Eventual Consistency Solutions

Solution ASYNC1 : In-DB Event Store

This solution trades the strong consistency guarantee of the SYNC solutions for the performance and horizontal scaling possibilities of an Event Sourcing asynchronous system.

For this solution, we require 3 separate tables:

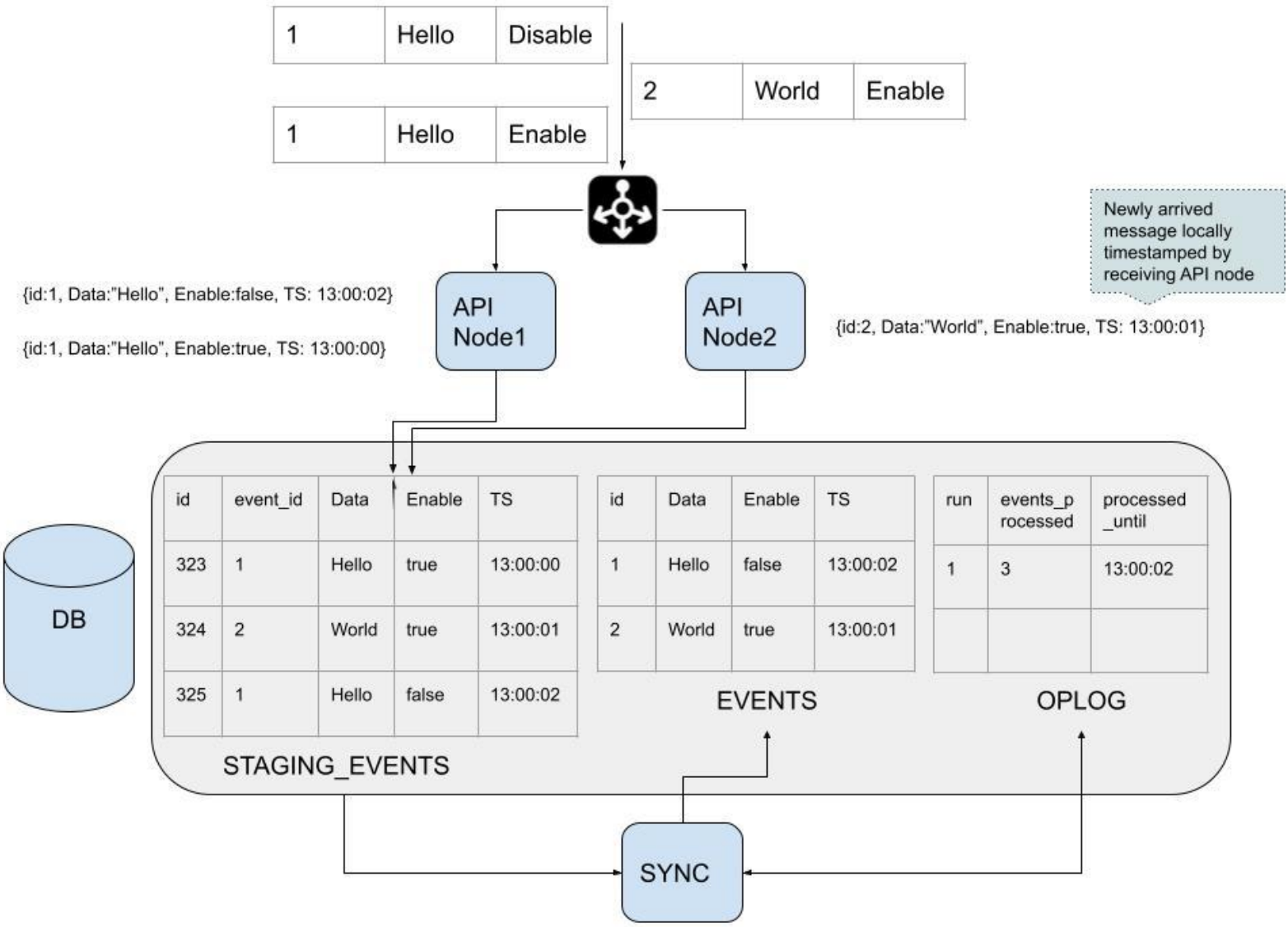
- STAGING_EVENTS
- STATUS
- OPLOG

Instead of putting the responsibility to INSERT and UPDATE event records directly into the main STATUS table, we will limit the functionality of the API to only write (INSERT) any new messages(events) into a new “STAGING_EVENTS” table. Thus it is only synchronous wrt recording the event in this “stating” table, not in the table that reflects the most recent state for each event/message.
This staging table contains all of the fields of the message, along with its own PK “ID”, separate from the event ID that is sent in the message itself. The original event ID carried in the actual call to the API actually is a foreign key in this table.
Thus, any new message that hits the API, regardless of its ID or Timestamp, gets simply inserted into STAGING_EVENTS. This is a fairly low-impact and low-latency operation, as we are simply guaranteed that the event was recorded.

An additional service named SYNC, which runs periodically (and independently), is tasked to take “raw events” from the STAGING_EVENTS table, process them and INSERT/UPDATE the original STATUS table - where the message ID is the primary key (unique). The timestamp for the last event processed is stored in the OPLOG table, which is read when SYNC starts a new iteration, and updated as part of the transaction that inserts/updates events into the MESSAGES table. The OPLOG table effectively servers to monitor the progress of the SYNC service, as it updates the STATUS table messages with new events from STAGING_EVENTS.

The SYNC service runs with a slight delay behind real-time, to allow any momentary re-ordering to “settle”. The lag time is TBD, but can likely be ~1-5 seconds behind the current time, to be adjusted based on system requirements.

The STATUS table is then the “authoritative” source of information for any queries regarding any existing message. Its state is lagging slightly behind (it is dependent on the SYNC service for updates), thus this is an eventually-consistent system, however it will never allow for reordering of events.



ASYNC1 Architecture Diagram

Limitations and Trade-offs

The most obvious limitation in this case is that we no longer have hard consistency guarantees. We can tune the timing (and performance) of the SYNC job to be as close to real-time as possible, but we can never guarantee consistency. In other words, the user of the API should be aware that if he immediately queries for data he recently updated, there is a chance he will not get the most up to date response. This is the case with many highly scalable systems, such as most NoSQL databases (e.g. MongoDB, unless it’s explicitly configured for consistency - at the cost of performance).

