TECHNICAL WHITEPAPER

Using the Ably realtime platform at scale

Written by Paddy Byers, CTO

ably

Table of contents

Introduction

This note contains an overview of the considerations when structuring a large application using the Ably platform. The aim is to provide a commentary on the limits that exist within the system, and how this impacts how applications are organised.

System overview

Ably is generally hosted on AWS EC2 infrastructure and uses associated services. A cluster typically exists in multiple regions, but there is no specific constraint on how it is deployed; some clusters run in a single region only, and others exist in multiple regions.

The deployment in each region is as independent as possible; each is responsible for terminating subscriber connections, handling client REST traffic, managing channels and routing messages between publishers and subscribers.

When there is activity on a given channel in multiple regions, then messages are passed peer-to-peer between the regions; there is no single point through which all messages are routed in a channel.

The principal functionality in the system is deployed in two roles:

- **Frontend**: these processes handle REST requests and realtime connections (comet and websocket);
- **Core**: these processes perform all central message processing for channels.

These scale independently in each region according to load; we monitor CPU and memory metrics on each instance, and trigger autoscaling by aggregate metrics on those parameters.

Horizontal scalability is achieved by placing load on the cluster in a way that can take advantage of additional capacity that arrives in response to scaling events. The primary means for doing this is the placement of channels across the group of core processes; each core has a set of pseudorandomly-generated hashes, and consistent hashing is used to determine the location of any given channel. As a cluster scales, channels relocate, ensuring that load remains evenly distributed.

In the case of frontend processes, autoscaling starts more frontend instances which are added to the loadbalancer; these new instances acquire a fraction of new connections and REST requests.

Although the system is designed to scale horizontally - so in principle any arbitrary load can be handled by making a cluster sufficiently large - there are always practical limits. This document explores those limits and discusses how applications can be structured so as to avoid limits being a problem in practice.

Platform limits

The discussion considers limits in a number of categories and is structured according to the principle limits that exist on any account.

Message processing

Maximum number of channels

(channels)

This is a limit on the number of concurrent channels active at any time; each active channel occupies memory on a core process and, to a lesser extent, CPU, even when idle. In a shared cluster, this ensures that the account does not consume a disproportionate fraction of the cluster capacity. In a dedicated cluster, which can freely scale to meet the demand of a single account, the practical limits are cost - the cost of running the cluster at a given scale - and the ultimate cluster size. Ultimate cluster size constraints are discussed below.

Messages in a single channel

(messages.perChannel.*)

There are two limits here: the `maxRate` (the instantaneous rate of messages on the channel) and `maxBandwidth` (the instantaneous data rate for inbound messages on the channel). This is the most important practical limit when deciding how to structure an app that has a very high volume of messages.

The design constraints are that:

1. a single channel is the unit of distribution in the Ably system - that is, any given subscriber, if subscribed to a channel, is sent all of the messages on a channel;
2. all core message processing for a single channel takes place on one core instance, which means that there is a maximum message rate that is sustainable on a single channel, and scaling the cluster will not increase that limit.

So, when structuring an app, we recommend that sufficiently many channels are used that the load can be distributed horizontally to the required degree. We do not recommend setting the `messages.perChannel.maxRate` limit to anything higher than 200 messages/second. Although a single instance can handle many times this number, it is also possible that multiple heavily-loaded channels could be placed at the same instance. So in order to be reasonably confident that channel throughput on a single core process is not a bottleneck, the maxRate needs to be set to a level that is comfortably below the theoretical single-channel maximum.

When assigning messages to channels, also bear in mind point (1) above; using too few channels will also risk an unnecessary increase in the total number of messages, as messages might be sent to subscribers that do not really need them.

A future development we are exploring is the implementation of sharded channels - this is essentially a way that allows the traffic for a single notional channel to be split into multiple physical channels. A mechanism to support this is currently under development, although so far the applications will be limited to internal channels and metachannels, where the number of shards required can be controlled fully by the Ably system and does not depend on external factors.

Maximum message size

(messages.maxSize)

This is the maximum size of the payload of any message. This is limited by default to 64KB, with a hard system limit of 2MB.

