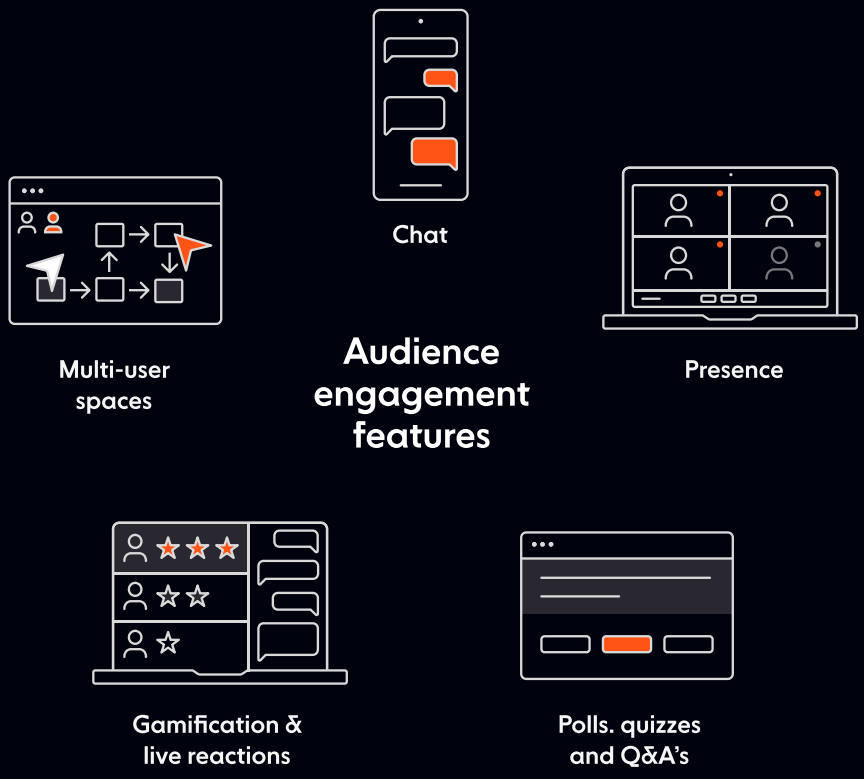
ably

WHITEPAPER

Five WebSocket challenges to building audience engagement features at scale

In recent years, interest in audience engagement has skyrocketed. Think of interactions like virtual polls, quizzes and Q&A sessions, interactive presentations, breakout rooms, or chat, and how often we use and benefit from them in our day-to-day lives.

Shared, live digital experiences have become commonplace. Every organization building the next-generation applications for social live streaming, online education, or employee communication and collaboration considers audience engagement features as must-have requirements.



This whitepaper is a deep dive into some aspects to consider when designing for audience engagement using the WebSocket protocol at a scale that supports **thousands or even millions of concurrent connections.**

Contents

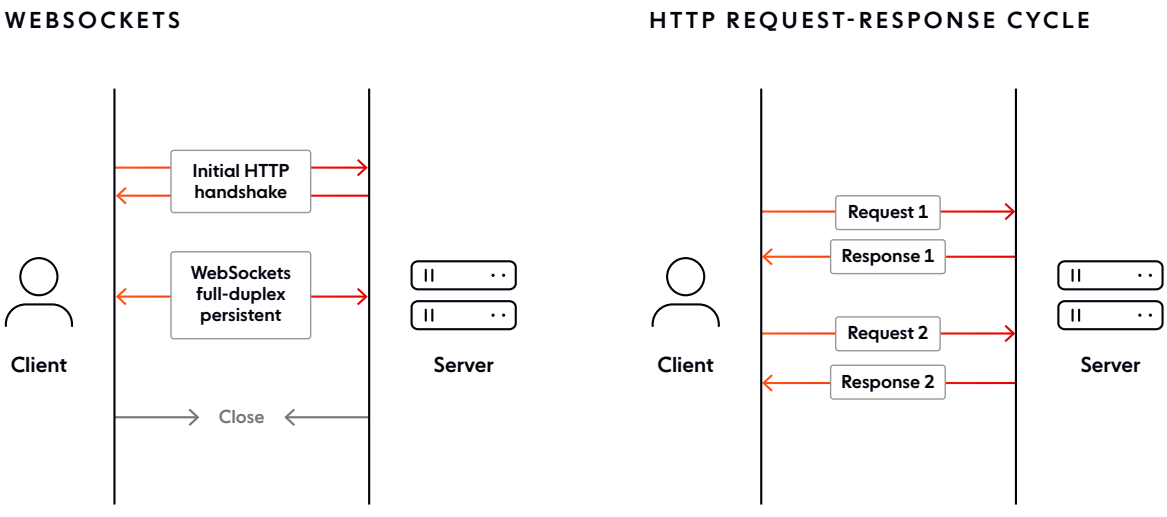
Five WebSocket challenges to building audience engagement features at scale

