Notification Systems and Real-Time Feeds with Message Queues

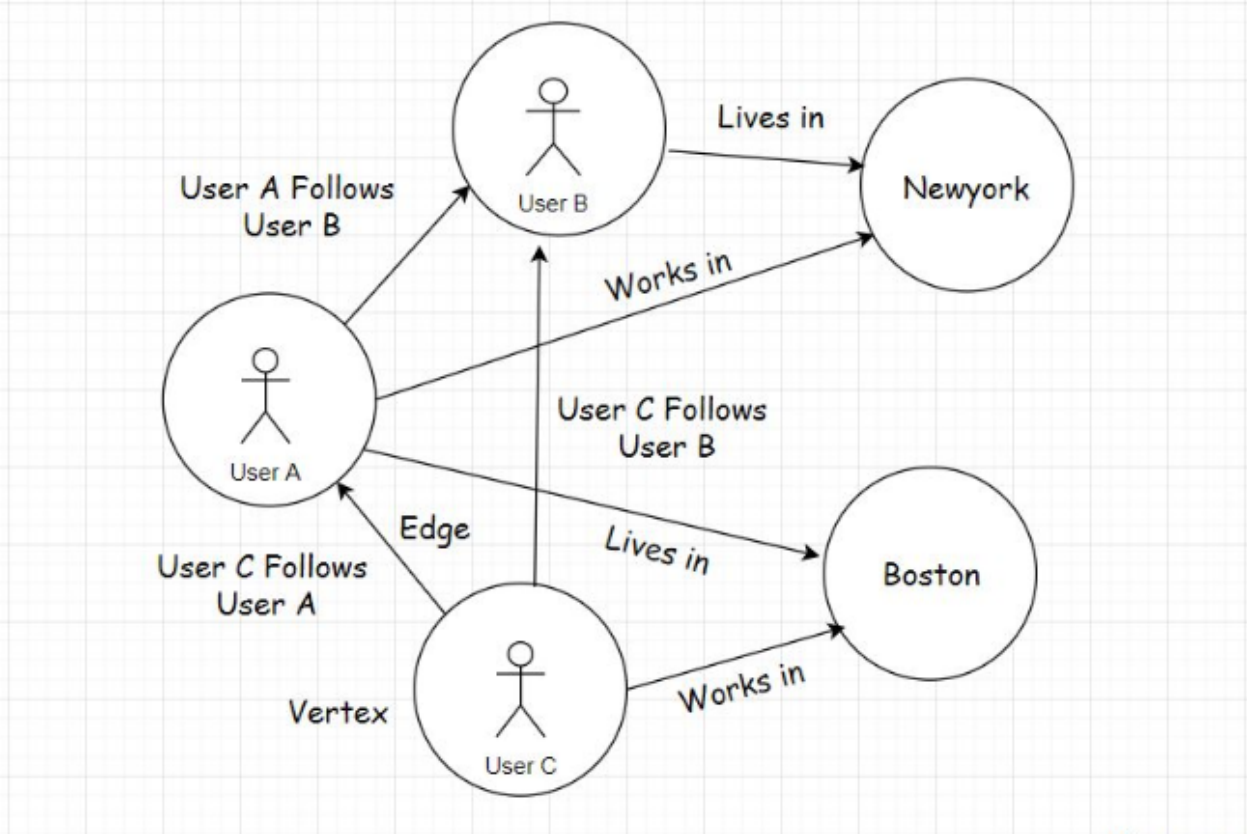
How notification systems and real-time feeds are designed with the help of message queues.

Notification systems and real-time feed are complex modules in today's modern Web 2.0 applications. They involve a lot of tasks such as understanding user behavior, recommending new and relevant content to the users on the platform, ingesting data from different sources, matching user behavior with that data, and so on. They also leverage machine learning in order to be more effective.

Notification system - use case

Imagine building a social network like Facebook using a relational database. A message queue is required to add asynchronous behavior to our application.

In the application, a user will have many friends and followers. This is a many-to-many relationship, like a social graph. One user has many friends, and they would be friends of many users.