This limit is imposed because, as the size of each message grows, both network load and memory pressure increase, both on core and frontend processes. Core memory pressure can, in principle, be managed by increasing the number of core processes (ie distributing the channels across more instances) but there is also an impact on the throughput achievable on each individual channel, so scaling can only address the issue to a certain degree. In addition, the processing cost of each individual message increases with size, and this has an impact on message transit latency.

The forthcoming delta subscription feature will help with one aspect of this - reducing the outbound bandwidth when there is fanout of large messages to subscribers. However, calculation of a delta is at least O(N) in the size of the message, and this also imposes a limit on the maximum allowable message size. We are testing the delta feature with base message sizes of up to 2Mb, and this leads to a delta calculation time of tens of milliseconds; this will therefore impose a much lower maximum message rate on a channel, and will add that processing time (and any associated queueing delays) to the transit latency.

Maximum channel creation rate

(channels.maxCreationRate)

This is the maximum global aggregate instantaneous rate of channel activation. It exists because there is work required to activate any channel, and there is a maximum rate at which any individual instance can do that. For certain workloads, the channel churn rate is the dominant load, as opposed to per-message processing. The maxCreationRate on an account is determined based on the total channel number but is typically limited to hundreds of channels per second. The capacity of the system in respect of this limit can be increased by increasing the number of core instances. Channel creation rate is unlikely to be the factor that is the ultimate limit to scaling of any individual app, but it is a factor that impacts the cost of maintaining a cluster.

Applications can also take steps to minimise the impact of this limit by avoiding unnecessary churn or chatter in channel activity: for example if large numbers of channels are continuously being activated and deactivated then either the application can preserve the activity of channels, or reduce the number of those channels by sharing them in some way.

Instantaneous message rate

(messages.instantaneous.*)

These limits derive from the hourly and monthly message limits on an account (which themselves are used to determine the overall charge for a given package). These instantaneous limits ensure that large surges in message rate are rate-limited. In a shared cluster this means that a single app cannot have an adverse effect on the overall load on the cluster to the detriment of other apps. In a dedicated cluster, these instantaneous limits ensure that very sharp spikes in load are rate-limited, because it is not possible to scale a cluster rapidly enough to be able to handle arbitrary spikes.

If an application does have very short-term load spikes then the following approaches are possible:

- ensure that the cluster is provisioned with capacity to handle those spikes (even if it means that the system is operating below-capacity most of the time);
- allow those spikes to be rate-limited while autoscaling reacts to the load increase. If a load spike is sustained for 30s then the primary load metrics (CPU and memory) will initiate scaling, but then that capacity can take up to 5 minutes to come into effect (dependent on AWS response times and instance boot/initialisation time);
- If the time of a load spike is known in advance then it is possible to schedule scaling explicitly ahead of that event.

Connection handling

Maximum number of connections

(connections)

This is a limit on the number of concurrent connections active at any time; as with channels, a connection occupies memory on a frontend process and also CPU, even when idle. In a shared cluster, this ensures that the account does not consume a disproportionate fraction of the cluster capacity. In a dedicated cluster, which can freely scale to meet the demand of a single account, the practical limits are the same as for channels; cost - the cost of running the cluster at a given scale - and the ultimate cluster size. Ultimate cluster size constraints are discussed below.

Maximum publish rate on a connection

(messages.perConnection.maxInboundRate)

A single connection is terminated at a single frontend process, and this process has a maximum capacity for processing inbound messages. This requires us to impose a limit on the maximum publish rate on a single connection; for publishing at sustained high rates, beyond that supportable by a single connection, it is necessary to distribute the work to multiple senders with independent connections, or to use the REST API (which is able to loadbalance the work to multiple frontend processes). We recommend that an absolute maximum limit of 200 messages per second is used. Applications affected by this limit should consider either migrating to use the REST API for publishing, or partition the work of publishing so that it can be undertaken by multiple separate connections.

Maximum subscribe rate on a connection

(messages.perConnection.maxOutboundRate)

As with the maximum publish rate it is necessary to impose a limit on the rate of outbound messages processed on a connection, in order that the single instance that terminates that connection is not overloaded. Even though the maximum rate on a single channel is limited, it is possible for a connection to attach to many channels, so it can exceed the single-channel rate by 2 orders of magnitude.