An introduction to the WebSocket protocol

When talking about audience engagement, we should consider that participants must be able to interact instantly. You need to consider using an async, event-driven architecture powered by an adequate transport protocol, such as WebSocket.

Before WebSocket connections came along, the realtime web existed, but it was difficult to achieve, typically slower, and hacked existing web technologies that were not designed for realtime web applications. The WebSocket protocol paved the way to a truly realtime experience.

Standardized in 2011 by RFC 6455, WebSocket is a thin transport layer built on top of a device's TCP/IP stack. It enables two-way, full-duplex communication over a persistent connection. Compared to REST, WebSocket connections allow for higher efficiency; they scale better because they do not require the HTTP request/response overhead for each message sent and received. Furthermore, the WebSocket protocol is push-based, enabling you to push data to connected clients as soon as events occur. In contrast, with HTTP, you have to poll the server for new information. In the context of a live event, you need to stream data in as close as possible to realtime.



WebSocket integration

You can take several paths to integrate WebSocket capabilities into your tech stack.

The first option is to build your own WebSocket-based messaging solution from scratch and tailor it to your needs using your preferred technologies. For example, DAZN, a well-known provider of live sports streaming services, used the WebSocket protocol to build a custom solution for broadcasting messages to millions of users.

Another option is to use open-source technologies like Socket.IO combined with Redis to run multiple Socket.IO instances in different processes or servers, pass events between nodes, and broadcast messages via the Redis Pub/Sub mechanism. However, Socket.IO has its limitations because it's a simple solution: essentially just a wrapper around the WebSocket API in browsers that offers limited additional functionality. Also, Socket.IO doesn't provide strong messaging guarantees (ordering, guaranteed delivery, and exactly-once semantics). If data integrity is essential to your use case, you will have to build separate components or mechanisms to ensure it.

One of the biggest challenges in audience engagement is to scale swiftly to cope with unpredictable numbers of users, while at the same time providing dependable and uninterrupted experiences. Scaling your system to handle millions of concurrent WebSocket connections dependably is a complex and time-consuming undertaking that requires significant financial, engineering, and DevOps resources. Most of the time, you are better off using a fully-managed WebSocket solution to handle engineering complexities at scale.

Ably's WebSocket solution

The Ably platform provides edge messaging infrastructure to power shared live experiences for millions of global users. We have helped customers reduce time-to-market by an average of 3 months, and build online experiences that engage and retain audience engagement.

We've built our own protocol on top of the WebSocket protocol. It allows you to communicate via WebSocket connections using pub/sub messaging.

