LEVERAGING RABBITMQ QUORUM QUEUES AND ASP.NET CORE FOR A ROBUST AND SCALABLE NOTIFICATION SYSTEM

Sipilov I.

Sipilov Ivan — Head of Platform Development,
CITADEL HEDGE FUND,
FOUNDER OF NANNYSERVICES.CS,
NEW YORK, USA

Keywords: *Real-time notification systems, RabbitMQ quorum queues, Raft Consensus Algorithm, ASP.Net Core notification architecture, fault tolerance in distributed systems, scalability of notification systems, flexibility in notification templates, event-driven notification generation, message-driven architecture, publish-subscribe model, data enrichment for notifications, reliable message delivery, high-volume notification handling, message replication, distributed system reliability.*

Introduction

In the age of real-time systems and highly dynamic business processes, companies increasingly face the challenge of creating notification systems that are both flexible and scalable, capable of handling diverse types of notifications in large volumes while maintaining high performance and fault tolerance. This paper outlines the design of a notification system built with RabbitMQ quorum queues, which use the Raft Consensus Algorithm, and ASP.Net Core. The system addresses several key requirements, such as:

1. **Flexibility**: The ability to easily add new notification types with custom templates and data models.
2. **Real-time responsiveness**: Instant reaction to business events occurring in core systems by triggering appropriate notifications.
3. **Scalability**: Handling high volumes of notifications efficiently, without degradation in system performance.
4. **Reliability and Fault Tolerance**: Ensuring no notifications are lost due to system failure and avoiding duplicate notifications.

By leveraging RabbitMQ quorum queues, the system achieves high reliability, scalability, and fault tolerance, essential for modern distributed systems where real-time notifications are critical.

Problem Statement

To achieve the objectives outlined above, the notification system addresses several key problems that are inherent in modern distributed applications:

1. Flexibility in Adding New Notification Types

Businesses constantly evolve, and with that, their communication strategies must adapt. A system that requires manual intervention for adding or modifying notification types quickly becomes a bottleneck. The notification system must provide an architecture where new types of notifications can be added with their own unique templates and data models, without the need for extensive re-engineering.

2. Real-time Reaction to Business Events

The system needs to react to business events, such as user transactions or status updates, in real-time. These events can be critical to customer engagement and satisfaction, especially in industries like finance, e-commerce, or social platforms, where immediate feedback is expected. Triggering the correct notification for a given event type must happen instantly, without delay.

3. Handling High Volumes of Notifications

Scalability is essential when dealing with systems that generate large volumes of notifications, such as millions of email, SMS, or push messages per day. The system must handle these volumes while maintaining performance, ensuring that notification generation and delivery occur without bottlenecks.

4. Ensuring Fault Tolerance and Reliability

Distributed systems are prone to failures, whether partial (component failure) or total (system-wide outage). It is crucial that the system remains highly available and ensures that no notifications are lost in the event of a failure. Furthermore, the system should avoid duplicating notifications, even if failures occur mid-process.

Technologies and Architecture

RabbitMQ Quorum Queues

RabbitMQ is a message broker that implements the AMQP (Advanced Message Queuing Protocol). It provides powerful mechanisms for message queuing, routing, and handling, enabling reliable asynchronous communication between different parts of a system. RabbitMQ quorum queues offer robust message replication and fault tolerance using the Raft Consensus Algorithm, which ensures that messages are replicated across multiple nodes, preventing data loss in case of node failures.

Quorum queues in RabbitMQ are a critical component in this architecture because they guarantee that messages, such as notifications, are not lost, even if individual nodes in the cluster experience failures. Raft ensures that at least a majority of nodes agree on the state of the queue, meaning the system can handle various failure scenarios without risking data loss.

ASP.Net Core

ASP.Net Core provides the framework for building the core business logic and integrating the various subsystems. It allows for real-time event handling and provides the necessary APIs to handle different types of notifications (e.g., emails, SMS, and push notifications). ASP.Net Core integrates seamlessly with RabbitMQ, allowing for efficient message processing and routing.

Solution Architecture

The system is based on a **message-driven architecture** that allows for real-time processing of business events and decouples the generation of notifications from their delivery. This design uses RabbitMQ quorum queues to handle message persistence, fault tolerance, and message routing.

I. Event-Driven Notification Generation

The notification system is designed to react immediately to business events, using a **publish-subscribe** model facilitated by RabbitMQ. In this architecture, the decoupling of event producers and consumers enables high scalability and flexibility.