So, when a user creates a post on the website, the application will persist it in the database. There will be one **User table** and another **Post table**. Since one user will create many posts, it will be a one-to-many relationship between the user and their posts.

As we persist the post in the database, we also have to show the information posted by the user on the home page of their friends and followers, even send notifications if needed.

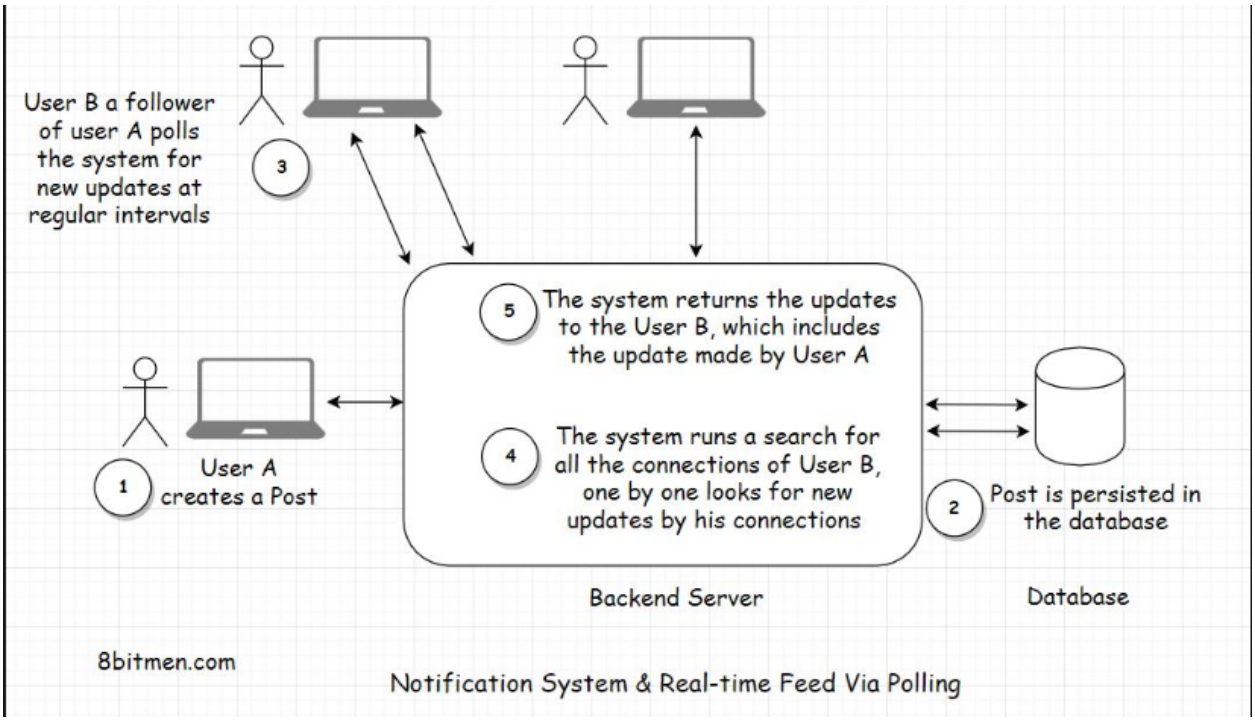
Pull-based approach

One straightforward way to implement this without the message queue will be to have every user on the website poll the database at regular short intervals if any of their connections have a new update.

For showing notifications on a certain user profile, we will query all the connections of the user from the database and then run a check on every connection one by one if any new information is posted by them.

If there are new posts created by the user's connections, the query will pull them all and display the posts on the home page of the user's profile. We can also send the notifications to the user about the new posts, tracking them with the help of a **Boolean** notification counter column in the User table and adding an extra AJAX poll query from the client for new notifications.

Whenever the database query finds new posts, it changes the new notification counter to true. When the counter is true, responding to the Ajax poll request, the application sends a notification to the user that you have recent posts made by your connections.



There are two major downsides to this approach -

1.) We are polling the database so often. This is expensive. It will consume a lot of bandwidth and put a lot of unnecessary load on the database.