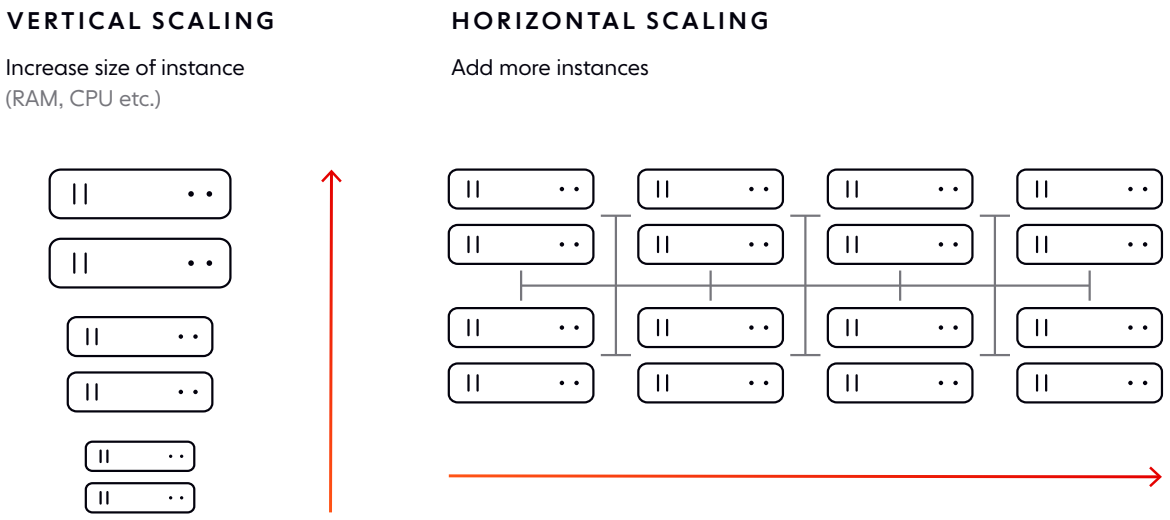
The WebSocket challenges for audience engagement at scale

Let's now examine what a WebSocket-based system must address to support scalable and dependable audience engagement features.

1. Server layer availability

You need to make sure your server layer can sustain an unknown but potentially very high number of concurrent WebSocket connections.

- **Vertical scaling.** Also known as scaling up, it implies adding more power (e.g., CPU, RAM) to an existing machine.
- **Horizontal scaling.** Also known as scaling out, it involves adding more machines to the network, which share the processing workload.



At first glance, vertical scaling seems attractive, as it's easier to implement and maintain than horizontal scaling. But, what happens if the number of concurrent WebSocket connections proves to be too much to handle for just one machine? Or, what happens if you need to upgrade your server? The availability of your system would be severely affected.

In contrast, horizontal scaling is a more dependable model in the long run. Even if a server needs to be upgraded or crashes, you are in a much better position to protect your overall availability since the failing machine's workload can be distributed to the other nodes in the network.

Of course, horizontal scaling comes with its own complications - a more complex architecture, load balancing and routing, and increased infrastructure and maintenance costs, to name a few. One of the biggest challenges is automatically syncing message data and connection state across multiple WebSocket servers in milliseconds. Given that these servers need to be connected in a stateful manner with the clients, state synchronization becomes a tough nut to crack. Furthermore, you'll need to ensure synchronization is always available and globally reliable.

Elasticity

In addition to horizontal scaling, you should also consider the elasticity of your server layer. To successfully handle WebSocket connections at scale, you need to be able to automatically scale to quickly add more servers into the mix so that your system has sufficient capacity to deal with potential usage spikes at all times.

HOW WE SOLVE IT

Dynamic elasticity and high availability guaranteed

Ably is meticulously designed to be elastic and highly available:

- 50% global capacity margin for instant surges.
- Scale to an unlimited number of channels and concurrent connections.
- Reacts to changes to double connection and channel capacity within minutes.
- 99.999% uptime SLA.