Very high message rates on a single connection also lead very quickly to memory pressure in the frontend as the data rate reaches the sustainable data rate of the connection; if the system is unable to send messages on the connection at a rate that matches the arrival of those messages, then messages are dropped.

We recommend that an absolute maximum limit of 200 messages per second is used on an outbound connection. The bandwidth of a client connection is outside our control, and many clients will have a much lower sustainable rate. Applications affected by this limit might be required to send fewer individual messages by sending updates less frequently.

Ably currently does not support conflation as a way of managing the backlog of messages waiting to be sent on a given connection, but this is a planned future development.

Maximum connection creation rate

(connections.maxCreationRate)

As with channels, there is work required to establish a connection and for certain workloads, the load associated with connection establishment dominates over the load associated simply with maintaining mostly-idle connections. The maxCreationRate is calculated as a function of the maximum number of connections but would typically be limited to thousands of connections per second. The capacity of the system in respect of this limit can be increased by increasing the number of frontend instances.

The primary impact of this limit is that, for certain applications, the frontend capacity of the system is scaled to meet the connection churn load, not the absolute connection count; this is just a cost impact.

A feature currently in development that will help to address this limit is that of stateless connections; these are connections that subscribe only to a fixed set of channels that are specified at the time of connection establishment. The setup cost of a stateless connection is dramatically lower than a conventional Ably realtime connection.

Connection creation rate in practice only becomes a significant limitation in applications that experience massive connection rates at the start of a specific event; online contests or sports events, for example, may experience millions of connections being established within a few minutes. In such applications, the time of the surge event is usually known ahead of time, and capacity can be pre-provisioned to support the increased instantaneous connection creation rate.

Maximum number of channels attached on a single connection

(channels.perConnection)

This limit exists to ensure that there is a bound on the size of the connection state for any single connection. This means that the work is bounded when a connection resumes after a disconnection. The default limit is 200 channels per connection, but this is somewhat arbitrary; we have not explored where the ultimate limits are.

The practical impact of this limit is seen in applications where there are long-lived connections, but the channel set changes continuously, and would grow in an unbounded way if there was no limit. Applications are encouraged to avoid this limit by actively managing attachments - ie detaching channels when they are no longer needed - and by partitioning work across multiple connections if there really is a need to perform processing on very large numbers of channels in one place.

Reactor Integrations

Ably's Reactor Integrations are a set of mechanisms that enable messages to be forwarded to various destinations such as cloud functions, instead of only to connected client subscribers.

Instantaneous rate limits

(reactor.*.instantaneous.*)

These are global aggregate rate limits for the processing of messages to various reactor destinations. These exist to ensure that bounds exist on the resources allocated to those functions - compute and memory resources for request processing, and other resources associated with multiple concurrent outbound connections. In addition, as with other instantaneous rate limits, these limits ensure that load spikes do not result in the core processing layer being overloaded.

Applications can address the effects of these limits in several ways:

- choose destinations that are appropriate to the traffic type. For example we support higher instantaneous limits for AWS Kinesis, say, as compared to regular webhooks, because there is much greater assurance that the endpoint can reliably sink the required message volume;
- Consider whether or not a more fine-grained channelFilter can reduce the peak load. It might be appropriate to split traffic into separate channels if this means that reactor rules need to be enabled on only a subset of the channels.
- Consider the general approaches to handling of spikes: over-provisioning, scheduled provisioning, or simply allowing messages to be rate-limited until such time as scaling can take place to handle the spike.

Max concurrency

(reactor.* maxConcurrency)

This is a limit on the number of in-flight requests at any given time to a given reactor destination. As with message rates, this limit exists to ensure there is a bound on the resources allocated to a given endpoint. Applications that experience issues with this limit should primarily consider whether or not a different destination type would be more appropriate: Ably should not be consuming resources waiting for reactor endpoints to respond; if there is an issue with getting messages to a particular destination, then some intermediate queue should be used so that messages can be drained from Ably into that queue before onward processing.

Presence

Presence messaging is subject to all of the limits associated with the processing of normal messages, with one principal additional limit.