Solution ASYNC2: Events in message queue

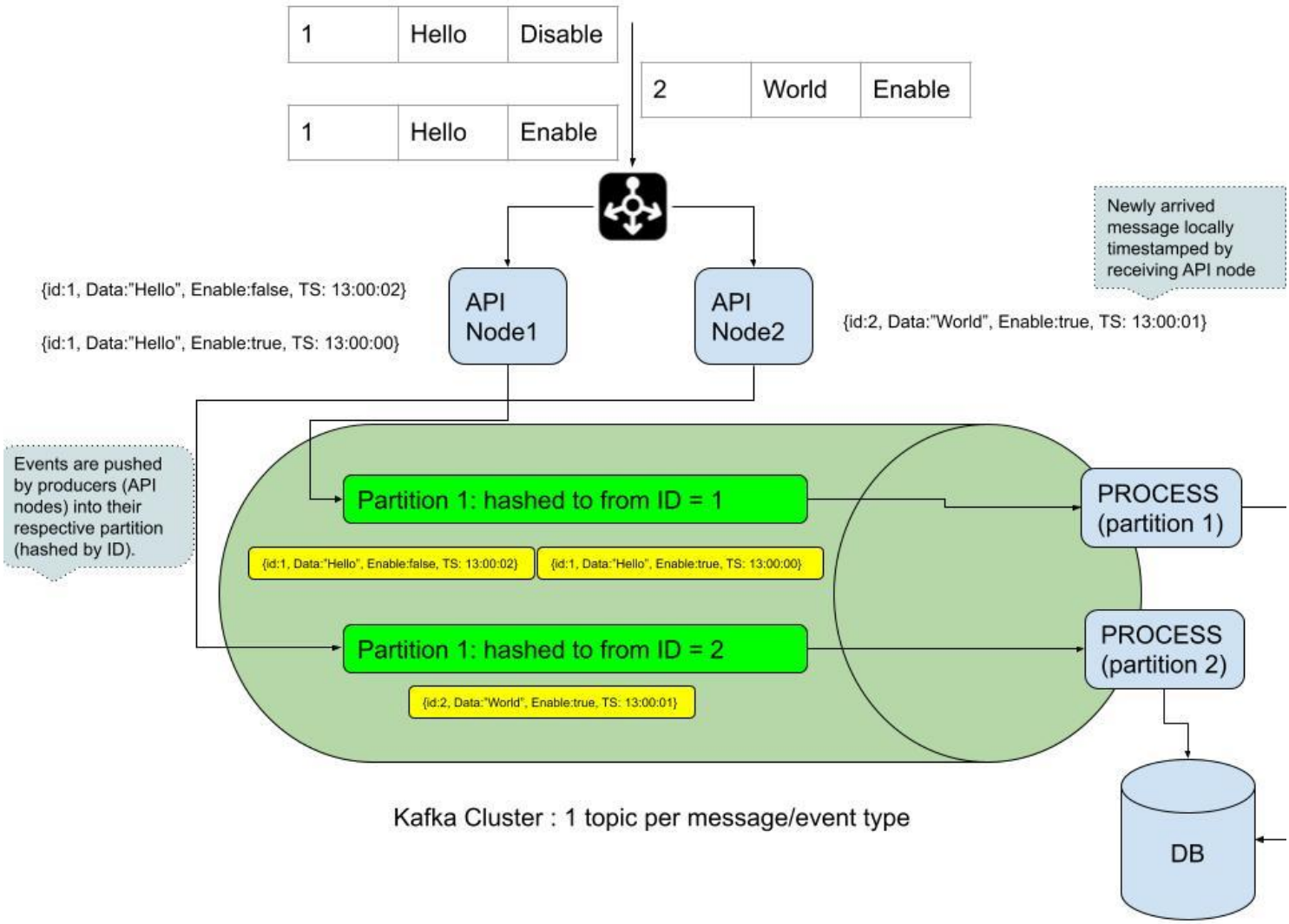
This solution is similar to ASYNC1, in the sense that we process each update (message) as an event, and do not guarantee strong consistency. However, for even better horizontal scaling, we are using Kafka as the queuing system.

The API service is configured to receive user messages (events) and push them into a single Kafka topic, after timestamping them locally. The topic is partitioned based on a hash of the message ID - this guarantees that there is no re-ordering of messages as long as they belong to the same set of IDs. In other words, if messages with IDs {1,4,8,13} are mapped to Partion 1, then the relative order of their updates is preserved - within Partition 1.
The API nodes act as the **Producers** in this case.

We then spin up a single instance of a Consumer service (lets call it **PROCESS)** per partition, which is responsible for reading events/messages from its partition, processing them in memory and periodically writing updates to the database.

The database schema design changes somewhat in this approach, since we no longer need the **STAGING_EVENTS** table (as the events are effectively stored in Kafka). However, we still make user of the **OPLOG** table to keep track of the offset for each consumer group (effectively 1 per partition). Also, we retain the **EVENTS** table - which is still our authoritative state for any recorded/updated messages. This allows us to gracefully resume event processing in the event that one of the consumers dies.

Scaling this system is fairly easy - we simply increase the number of partitions (and spin up additional **PROCESS** nodes to handle consuming those partitions).



Limitations & Trade-offs

The main trade-off in this design (aside from an increased complexity due to the introduction of a messaging system) is balancing the frequency of writes into the main **EVENTS** table.
We can achieve near real-time consistency by making frequent writes/updates, at an increased risk of temporarily allowing reordered events and the decreased performance due to having to lock rows more often.
Alternatively, we can allow each consumer instance to buffer events for several seconds, before pushing the most up-to-date state to the **EVENTS** table - thus allowing for a higher overall throughput of updates.