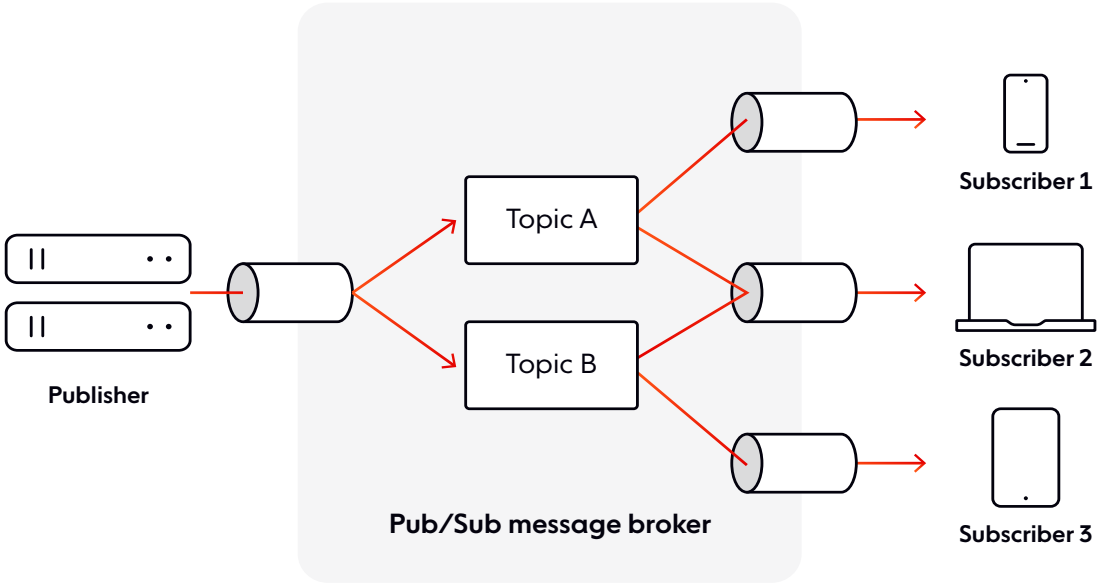
Learn more about Ably's availability

2. An architecture pattern designed for scale

The audience size for interactive, realtime digital experiences is frequently unpredictable. Therefore, your system should be designed to handle an unknown but potentially very high number of concurrent users.

To deal with this unpredictability, you should architect your system based on a pattern designed for scale. One of the most popular and dependable choices is the pub/sub pattern. Pub/sub provides a framework for publishers (typically your server) and subscribers (end-user devices in our context) to exchange messages.

Publishers and subscribers are decoupled by a message broker, which usually groups messages into channels (or topics). Publishers send messages to channels, while subscribers receive messages by subscribing to relevant channels.



The pub/sub system's decoupled nature means your apps can potentially scale to limitless subscribers. A significant advantage of adopting the pub/sub pattern is that you often have only one component that has to deal with scaling WebSocket connections - the message broker. As long as your message broker can scale predictably, it's unlikely you'll have to add additional components or make any other changes to your system to manage an unpredictable number of WebSocket connections.

While pub/sub is an excellent choice from a scalability perspective, it's worth mentioning that most pub/sub solutions come with challenges and limitations of their own, such as message delivery issues (guaranteed delivery, ordering, exactly-once semantics). Such shortcomings usually stem from the decoupled nature of pub/sub and you will need to find ways to mitigate them.

HOW WE SOLVE IT

Ably's pub/sub implementation

Ably is an enterprise-grade edge messaging platform that uses pub/sub to deliver data in realtime. Our platform offers a straightforward and scalable way to distribute data to end-user devices over a global edge network at consistently low latencies without managing or scaling infrastructure.

Our pub/sub APIs are feature-rich and enable you to deliver billions of messages every day to millions of users. We've designed our pub/sub service to overcome traditional limitations and provide data integrity guarantees (ordering, guaranteed delivery, exactly-once semantics) at scale.

Learn more about:

Ably's pub/sub messaging capabilities

Ably's data integrity guarantees

3. A fault-tolerant system

When you're building audience engagement features, providing reliable, uninterrupted experiences that match and exceed user expectations is crucial. You need to think about engineering dependability and fault tolerance into your system.

To make your system fault-tolerant, you must ensure it's redundant against instance or even datacenter failures. This implies distributing your infrastructure across multiple availability zones in the same region. You might want to take it further and have your infrastructure distributed across multiple regions. This way, even in the event of a region failure, your system would still be operational.