Maximum members present on a channel

(presence.maxMembers)

This limit is to ensure that there is a bound on the size of a presence set, which affects:

- synchronisation of the presence set at the client end of a realtime connection;
- migration of presence data when a master role for a channel relocates.

There is also an antipattern whereby the members present in a channel are also subscribed to presence updates for that channel, which results in a total message volume on the channel which is O(N^2) where N is the number of subscribed and present members. It is common for applications initially to be organised this way, only to become unsustainable as the user population grows. This limit is therefore only raised in specific cases where it is known that the N^2 problem does not exist.

REST API

The REST API is served by frontends. Client requests are distributed across the available frontend instances in a region by a loadbalancer. The client/server relationship is stateless, so the only relevant limits are instantaneous request rate limits.

API request rate limit

(apiRequests.instantaneous.*, tokenRequests.instantaneous.*)

As with the other instantaneous limits, these limits exist to ensure that excessive load is not generated when there is a spike in the request rate, so a cluster that has been sized on the basis of the monthly request limits has sufficient capacity to absorb load bursts.

As for instantaneous message rate limits, if an application does have very short-term load spikes for the REST API, then the following approaches are possible:

- ensure that the cluster is provisioned with capacity to handle those spikes (even if it means that the system is operating below-capacity most of the time);
- allow those spikes to be rate-limited while autoscaling reacts to the load increase;
- If the time of a load spike is known in advance then it is possible to schedule scaling explicitly ahead of that event.

Cluster size limits

Provided the system capacity scales horizontally in certain dimensions - such as numbers of channels and numbers of connections - then it is possible to achieve very significant scale simply by adding more instances to the cluster. However, this process itself has limits as the overheads of managing a large cluster become unsustainable.

An Ably cluster uses a gossip-based discovery layer to track all instances present in the cluster; this operates entirely peer-to-peer, and is used for discovery of the cluster topology and sharing of cluster parameters (addresses, ports, connection policies etc for all endpoints). This layer is also used for liveness detection: each node's visibility in the cluster depends on it being connected to the discovery layer and responsive to liveness checks. The discovery layer itself performs liveness detection on the discovery nodes via gossip.

One set of scalability limits of the platform derive from this structure: there are limits to the number of discovery nodes that can participate due to the scaling characteristics of gossip - the network traffic at any single gossip node scales linearly with the total cluster size if other parameters are kept constant (eg the time taken to detect a dead node).

Each node maintains a map of the entire cluster and this is also a theoretical limit on the ultimate size of the cluster - although there are obvious changes that can be made to avoid the need for every node to have visibility of every other node, if this became the limiting factor.

There are also second-order effects of having a very large cluster: the rate of cluster change events (eg occurring as a result of instance failure) is currently very low, but would grow linearly with cluster size. Deploying software updates becomes problematic - software updates that take hours to deploy fully to a large cluster would take days, without a new deployment strategy.

We have tested the existing architecture up to thousands of nodes without problems, and the system can scale up by at least one order of magnitude larger than the size of our existing main production cluster; the first limits that are reached under the present architecture are things that we know how to address when the need arises, so there is clear potential to scale beyond the scale that we have run in testing so far.

About Ably

Ably is a realtime messaging platform. We deliver billions of realtime messages every day to more than 50 million end-users across web, mobile, and IoT platforms. We power things like HubSpot's live chat, in-play scores for millions of Australian Open fans, and real time transit updates for three million Chicagoans.

Ably's platform is mathematically modeled and architected around Four Pillars of Dependability. Where other providers sacrifice integrity of data for low latencies, or vice versa, Ably guarantees message ordering and delivery without sacrificing latencies, fault tolerance or service availability. This approach means we can provide a realtime service guaranteed to operate within strict, transparent boundaries that are dependably predictable.

That's why developers trust Ably to build realtime capabilities in their apps. Our feature-rich platform includes multi-protocol pub/sub messaging, presence, push notifications, free streaming data sources from across industries like transport and finance, and integrations that extend Ably into third-party clouds and systems like AWS Lambda and RabbitMQ.

Businesses use Ably to distribute their streaming data to other businesses. This allows them to offload the engineering and data delivery challenges of providing data streams that are performant, reliable, and available to consume with various protocols.