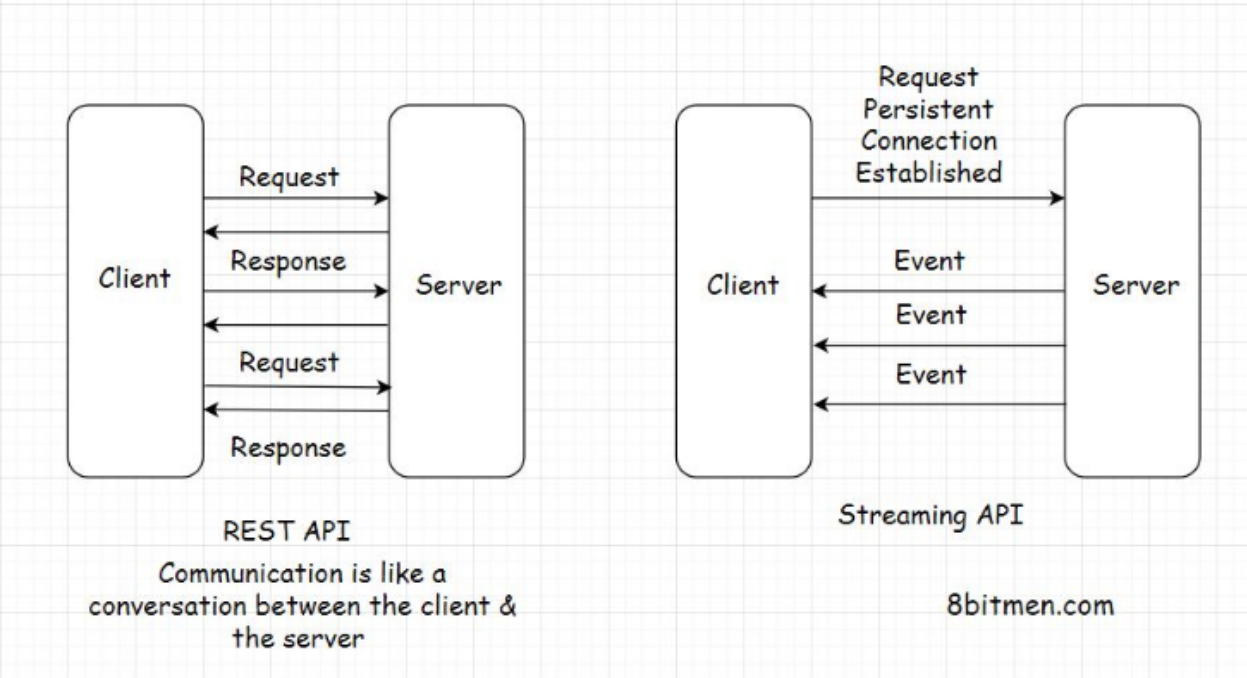
2.) A user's post displayed on the home page of their connection's will not be in real-time. The posts won't show until the database is polled. We may assume this as real-time, but it is not really real-time.

Push-based approach

Let’s make our system more performant. Instead of polling the database now and then, we will take the help of a message queue.

In this scenario, when a user creates a new post, it will have a **distributed transaction**. One transaction will update the database, and the other will send the post payload to the message queue. Payload means the content of the message posted by the user.