1. Event Declaration and Publishing

Each subsystem responsible for executing business transactions (e.g., subscription handling, payment processing) declares a set of events it will publish upon completion of those transactions. These events represent significant business activities that may trigger a notification. For example, a subsystem responsible for managing user subscriptions might declare events like:

- **"Premium subscription purchased"**
- **"Subscription renewal"**
- **"Subscription expiration"**

After the business transaction is processed, these events are published into the system. The notification system listens to these events to determine when to generate notifications. The events are dispatched into the integration bus, which uses RabbitMQ as the transport and storage layer. Each event type is assigned its own **fan-out exchange** within RabbitMQ.

In a fan-out exchange, messages are broadcasted to all queues bound to the exchange. This means that multiple subsystems, including the notification system, can independently subscribe to the same events and process them without interfering with each other.

2. Subscription and Queue Binding

Any subsystem that is interested in consuming these events binds its own RabbitMQ quorum queue to the corresponding fan-out exchange. For instance, the notification subsystem would bind its queue to the events it is responsible for handling, such as those related to user actions like purchasing a subscription or renewing it.

This approach provides significant advantages:

- **Decoupling**: Producers of events (such as the payment processing system) are completely decoupled from the consumers (like the notification system). This ensures that changes in one subsystem do not directly impact others.
- **Scalability**: Multiple consumers can independently bind to the same event, allowing the system to scale horizontally. For example, both a logging service and a notification service can listen to the same "subscription purchased" event without stepping on each other.
- **Fault Tolerance**: Using quorum queues ensures that messages are reliably stored even during system failures. Because these queues are durable, they persist through crashes and reboots.

II. Data Enrichment: Transforming Raw Events into Rich Notifications

Once an event reaches the notification system via its queue, the process of data enrichment begins. This step is crucial for generating rich, personalized notifications that contain all the necessary information and adhere to the correct template.

1. Template and Configuration Retrieval

The notification system must determine how to handle each event, which involves looking up the correct configuration settings for each notification type. These configurations are stored in a relational database and can be dynamically edited through an administrative interface.

The configuration settings include:

- **Notification channels**: Specifies the channels through which the notification will be sent (e.g., email, SMS, push notifications).
- **Template information**: Identifies the file names of the templates used for the specific notification type.
- **Business rules**: Defines conditions that control whether the notification should be sent, such as toggles for sending notifications under specific circumstances.

These configurations allow the system to handle notifications for a wide range of business events dynamically, without needing hardcoded logic for each event type. If the business requirements change, such as introducing a new notification channel or updating the message template, these updates can be made in the admin interface without requiring changes to the underlying code.

2. Data Augmentation

For many notification types, the raw event data itself is insufficient for creating a rich notification. To generate a complete notification, additional data may need to be fetched from external systems.

For example, consider an event that indicates a user has purchased a premium subscription. To generate a complete notification, the system may need to retrieve additional details about the user (e.g., their name, email preferences) or their subscription plan (e.g., expiration date, payment method). These details are fetched from external systems such as:

- **User data services**: To obtain personal information about the user.
- **Subscription services**: To retrieve information related to the purchased plan.
- **Search subsystems**: To quickly access projections of relevant data using systems like Elasticsearch.

This external data is combined with the original event data to form a complete **notification model**. This model includes both the business event information (e.g., user purchased a subscription) and any additional information necessary for the notification (e.g., the user's name and email address).

By using a **request-response** model with external services, the notification system ensures that it has all the relevant data needed to enrich notifications. The request to these external systems occurs asynchronously, so the main notification process is not delayed while waiting for data retrieval.

3. Template Rendering and Final Notification Model

Once all the necessary data is collected, the system prepares the final notification by rendering the appropriate template. This involves filling in placeholders in the template with the data retrieved during the enrichment process. The resulting notification is fully customized for the recipient, ensuring that it is both informative and tailored to their situation.

At this point, the system determines the notification channels to use (e.g., SMS, email, push) and creates a separate message for each channel. Each message is then sent to the appropriate downstream process for delivery.

