return;
    this.timer = setTimeout(() => this.flush(), FLUSH_MS);
  }

  flush() {
    this.timer = null;
    if (this.ws.readyState !== this.ws.OPEN) return;

    const batch = [...this.queue, ...this.latest.values()];
    this.queue.length = 0;
    this.latest.clear();
    if (batch.length === 0) return;

    // If the client is already behind, do not add to the pile.
    if (this.ws.bufferedAmount > MAX_BUFFER) {
      metrics.dropped.inc({ reason: 'backpressure' }, batch.length);
      return;
    }
    this.ws.send(JSON.stringify({ v: 1, t: 'batch', d: batch }));
  }
}
```

Two details make the difference. First, the timer only arms when there is something to send, so an idle connection costs nothing. Second, the split between set and push: a quote or a cursor position can be conflated without losing useful information, but a chat message or a payment confirmation cannot. Mixing both in the same queue is the standard way to lose the events that mattered.

The measured effect in real services is large: going from forty frames per second per socket to one every 25 ms cuts process work more than it cuts bandwidth, because it removes syscalls and garbage collector pressure. It is by far the best effort-to-result change in this guide.

Fan-out at scale: sharded Pub/Sub and the arithmetic behind your bill

The pattern from the previous section — one Redis subscription per instance plus a local room index — works until it does not, and it helps to know where that boundary sits before you reach it.

In classic Redis Pub/Sub, a published message is delivered to every subscriber of the channel. In Redis Cluster that means the message travels to *every* node, because any of them might hold an interested subscriber. With six nodes and ten thousand publishes per second, your internal bus moves sixty thousand messages per second even though each message only concerns one instance. That is traffic doing nothing, paid for in Redis CPU and network.

Sharded Pub/Sub, available since Redis 7.0 through SSUBSCRIBE and SPUBLISH, ties each channel to its hash slot and delivers only within that shard. The code change is small; the effect on your Redis bill is not:

```
import { createClient } from 'redis';

const pub = createClient({ url: REDIS_URL });
const sub = pub.duplicate();
await Promise.all([pub.connect(), sub.connect()]);

// Before: publish/subscribe. On cluster, that talks to every node.
// Now: sPublish/sSubscribe. The message never leaves the channel's shard.
async function joinRoom(room, ws) {
  const channel = 'room:' + room;
  if (!local.has(channel)) {
    local.set(channel, new Set());
    await sub.sSubscribe(channel, (raw) => deliverLocal(channel, raw));
  }
  local.get(channel).add(ws);
}

async function leaveRoom(room, ws) {
  const channel = 'room:' + room;
  const set = local.get(channel);
  if (!set) return;
  set.delete(ws);
  // Without this line you accumulate dead subscriptions until Redis complains.
  if (set.size === 0) {
    local.delete(channel);
    await sub.sUnsubscribe(channel);
  }
}
```

The other design decision with real impact is channel granularity. There are two extremes and a sensible point between them. A single global channel carrying everything forces each instance to receive and discard 90% of the traffic; it is simple and works fine up to a few thousand messages per second. A channel per room means each instance only receives what it needs, but with a hundred thousand active rooms you have a hundred thousand subscriptions spread across the cluster, and subscribe/ unsubscribe operations become the bottleneck when users join and leave quickly.

The middle ground that holds up best is grouping rooms into a fixed number of channels by hash: 'room:' + (hash(room) % 256). Each instance keeps 256 stable subscriptions, joining a room never touches Redis at all, and wasted traffic is divided by 256 compared to the single-channel design. It is the boring solution and it is usually the right one.

```
const SHARDS = 256;
const shardOf = (room) => 'fan:' + (xxhash32(room) % SHARDS);
```

```
// Once, at startup. No dynamic subscriptions on the hot path.
for (let i = 0; i < SHARDS; i++) {
  await sub.sSubscribe('fan:' + i, onShardMessage);
}

function onShardMessage(raw) {
  const { room, payload } = JSON.parse(raw);
  const set = localRooms.get(room);
  if (!set) return; // not for this instance: discarding is cheap
  for (const ws of set) trySend(ws, payload);
}
```

There is one piece of arithmetic worth writing out. If a room has 50,000 members spread across 10 instances and you publish 20 messages per second, Redis moves 200 messages per second (20 × 10) while your processes execute a million send() calls per second between them. The bus is rarely the bottleneck: the last hop is. When someone proposes replacing Redis with Kafka or NATS to “handle the fan-out”, the right question is which of those two numbers is in the red, because swapping the bus does not touch the one that usually is.

Reliable delivery: Pub/Sub is not a queue

This deserves repeating because it is the confusion that makes the most data disappear without a trace. Redis Pub/Sub delivers to whoever is listening *at that instant*. If your instance was restarting, if the subscriber lost its connection for half a second, if the process was busy: the message is not retried and not stored. No error, no metric, nothing. It simply did not arrive.

For anything a user could notice missing — a chat message, a notification, an order status change — you need a durable log. Redis Streams fills that role without adding a new component to your architecture, and it fits naturally with the id-based resume you already saw in the reconnection section.