Of course, building a distributed, fault-tolerant system that can dependably scale to handle thousands or even millions of WebSocket connections is a difficult challenge, which involves significant engineering and DevOps efforts, and infrastructure-related costs. Most of the time, it's more practical to offload at least parts of this complex burden to a fully-managed battle-tested solution.

HOW WE SOLVE IT

Ably is built with fault tolerance in mind

Our platform is engineered to provide regional and global fault tolerance. We've designed Ably around statistical risks of failure, building in sufficient redundancy at regional and global levels to ensure continuity of service, even in the face of multiple infrastructure failures.

Learn more about:

Message durability and quality of service across a large-scale distributed system

Engineering dependability and fault tolerance in a distributed system

4. Fallback transports

Despite widespread platform support, some proxies don't support the WebSocket protocol or terminate persistent connections, and some corporate firewalls block specific ports, such as 443 (the standard web access port that supports secure WebSocket connections). In addition, theWebSocket protocol is still not entirely supported across all browsers.

Imagine you've developed an interactive platform with WebSocket connections to deliver features such as interactive presentations; Q&As; virtual quizzes and polls; and breakout rooms and chat. Your product attracts interest from businesses and organizations of all types and sizes, as it's ideal for workshops, training sessions, and daily meetings. So what do you do to ensure you can offer your features to customers, knowing that you may not be able to use WebSocket connections in all situations?

If you foresee clients connecting from within corporate firewalls or otherwise tricky sources, you most likely need to consider supporting fallback transports, such as XHR streaming, XHR polling, or long polling. Developing your own fallback capability is a complex process that takes time and resources. In most cases, to keep engineering complexity to a minimum, you are better off using an existing WebSocket-based solution that includes fallback options.

However, it's not enough to only have fallback capabilities. In the context of scale, another essential thing to consider is the impact fallbacks have on the availability of your system. Let's assume you have tens of thousands of concurrent users using your interactive platform and there's an incident that causes a significant proportion of the WebSocket connections to fall back to long polling. There are some notable differences between solutions that use WebSocket connections and those that use long polling. The latter is much more resource-intensive on servers and comes with additional complexity in its implementation. Your server layer needs to be elastic with sufficient capacity to deal with the increased load.

HOW WE SOLVE IT

Fallbacks and limitless capacity

At Ably, we not only provide our own protocol built on top of a WebSocket solution, but we also support multiple fallback options.

We operate with a 50% global capacity margin to deal with instant surges in demand, and we can quickly increase capacity even further to deal with any increased load, such as the one created by WebSocket connections falling back to other transports.

Learn more about:

The Ably protocol

The fallbacks we support

Ably's elasticity and availability

5. Connection continuity

It's common for devices to experience changing network conditions. Devices might switch from a mobile data network to a Wi-Fi network, go through a tunnel, or experience intermittent network issues. Scenarios like these may lead to WebSocket connections being dropped, and they will have to be re-established.

For some use cases, **data integrity is crucial** and, once a WebSocket connection is re-established, the stream of data must resume precisely where it left off. Think, for example, of features like live chat, where missing messages due to a disconnection or receiving them out of order leads to a poor user experience and causes confusion and frustration.

If resuming a stream exactly where it left off after brief disconnections is important to your audience engagement use case, your system needs to be stateful, and you need a resilient storage strategy. Here are some things you'll need to consider:

- **Caching messages in front-end memory.** How many messages do you store, and for how long?
- **Moving data to persistent storage.** Do you need to transfer data to disk? If so, where do you store it, and for how long? How will clients access that data when they reconnect?
- **How does the stream resume?** How do you know which stream to resume when a client reconnects and where exactly to resume it from? Do you need to use a connection ID to establish where a connection broke off? Who needs to keep track of the connection breaking down - the client or the server?