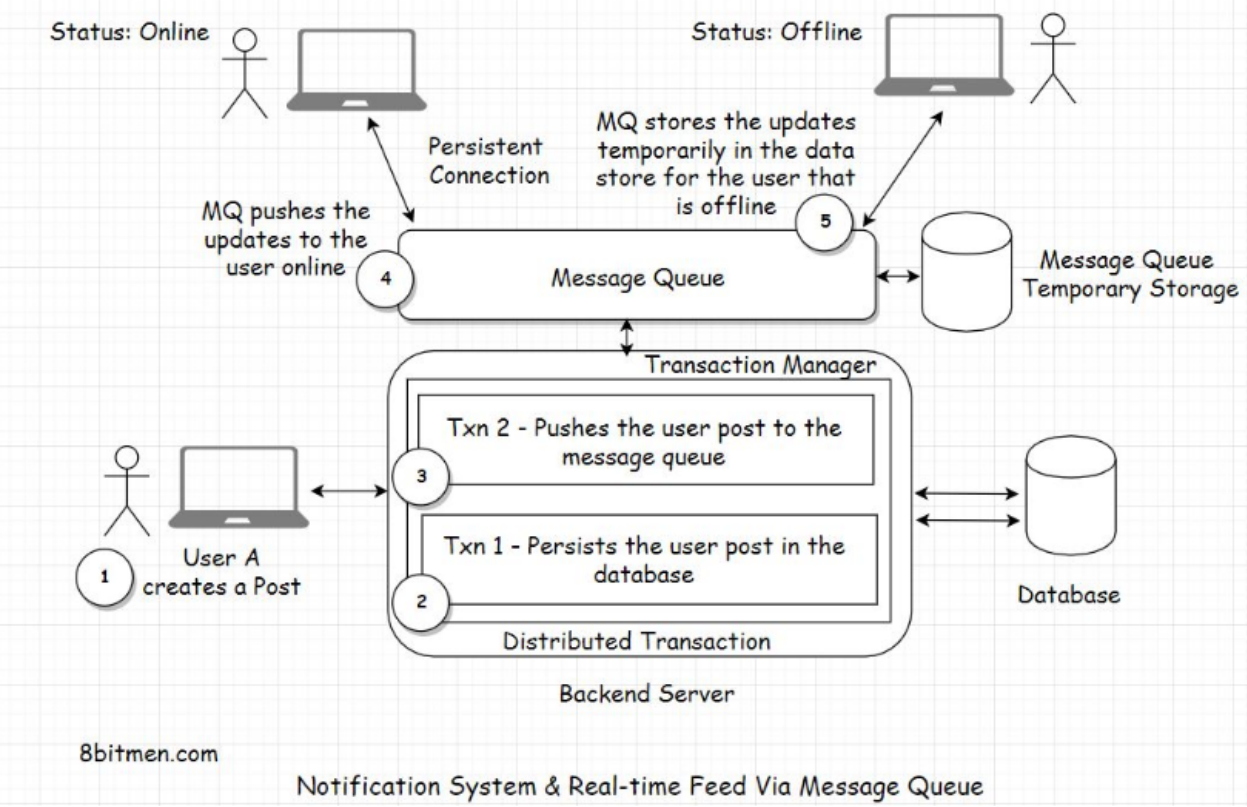
Notification systems and real-time feeds establish a persistent connection with the database to facilitate streaming data in real-time. We have already been through this.



On receiving the message, the message queue will asynchronously and immediately push the post to the user’s connections that are online. There is no need for them to poll the database regularly to check if any of their connections have created a post.

We can also use the message queue temporary storage with a TTL for storing the payload until the connections of the user come online and then push the updates to them. We can also have a separate key-value database to store the user’s details required to push the notifications to their connections, like their connection list and stuff. This would avert the need to poll the database to get the connections of the user every time the user creates a post.

So, did you see how we transitioned from a pull-based approach to a push-based approach with the help of message queues? This shift will spike the application's performance and cut down a lot of resource consumption, saving truckloads of our hard-earned money.



There are several ways we can handle distributed transactions. Though the transactions are distributed, they can still work as a single unit. In case the database persistence fails, the application will roll back the entire transaction. There won't be any message push to the message queue either.

What if the message queue push fails? And the database transaction succeeds? Do you want to roll back the transaction, or do you want to proceed? The decision depends entirely on you, how you want your system to behave.

Even if the message queue push fails, the message isn't lost. It can still be persisted in the database.

When the user refreshes their homepage, you can write a query to poll the database for the new updates, taking the polling approach we discussed initially as a backup.

Or you can totally roll back the transaction, even if the database persistence transaction is a success but the message queue push transaction fails. The post hasn’t gone into the database yet as it is generally a two-phase commit.

We can always write custom code to manage the distributed transactions or can choose to leverage the distributed transaction managers that the frameworks offer.

Post creation is an event that a user triggers in the application. Similar events are liking a post, photo, video, watching a live-stream and so on.

Just like posts, message queues can push these events too to the connections of a user.

How does LinkedIn identify its users online?