```
// Publish: the stream is both bus and resume log.
await pub.xAdd('room:' + room, '*',
  { type: event.t, data: JSON.stringify(event.d) },
  // Approximate retention by entry count (far cheaper than exact).
  { TRIM: { strategy: 'MAXLEN', strategyModifier: '~', threshold: 10000 } }
);
```

Retention comes from a simple calculation: how far back you want to be able to resume, times your event rate. If you publish 5 events per second in a room and want to cover a thirty-minute disconnection, you need 9,000 entries; MAXLEN ~ 10000 gives you room. The ~ modifier lets Redis trim whole macro nodes instead of entry by entry, and the cost difference is noticeable on busy streams.

When the work derived from the event also cannot be lost — sending an email, charging a card, updating an index — consumer groups come in. Every delivered message stays in that consumer's pending list until it is acknowledged with XACK. If the process dies between delivery and acknowledgement, the message is still there, invisible to the rest of the group, until someone claims it:

```
const GROUP = 'fanout';
const ME = process.env.POD_NAME;
```

```
// Happy path: read what nobody has seen yet.
async function consume() {
  const res = await redis.xReadGroup(GROUP, ME,
    [{ key: STREAM, id: '>' }], { COUNT: 100, BLOCK: 5000 });
  for (const msg of res?.[0]?.messages ?? []) {
    await handle(msg);
    await redis.xAck(STREAM, GROUP, msg.id);
  }
}

// Recovery path: claim what a dead peer left half-done.
// Claim and process in one step; claiming alone just moves the problem.
async function reapIdlePel() {
  let cursor = '0-0';
  do {
    const { nextCursor, messages } = await redis.xAutoClaim(
      STREAM, GROUP, ME, 60000 /* idle ms */, cursor, { COUNT: 50 });
    for (const msg of messages) {
      await handle(msg);           // handle MUST be idempotent
      await redis.xAck(STREAM, GROUP, msg.id);
    }
    cursor = nextCursor;
  } while (cursor !== '0-0');
}

setInterval(reapIdlePel, 15000);
```

Two details decide whether this works: xAutoClaim only claims entries idle for longer than the threshold, so a slow-but-alive consumer does not get robbed; and claiming and processing must happen together, because if you only claim, the entries move into your pending list and the problem has changed location without being solved. Since the same message can be processed twice — the process can die right after handle and before xAck — the handler has to be idempotent. Here idempotency is not an optional best practice: it is the price of admission.

Application-level acks for what the client sends up

The other direction has the same problem and the same solution. When the client sends something that must not be lost, the browser's send() guarantees nothing: it hands the bytes to a local buffer, and if the connection drops right then the message evaporates without an error.

```
// Client: own identifier, retry until acknowledged.
const pending = new Map();

function sendReliable(type, data) {
  const id = crypto.randomUUID();
  const msg = { v: 1, t: type, id, d: data };
  pending.set(id, msg);
  if (ws.readyState === WebSocket.OPEN) ws.send(JSON.stringify(msg));
  return id;
}

ws.addEventListener('message', (e) => {
```

```
  const m = JSON.parse(e.data);
  if (m.t === 'ack') pending.delete(m.d.id);
});

// On reconnect, resend anything unacknowledged. The server dedupes by id.
ws.addEventListener('open', () => {
  for (const msg of pending.values()) ws.send(JSON.stringify(msg));
});
```

The server stores seen identifiers in a Redis key with a TTL (a few hours is plenty) and replies with the same acknowledgement if the message was already processed. It is the same idempotency-key pattern used in payment APIs, applied to a channel where retries are not the exception but the norm.

Per-room authorization: the client never decides what it subscribes to

The handshake tells you *who* the user is. It does not tell you what they are entitled to. That distinction gets lost easily because subscription code usually starts life in a demo, where the server does whatever the client asks:

```
// The most common authorization bug in real-time services.
ws.on('message', (raw) => {
  const msg = JSON.parse(raw);
  if (msg.t === 'subscribe') rooms.join(msg.room, ws); // anyone, any room
});
```

With that, an authenticated user of tenant A types {"t":"subscribe","room":"org:B:alerts"} in the browser console and starts receiving tenant B traffic. There is no exploit and no vulnerability chain: it is the feature exactly as written. And because the channel never touches your HTTP middleware, none of the protections you do have on the REST API apply here.

The rule is simple: every subscription is an authorization decision, just like every endpoint. And since it runs on the hot path, it deserves a short-lived cache:

```
const ROOM_RE = /^(chat|alerts|board):[a-z0-9-]{1,64}$/;
const authzCache = new Map(); // 'userId|room' -> { ok, exp }
const AUTHZ_TTL_MS = 30000;

async function canSubscribe(user, room) {
  if (!ROOM_RE.test(room)) return false;

  const key = user.id + '|' + room;
  const hit = authzCache.get(key);
  if (hit && hit.exp > Date.now()) return hit.ok;

  const ok = await db.userCanRead(user.id, room); // your real logic
  authzCache.set(key, { ok, exp: Date.now() + AUTHZ_TTL_MS });
  return ok;
}

ws.on('message', async (raw) => {
  const msg = safeParse(raw);
  if (!msg) return ws.close(1003, 'bad payload');
```