HOW WE SOLVE IT

Ably's approach to handling reconnections with continuity

Ably messages, sent to consumers, can be placed in persistent storage rather than delivered in a fire-and-forget fashion. This means that previously delivered messages can be resent if something happened that prevented them from being processed on the client-side.

In addition, each message sent to users has an Ably-assigned serial number. This identifier is persisted on the client-side upon receipt of the message. In the event of a client crash or a disconnection, when the connection is restored, the client sends Ably the serial number of the last message it received. This way, the stream of data can resume exactly where it left off.

Learn more about:

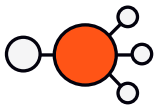
History/persisted data

Message durability and quality of service across a large-scale distributed system

Ably's data integrity guarantees

Additional challenges

When scaling a WebSocket solution to deliver shared live digital experiences to millions of users, here are some additional challenges to consider:



Load balancing strategy

Should you use sticky sessions or non-sticky load balancing? What allocation method (e.g., round-robin, least connections) is the best for your specific use case? How do you make your load balancers elastic? How do you route traffic away from unhealthy datacenters?

Managing backpressure

You can use several mechanisms to manage backpressure, such as controlling the rate at which you're sending data to consumers, buffering, or conflation/delta compression.

Terminating connections

What load shedding mechanism are you going to use to prevent overload, and how will you progressively allow new WebSocket connections to be established?

Connection detection

Sending heartbeats to check if a WebSocket connection is alive directly impacts the scalability and dependability of your system, especially when you're handling millions of concurrent WebSocket connections. The more frequent the heartbeats, the more load created. Your system must have the capacity to deal with it.

System abuse

You need to take the necessary precautions and ensure you can deal with system abuse, such as DoS attacks. These incidents aren't always the result of malicious actions. They can also arise from legitimate operations that cause a massive number of new connections in a short period, which can seriously hinder your system's performance and availability. You need to think about implementing exponential backoff and rate-limiting mechanisms.

Final thoughts

As you design a platform for audience engagement built on the WebSocket protocol, you will encounter numerous challenges to providing dependable and uninterrupted experiences that satisfy attendees and event organizers alike.

Ably provides enterprise-grade edge messaging to deliver data at low latency. We make additional data integrity guarantees to overcome the issue of operating over an unreliable network, and higher-level functionality to aid building rich live experiences, such as presence, history, and authorization.

Every day, we underpin a range of shared, live and collaborative experiences as we deliver billions of messages to millions of users, for thousands of companies, at consistently low latencies over a secure, reliable, and highly available global edge network.

As a fully-managed platform, Ably enables companies to simplify engineering, minimize DevOps overhead, reduce infrastructure costs, increase development velocity, and scale to meet any demand. Organizations such as Hopin, Mentimeter, Tennis Australia, Vitac, or Onedio trust us to provide edge messaging infrastructure to underpin their products.

If you want to talk more about WebSocket solutions or if you'd like to find out more about Ably and how we can help you build scalable audience engagement platforms or apps, get in touch or sign up for a free account.

LEARN MORE

References and further reading

WebSockets - A Conceptual Deep Dive

Ably: A platform engineered around Four Pillars of Dependability

Case Study: Wooclap

Case Study: Onedio

About Ably

Ably is an edge messaging platform for developers. There's no infrastructure to provision or manage, just an evolving suite of SDKs and APIs that give you the freedom and flexibility to power shared live experiences with a few lines of code. Our mathematically modeled system design provides a global edge network that brings users closer to your app; unique data ordering and delivery guarantees ensure a seamless end-user experience; a legitimate 99.999% uptime SLA is underpinned by fault tolerant infrastructure; and instant elasticity enables effortless scale.

Brands like HubSpot, Toyota, and Webflow trust Ably to power shared live experiences like business-critical chat, order delivery tracking, or document collaboration for millions of simultaneously connected devices around the world.

Sign up for a free account

Get in touch

Visit
ably.com

Call
+44 20 3318 4689 (UK)
+1 877 434 5287 (USA)

Email
hello@ably.com