```
  if (msg.t === 'subscribe') {
    if (ws.rooms.size >= MAX_ROOMS_PER_CONN) {
      return send(ws, { t: 'error', d: { code: 'too_many_rooms' } });
    }
    if (!(await canSubscribe(ws.user, msg.room))) {
      // Do not distinguish 'does not exist' from 'not allowed': that leaks.
      return send(ws, { t: 'error', d: { code: 'forbidden' } });
    }
    rooms.join(msg.room, ws);
    send(ws, { t: 'subscribed', d: { room: msg.room } });
  }
});
```

The regular expression is not decoration. Without it, a room name is an arbitrary string that will end up concatenated into a Redis key and, in some designs, into a query. Validating the shape before using it closes that door and also stops a client from creating a million empty rooms with random names and filling your room map.

Live revocation: the permission that expired mid-connection

A WebSocket connection can live for hours. If you remove someone from a project, their HTTP session stops working on the next click; their WebSocket keeps receiving events until they happen to disconnect. It is the real-time equivalent of a token you cannot revoke.

The solution that adds the least new code is a control channel carrying revocations, consumed by every instance:

```
// Publisher: when permissions change, tell the whole cluster.
await pub.publish('control', JSON.stringify({
  t: 'revoke', userId, room
}));

// Each instance, in its control subscriber:
sub.subscribe('control', (raw) => {
  const m = JSON.parse(raw);
  if (m.t !== 'revoke') return;

  for (const ws of rooms.membersOf(m.room)) {
    if (ws.user.id !== m.userId) continue;
    rooms.leave(m.room, ws);
    authzCache.delete(m.userId + '|' + m.room);
    send(ws, { t: 'unsubscribed', d: { room: m.room, reason: 'revoked' } });
  }
});
```

Notice that revocation also deletes the authorization cache entry. Without that line the client resubscribes a second later and the cache says yes for the thirty seconds it had left. It is a bug you will not catch in manual testing because the window is short, and in production it means revocation works about half the time.

For the multi-tenant case, the discipline that prevents nearly every incident is that the tenant identifier never comes from the message: it is taken from the connection's authenticated context and prefixed to the room

name on the server. If the client can write the tenant, isolation depends on nobody ever making a validation mistake.

Rate limiting on the channel: abuse does not come through the API

Every service has rate limiting on the HTTP API. Very few have it on the WebSocket channel, and yet that is where an attacker stops paying the cost of a handshake per request: they open one connection and send a hundred thousand messages down the same tunnel. You need limits in three distinct places, because they protect against different things.

At the handshake, so nobody opens ten thousand connections from one IP and drains your descriptors. **Per connection**, so one client cannot saturate the event loop with inbound messages. **Per payload**, so a single frame cannot allocate a hundred megabytes before you get a chance to reject it.

```
import { WebSocketServer } from 'ws';

const wss = new WebSocketServer({
  noServer: true,
  // A frame larger than this closes the connection with 1009.
  maxPayload: 64 * 1024,
  perMessageDeflate: false
});

// Per-connection token bucket: 20 msg/s sustained, bursts of 50.
class TokenBucket {
  constructor(rate, burst) {
    this.rate = rate; this.burst = burst;
    this.tokens = burst; this.last = Date.now();
  }
  take() {
    const now = Date.now();
    this.tokens = Math.min(this.burst, this.tokens + ((now - this.last) / 1000) * this.rate);
    this.last = now;
    if (this.tokens < 1) return false;
    this.tokens -= 1;
    return true;
  }
}

wss.on('connection', (ws) => {
  ws.bucket = new TokenBucket(20, 50);
  ws.strikes = 0;

  ws.on('message', (raw) => {
    if (!ws.bucket.take()) {
      metrics.throttled.inc();
      // Three strikes: closing on the first one punishes bad networks.
      if (++ws.strikes > 3) return ws.close(1008, 'rate limit');
      return send(ws, { t: 'error', d: { code: 'rate_limited' } });
    }
    handle(ws, raw);
  });
});
```

The three-strikes detail is not leniency: a phone coming out of a tunnel can flush its whole queue of pending messages at once and trip the limit with no bad intent. Closing on the first offence turns poor coverage into a reconnection loop.

At the handshake, the per-IP limit applies before you accept the upgrade, and you want to count live connections rather than attempts:

```
const perIp = new Map();
const MAX_PER_IP = 50;

server.on('upgrade', (req, socket, head) => {
  const ip = req.headers['x-forwarded-for']?.split(',')[0].trim() || req.socket.remoteAddress;
  const n = perIp.get(ip) || 0;
  if (n >= MAX_PER_IP) {
    socket.write('HTTP/1.1 429 Too Many Requests\r\n\r\n');
    return socket.destroy();
  }
  // A half-negotiated socket consumes resources too.
  socket.setTimeout(10000, () => socket.destroy());

  authenticate(req).then((user) => {
    if (!user) { socket.write('HTTP/1.1 401 Unauthorized\r\n\r\n'); return socket.destroy(); }
    wss.handleUpgrade(req, socket, head, (ws) => {
      perIp.set(ip, n + 1);
      ws.on('close', () => {
        const c = (perIp.get(ip) || 1) - 1;
        c <= 0 ? perIp.delete(ip) : perIp.set(ip, c);
      });
      ws.user = user;
      wss.emit('connection', ws, req);
    });
  });
});
```

The socket setTimeout covers an attack almost nobody tests for: open TCP connections, begin the upgrade, and never finish it. Without that timer, every half-finished attempt holds a descriptor indefinitely. And clearing the counter on close matters as much as the limit itself: without it, perIp is a memory leak that eventually starts blocking legitimate users.

Observability: six metrics, and why these six

A poorly instrumented real-time service is diagnosed through user complaints, and complaints arrive late and badly described (“sometimes messages do not reach me”). With six series you can explain almost any incident without opening a debugger.

- **Active connections** (gauge, per instance). The shape of the curve says more than the number: a step down is a deploy or a crash; a steady climb with flat user counts is a zombie connection leak.
- **Closes by reason** (counter labelled reason: normal, heartbeat, backpressure, rate_limit, shutdown). This is the metric that turns “they get disconnected” into a specific problem.
- **Outgoing buffer bytes** (histogram, sampled). Gives you the distribution of slow clients. The p50 is always zero; what matters is the p999.

- **Dropped messages** (counter labelled reason). If you drop by policy, you have to know how much.
- **Publish-to-deliver latency** (histogram). The only metric that measures what the user perceives.
- **Event loop lag** (histogram). In Node.js it is the early indicator of nearly everything: per-client serialization, oversized JSON, synchronous work on the broadcast path.

```
import client from 'prom-client';
import { monitorEventLoopDelay } from 'perf_hooks';

const connections = new client.Gauge({
  name: 'ws_connections_active', help: 'Open connections'
});
const closes = new client.Counter({
  name: 'ws_closes_total', help: 'Closes by reason', labelNames: ['reason']
});
const outbuf = new client.Histogram({
  name: 'ws_send_buffer_bytes', help: 'Sampled bufferedAmount',
  buckets: [0, 4096, 65536, 262144, 1048576, 8388608, 33554432]
});
const delivery = new client.Histogram({
  name: 'ws_publish_to_deliver_seconds', help: 'Publish to deliver',
  buckets: [0.005, 0.01, 0.05, 0.1, 0.25, 0.5, 1, 2, 5]
});

// Event loop lag, at negligible cost.
const h = monitorEventLoopDelay({ resolution: 20 });
h.enable();
setInterval(() => {
  eventLoopP99.set(h.percentile(99) / 1e6); // ms
  h.reset();
}, 10000);

// Sample the buffer: walking 100k sockets every second is not free.
setInterval(() => {
  let i = 0;
  for (const ws of wss.clients) {
    if (i++ % 50 !== 0) continue; // one in fifty
    outbuf.observe(ws.bufferedAmount);
  }
}, 10000);
```

The cardinality rule is non-negotiable: **never** label by user, room or connection id. A room label with ten thousand rooms multiplies every series by ten thousand, and the thing that falls over is not your service but Prometheus. If you need that detail, it belongs in a structured log or a trace, not in a metric.

The canary: measuring delivery, not publication

The latency people usually measure is the server's: how long publishing takes. That number can look perfect while users receive nothing, because the problem is in the fan-out, the network, or a stuck client. The cheap way to measure the whole chain is a canary: a synthetic client, deployed like any other service, that connects as a real user and times the round trip.

```
// Canary: publish over HTTP, wait over WebSocket, export the difference.
```

```
setInterval(async () => {
  const nonce = crypto.randomUUID();
  const t0 = performance.now();

  const done = new Promise((resolve, reject) => {
    const timer = setTimeout(() => reject(new Error('timeout')), 5000);
    canaryWs.once('message', (raw) => {
      if (JSON.parse(raw).d?.nonce !== nonce) return;
      clearTimeout(timer); resolve();
    });
  });

  await fetch(API + '/canary', { method: 'POST', body: JSON.stringify({ nonce }) });

  try {
    await done;
    delivery.observe((performance.now() - t0) / 1000);
  } catch {
    canaryFailures.inc();
  }
}, 10000);
```

One canary per region and per instance catches the case no other metric sees: an instance that is healthy by its own standards, with open connections and low CPU, that stopped receiving from the message bus because its Redis subscriber dropped and nothing reconnected it. From the inside everything is fine; from the outside, 25% of your users are staring at frozen data.

Load balancers and proxies: the timeouts that cut healthy connections

A lot of stability problems blamed on application code turn out to be a timer in an intermediate component that nobody looked at. To a proxy, a WebSocket connection is an HTTP connection that has gone a long time without bytes. If its idle timeout is shorter than your heartbeat interval, the proxy closes it and your server does not find out until it tries to write.

The defaults that cause the most incidents, as of September 2026:

- **AWS Application Load Balancer**: 60 seconds idle by default, configurable up to 4,000. On Kubernetes you set it with the annotation alb.ingress.kubernetes.io/load-balancer-attributes: idle_timeout.timeout_seconds=600.
- **nginx**: proxy_read_timeout and proxy_send_timeout are 60 seconds. This is the number one cause of disconnections at exactly 60 seconds on ingress-nginx deployments.
- **Cloudflare**: closes idle WebSocket connections at around 100 seconds. It is not configurable on the usual plans, so your heartbeat simply has to be faster.
- **Envoy / Istio**: stream_idle_timeout of 5 minutes and a route.timeout of 15 seconds; that second one must be explicitly set to zero on WebSocket routes or it will cut the connection after fifteen seconds.

From which follows a rule worth writing down in the repository: **the heartbeat interval must be shorter than the smallest idle timeout anywhere on the path, with room for one lost packet.** If the shortest link is Cloudflare at 100 seconds, a ping every 30 leaves room for one to go missing without consequences. A ping every 90 does not.

```
# ALB ingress: these three values have to be counted together.
metadata:
  annotations:
    alb.ingress.kubernetes.io/load-balancer-attributes: idle_timeout.timeout_seconds=600
    alb.ingress.kubernetes.io/target-group-attributes: |
      stickiness.enabled=true,
      stickiness.type=lb_cookie,
      deregistration_delay.timeout_seconds=120
```

deregistration_delay is the one almost nobody tunes and the one that decides whether a deploy cuts live connections. The default is 300 seconds, which sounds generous, but if your terminationGracePeriodSeconds is 30, the pod dies long before the balancer has finished draining it. Both numbers have to be counted in the same direction: the balancer drains first, then the process dies.

On sticky sessions it is worth being precise, because they cause confusion. Once the upgrade completes, the TCP connection is already pinned to one target: it cannot migrate. Affinity matters for the *handshake*, not for the traffic after it, and it is mostly useful when the client makes a prior HTTP request whose state must be found on the same instance. It is not, and never was, a scaling strategy: if your design needs a user to always land on the same instance to work at all, what you have is unreplicated local state, and that breaks on the first restart.

Load testing: know your ceiling before production does

Every number in this guide is indicative. The only one worth making decisions on is your service under your workload, and getting it costs an afternoon. What separates a useful test from an invented number is the shape of the ramp and what you watch while it runs.

Jumping straight to peak load does not measure your capacity: it measures your behaviour under attack. Node.js also takes a few seconds to warm up the JIT, so the first data points always lie. A staged ramp with a long plateau is what reflects production.

```
// k6: staged ramp, application-level ping/pong, full round-trip measurement.
import ws from 'k6/ws';
import { check } from 'k6';
import { Trend, Rate } from 'k6/metrics';

const rtt = new Trend('app_rtt_ms');
const lost = new Rate('lost_messages');

export const options = {
  stages: [
    { duration: '2m', target: 5000 },   // gentle ramp
    { duration: '10m', target: 5000 },  // plateau: this is where you measure
    { duration: '2m', target: 20000 },  // second step
    { duration: '10m', target: 20000 },
    { duration: '2m', target: 0 }       // ramp down: avoids dangling connections
  ],
  thresholds: {
    'app_rtt_ms': ['p(99)<500'],
    'lost_messages': ['rate<0.001']
  }
};
```

```
export default function () {
  ws.connect(__ENV.WS_URL, { headers: { Authorization: 'Bearer ' + __ENV.TOKEN } }, (socket) => {
    socket.on('open', () => {
      socket.send(JSON.stringify({ v: 1, t: 'subscribe', room: 'load:' + (__VU % 200) }));
      socket.setInterval(() => {
        socket.send(JSON.stringify({ v: 1, t: 'echo', id: String(Date.now()) }));
      }, 5000);
    });

    socket.on('message', (raw) => {
      const m = JSON.parse(raw);
      if (m.t !== 'echo') return;
      rtt.add(Date.now() - Number(m.id));
      lost.add(false);
    });

    socket.setTimeout(() => socket.close(), 300000);
  });
}
```

The final ramp-down is not decoration: without it you end the test leaving tens of thousands of half-closed connections behind, and the report is contaminated.

While the test runs, what you look at is not whether it held, but four things at once: the **p99 round trip**, the bufferedAmount **distribution** (if the p999 starts climbing, your ceiling is closer than it looks), **RSS at constant load** (if it grows, you have a leak and the result does not project), and **event loop lag**. A service holding 50,000 connections with a 300 ms p99 event loop lag is not holding 50,000 connections: it is one spike away from becoming unresponsive.

The test almost nobody runs: the reconnection storm

The scenario that takes down real services is not steady load, it is recovery. With the test at its plateau, kill an instance holding 25% of the connections and watch. All those clients will reconnect, and if your backoff has no jitter they will do it simultaneously, against three instances that were already busy.

What to verify at that moment: that the peak of new connections per second spreads out over time instead of concentrating, that the surviving instances stay within their memory budget while absorbing the redistribution, and that id-based resume returns the missed events instead of leaving silent gaps. It is a twenty-minute test that saves you a 3 a.m. incident.

Choosing the server: ws, uWebSockets.js or Socket.IO

The library choice matters less than it seems at the start and more than it seems once you have a hundred thousand connections. It helps to know what each one buys you.

ws is the sensible default. It implements the protocol and little else, integrates with Node's HTTP server, and its API is the one you will see in most examples and answers. Its practical limit is in the tens of thousands of connections per process with moderate message rates. Being the slowest of the three does not make it a bad choice: most services never notice, and predictability is worth a lot.

uWebSockets.js is a C++ implementation with Node bindings. Published figures range from 3x to 10x more connections and throughput depending on the benchmark, and in demanding scenarios people have measured on the order of 120,000 connections each receiving a 1 KB message per second at 70% process CPU. The price is real: a different API, no drop-in integration with Express or Fastify, a smaller community and fewer examples to copy from when something breaks at three in the morning. It is a migration you justify with a measurement, not with an article.

Socket.IO is not WebSocket: it is its own protocol that *uses* WebSocket as a transport when it can and falls back to long polling when it cannot. In exchange you get rooms, acknowledgements, automatic reconnection and adapters for horizontal scaling without hand-writing the fan-out. If you are building something where those pieces are 80% of the work, it saves weeks. The costs to keep in mind are that the client has to be theirs (no new WebSocket() from another language without a compatible library), that per-message overhead is higher, and that the compatibility layer complicates diagnosis when the problem is on the network.

The criterion that works: start with ws if you control both client and server and the protocol is simple; start with Socket.IO if you need its primitives right now and your scale is moderate; migrate to uWebSockets.js when you have a graph proving the process is the bottleneck and not the design. Switching libraries to fix an architecture problem is the most expensive way of not fixing it.

When not to use WebSockets

Half the operational cost of this guide disappears if your use case does not need a bidirectional channel. It is worth asking the question up front, not afterwards.

If data only flows server to client — notifications, a dashboard that updates, job progress, tokens from a language model — Server-Sent Events are almost always the better choice. They run over plain HTTP, cross proxies without special configuration, the browser reconnects on its own and resumes with the Last-Event-ID header without you writing a line, and over HTTP/2 or HTTP/3 they do not suffer the old six-connections-per-origin limit. You give up the upstream channel, which in these cases you did not need: that is what your API is for.

If the frequency is low and latency is not critical, a request every few seconds is cheaper to operate than any persistent connection. No heartbeats, no reconnection, no state to drain on every deploy. It sounds inelegant and it is the right answer more often than people admit.

WebSockets win when traffic is bidirectional and frequent: live collaboration, shared editing, games, messaging, trading.

WebTransport is the development with the most runway. It runs over HTTP/3 and QUIC, offers multiple independent streams without head-of-line blocking and, above all, unreliable datagrams: for cursor positions or telemetry, where a lost packet does not matter because the next one corrects it, that is exactly the primitive that was missing. It reached Baseline status in mid-2026, when Safari 26.4 shipped it on 24 March. The obstacle today is the server side: Node.js ships neither a WebTransport client nor a server, and most of what runs in production leans on Go with quic-go and webtransport-go. If your backend is Node, WebTransport is still a project, not a choice.

One clarification that saves time: WebSocket over HTTP/2 is specified (RFC 8441) and over HTTP/3 too (RFC 9220), but to date no major browser or server has shipped it in production; Chrome is still at the prototype stage

and Firefox has announced no implementation. If someone proposes “WebSockets over HTTP/3” as the answer to a problem of yours, the short reply is that it does not yet exist where it would matter.

Case study: from 8,000 shaky connections to 120,000 stable ones

What follows is a composite case, with the shape and ordering these problems actually take. The numbers are representative of a live dashboard service running on four Node instances behind an ALB, not a laboratory measurement.

Starting point. 8,000 concurrent connections at peak. Three symptoms: each instance's RSS climbs through the day until the orchestrator kills it for memory, roughly once a day; p99 delivery sits around 4 seconds at peak; and every deploy produces a window of about 90 seconds of connection errors on the client.

Week 1 — Instrument before touching anything. The six metrics and a canary go in. The bufferedAmount histogram reveals what nobody suspected: 0.4% of connections are holding more than 30 MB of pending data. At 8,000 connections that is 32 sockets holding almost 1 GB between them. The source: one corporate customer on a slow VPN with a dashboard open on a wall-mounted TV.

Week 2 — Serialize once and apply a drop policy. JSON.stringify comes out of the loop and the buffer limit with key conflation goes in. RSS goes from climbing 200 MB an hour to staying flat; peak memory drops from 6 GB to 900 MB per instance. Latency improves as a side effect, because the event loop stops spending hundreds of milliseconds serializing on every broadcast.

Week 2 — Heartbeats aligned with the balancer. Auditing the path turns up an ALB still on the default 60-second idle timeout while the application heartbeat was at 45 — but the ping was only sent when there had been no traffic in that window, which in practice left gaps over a minute. The ping is fixed at an unconditional 30 seconds and the ALB timeout at 600. Phantom connections — open on the server, dead on the client — fall from 12% of the total to 0.3%.

Week 3 — Batch the frames. The outbox with a 25 ms flush goes in. Sockets go from receiving around 40 frames per second to 6 or 7, carrying the same information. Average process CPU drops from 78% to 31%. This is the change that frees up the most capacity of all.

Week 4 — Deploys without an error window. A preStop with 15 seconds of draining, terminationGracePeriodSeconds at 45, deregistration_delay at 120, an application close code telling the client to reconnect, and jitter in the backoff. The 90-second error window becomes staggered reconnections the user does not notice.

Week 5 — Sharded fan-out. Redis was at 65% CPU with a channel per room and 30,000 active rooms, almost all of it subscribe and unsubscribe operations as people opened and closed dashboards. It is replaced by 256 fixed channels with sSubscribe and local routing. Redis CPU drops to 11% and the noise in the graphs disappears.

Result. The same four instances hold 120,000 connections in a load test with a 180 ms p99 delivery, and the service has gone months without a memory-driven restart. None of the changes was hard on its own. The hard part was the order: without the first week's metrics, the team had spent months arguing about whether Redis was the problem, and it was not.

Common mistakes

- **Turning on** permessage-deflate **"because it saves bandwidth".** It ships disabled in ws for a reason: it keeps a zlib context per connection and, under high concurrency on Linux, causes severe memory fragmentation. Enable it only if you've measured your real workload and it pays off.
- **Autoscaling on CPU.** A WebSocket gateway saturates on memory, file descriptors and bandwidth long before CPU. Scale on active connection count; if you scale on CPU your HPA won't notice anything until pods start dying from OOM.
- **Leaving** maxPayload **unset.** With no limit, one client can make you allocate hundreds of megabytes with a single frame.
- **Treating Redis Pub/Sub as a queue.** It isn't one: no persistence, no acks, no retries. If the message matters, use Streams, JetStream or Kafka.
- **Labelling metrics per user or per connection.** That blows up cardinality in Prometheus. Measure aggregates: active connections, a bufferedAmount histogram, ping RTT, a counter by close code, and the resync_required rate —that last one is your best early warning that your replay window has become too short.
- **Only testing locally.** localhost has no latency, no proxies with idle timeouts, no mobile networks changing IP. Almost everything in this guide is invisible until there's a load balancer in the path.

And eight more that only show up at scale

- **Serializing inside the broadcast loop.** One JSON.stringify per client turns a single publish into hidden quadratic work. It is the most expensive performance mistake on this list.
- **Not deleting from the room map on close.** The socket dies, the reference stays. The leak is slow and steady, and shows up as bytes per connection growing with no explanation.
- **Labelling metrics by room or user.** Works locally with three rooms. With ten thousand, the thing that falls over is Prometheus.
- **Trusting the room name the client sends.** With no format validation and no permission check, subscription is an open door between tenants.
- **Measuring only publish latency.** The server can publish in 2 ms while the user receives nothing. Without a canary, that gap is invisible.
- **Forgetting the balancer's** deregistration_delay**.** Perfect draining in the application means nothing if the balancer keeps sending new connections to the pod that is shutting down.
- **Reconnecting with no attempt cap and no maximum delay.** Unbounded exponential backoff leaves the user waiting eight minutes after a ten-second network blip.
- **Testing steady load only.** The scenario that breaks services is recovery after losing an instance, and it never appears in a gentle ramp test.

Pre-production checklist

- maxPayload configured and perMessageDeflate off unless measurement says otherwise.
- Every send goes through a function that checks bufferedAmount, with an explicit drop, coalesce or close policy.

- Heartbeat interval *shorter* than the proxy or balancer idle timeout, and terminate() for connections that stop answering.
- Backplane subscribed per room, not a global channel. If messages can't be lost, use a durable one.
- Numbered events with a bounded replay window, and an explicit resync_required response when the gap is too wide.
- Client with exponential backoff and full jitter, resuming from its last id.
- SIGTERM that drains by staggering 1001 closes, plus preStop and a generous terminationGracePeriodSeconds.
- Origin validated at the handshake and single-use ticket auth, never a session token in the URL.
- Autoscaling on active connections, not CPU.

Extended checklist

- LimitNOFILE set on the service and verified *inside* the container, not on the host.
- One serialization per publish, and a reused buffer if the volume justifies it.
- An outbox with periodic flushing and key-based conflation for state data.
- Every subscription passes a permission check, with a short-lived cache invalidated on revocation.
- Room names validated against a regular expression before they touch Redis.
- Messages-per-second limit per connection, rooms-per-connection limit, and connections-per-IP limit.
- A timeout on the half-negotiated socket, so an unfinished upgrade cannot hold a descriptor forever.
- The six metrics exported, with no high-cardinality labels, and a canary per region.
- Heartbeat interval shorter than the smallest idle timeout on the whole path, with room for one loss.
- Balancer deregistration_delay and terminationGracePeriodSeconds counted together and in the right order.
- Events that cannot be lost on a durable stream, not on Pub/Sub, with retention calculated from your real rate.
- Idempotent handlers for everything reprocessed after an xAutoClaim.
- A staged-ramp load test and an explicit reconnection-storm test, both in CI or at least before every large change.

Frequently asked questions

How many connections can one Node.js process handle?

It depends on traffic, not on Node. With ws, moderate message rates and small per-connection state, 10,000 to 60,000 is the comfortable range; 100,000 is reachable but leaves no headroom. With uWebSockets.js the ceiling rises several times over. The number you need is yours, measured with the staged ramp from the load testing section.

Do I need sticky sessions to scale WebSockets?

No. Once the upgrade completes, the connection is already pinned to one instance by TCP itself. Affinity only

matters for the handshake and for flows where a prior HTTP request left local state. If your service *needs* the user to always land on the same instance, the real problem is unreplicated local state.

Can I put the token in the connection URL?

Plenty of people do, because the browser will not let you add headers to new WebSocket(), but it leaves the token in access logs, proxy histories and monitoring tools. The reasonable alternative is an HttpOnly cookie with SameSite, or a single-use, very short-lived ticket requested over HTTP and redeemed at the handshake.

Why does everyone disconnect at exactly 60 seconds?

Almost always nginx's proxy_read_timeout or the ALB idle timeout, both defaulting to 60 seconds. The clue is the precision of the number: network problems do not happen at exact intervals.

Compression, yes or no?

permessage-deflate is off by default in ws for a reason: every connection allocates its own zlib context, and the memory and CPU cost is considerable at scale. If your messages are large and repetitive it can pay off, but that is a decision you make from a measurement representative of your workload, never by default.

Redis Pub/Sub or Redis Streams?

Pub/ Sub for the ephemeral, where losing a message during a restart changes nothing: presence, cursors, quotes. Streams for anything the user could notice missing, and as the resume log on reconnect. Plenty of services use both, and that is fine.

How do I test this locally if the problem only appears at scale?

You do not test scale locally, but you can reproduce the conditions. A client that never reads from the socket reproduces backpressure with a single connection. tc netem with 300 ms of delay and 2% loss reproduces the phone on the subway. And killing a container by hand reproduces the reconnection storm in miniature.

Is it worth migrating to WebTransport now?

If your backend is Go and your case has natural datagrams — positions, telemetry, audio — then yes, it has had full browser support since March 2026. If your backend is Node.js, not yet: there is no first- class implementation, and you would be maintaining two transport paths for months in exchange for an improvement that is probably not your bottleneck.

What do I do about chronically slow clients?

Pick a policy and apply it uniformly: drop, conflate or disconnect. What you cannot do is not decide, because then the policy is “accumulate in memory until the process dies”, and that punishes every user on that instance because of one.

Glossary

- **Backpressure.** The situation where you produce data faster than the consumer can receive it. On WebSockets it shows up as bufferedAmount growing and never coming back down.

- **bufferedAmount.** Bytes already accepted by send() but not yet handed to the network. It is the most honest signal of a client's state.
- **Ping and pong control frames.** Messages from the WebSocket protocol itself, handled by the library, used to check that the other end is still alive.
- **Phantom connection.** A socket the server considers open that no longer exists on the client. It consumes memory and descriptors until a heartbeat catches it.
- **Fan-out.** Delivering a message published once to many recipients. The real cost is in the last hop, not in the bus.
- **Sharded Pub/Sub.** A Redis 7.0+ mode (SSUBSCRIBE, SPUBLISH) that limits delivery to the shard owning the channel instead of broadcasting across the whole cluster.
- **PEL (pending entries list).** In a Redis Streams consumer group, the list of messages delivered to a consumer and not yet acknowledged with XACK.
- **Conflation.** Replacing several pending values for the same key with the latest one. Valid for state, never for facts.
- **Jitter.** Randomness added to the wait between retries so that thousands of clients do not reconnect at the same instant.
- **Reconnection storm.** The spike of new connections following an instance failure. Without jitter it is capable of taking down the instances that were still healthy.
- **Event loop lag.** The delay between when a task should run and when it does. In Node.js it is the early indicator of almost any saturation problem.
- **Canary.** A synthetic client that exercises the full user path and exports the result as a metric. It catches what instrumentation from the inside cannot see.

None of these pieces is hard on its own. What makes them hard is that they all fail quietly and none of them show up in development: they show up at three in the morning, with real traffic and a recent deploy. Worth putting in place beforehand.