III. Reliable Notification Delivery (continued)

2. **Asynchronous Processing**: The **Message Gate** application pulls messages from the RabbitMQ quorum queues and handles the actual delivery of notifications to external providers. By processing these notifications asynchronously, the system achieves two important benefits:
   - **Decoupling the notification generation from delivery**: This allows the system to generate notifications as fast as possible, without being blocked by slow or unavailable external services.
   - **Improved throughput and performance**: As messages are placed into queues for later processing, the notification system can continue generating new notifications without waiting for the delivery system to catch up.
3. **Error Handling and Retries**: If an error occurs while attempting to deliver a notification (e.g., if an external email provider is down or an SMS gateway is temporarily unavailable), the system does not remove the message from the queue. Instead, it marks the message as pending and continues retrying until the notification is successfully sent. This retry mechanism is particularly important for ensuring **fault tolerance** and guaranteeing **no message loss**, as the system can recover from transient errors without dropping notifications.
4. **Guaranteed Delivery**: By utilizing RabbitMQ quorum queues, which are built on top of the Raft Consensus Algorithm, the system ensures **guaranteed message delivery** even in the face of hardware or software failures. Messages remain in the queue until they are successfully processed, ensuring that no notifications are lost. The Raft algorithm ensures consistency and fault tolerance by replicating messages across multiple nodes, preventing data loss even if individual nodes or brokers go down.
5. **Idempotent Processing**: To prevent duplicate notifications from being sent, the system ensures **idempotent message processing**. This means that even if a message is reprocessed due to a failure or retry, the

same notification will not be sent multiple times to the same recipient. This is achieved through tracking the state of each notification and verifying whether it has already been successfully delivered before processing it again.

Data Enrichment: A Deeper Dive into Transforming Raw Events

The data enrichment process plays a crucial role in transforming raw business events into meaningful, personalized notifications. This involves querying external systems for additional data, performing business logic transformations, and rendering notification templates with the enriched data.

1. Multi-level Data Augmentation

The raw events coming from the business transaction system usually contain only basic information. For example, an event like "subscription purchased" might only include a user ID and a timestamp. However, a rich notification (e.g., an email to the user) often requires more detailed information. To provide a meaningful and user-friendly notification, the system needs to augment the event with additional data from multiple sources.

- **User Information**: In many cases, the notification system will need to fetch detailed user information, such as the user's full name, contact details, notification preferences (e.g., email or SMS), and language preferences. This data is typically stored in a user management or identity service.
- **Business Entity Data**: For events tied to specific business entities (e.g., purchases, subscriptions), the system needs to retrieve data about the related entities. For example, in a subscription purchase event, the system might need to retrieve details about the purchased plan, such as the price, duration, and renewal date. This ensures that the notification provides the user with all the necessary details.
- **External Service Integrations**: To further enrich notifications, the system may interact with additional external services. For example, the system could call a **recommendation engine** to suggest additional services to the user based on their recent activity, or it might query an **Elasticsearch** service to pull the latest data projections about the user's account or recent activities.

Request-Response Communication with External Systems

The notification system interacts with these external data sources using an asynchronous **request-response** pattern. This ensures that data is fetched without blocking the notification processing pipeline. Each data request is sent to the appropriate external system, and the response is asynchronously processed by the notification system. This design prevents slow responses from blocking the generation of other notifications.

To optimize performance, the system also employs **caching** mechanisms, storing frequently accessed data (such as user profiles) in a fast-access cache (e.g., Redis) to reduce the number of external requests. This greatly improves system efficiency, especially in high-throughput scenarios.

2. Business Logic Transformations

Once all the necessary data has been gathered, the system performs several **business logic transformations** to prepare the final notification content. These transformations include:

- **Personalizing the notification content** based on user preferences, such as preferred language or time zone.
- **Applying conditional logic** to determine the channels through which the notification will be sent. For example, if a user has opted out of receiving SMS notifications, the system will automatically skip that channel.
- **Formatting data** for presentation. For example, dates might be formatted according to the user's locale, or prices might be displayed in the appropriate currency.

3. Template Rendering

After the data has been enriched and the necessary transformations have been applied, the system moves on to rendering the final notification content. This is done using pre-defined **templates** stored in the system's configuration. These templates define the structure of the notification for each channel (e.g., email, SMS, or push notification).

The system uses a **template engine** to replace placeholders in the template with the enriched data. For example:

- In an email notification template, placeholders for the user's name, subscription details, and purchase date might be replaced with the actual values retrieved during the data augmentation phase.
- In an SMS notification template, shorter, more concise placeholders are replaced with the appropriate data, formatted for the SMS medium.

Once the templates are rendered, the final notification messages are prepared for delivery to the appropriate channels (e.g., email, SMS, push notifications).

Conclusion

By leveraging RabbitMQ quorum queues and ASP.Net Core, the system achieves a highly scalable, flexible, and fault-tolerant architecture for generating and delivering notifications in real-time. The use of RabbitMQ quorum queues ensures **message reliability** and **system resilience**, allowing the system to recover gracefully from failures without losing notifications. The publish-subscribe model decouples event producers from consumers, allowing the system to scale independently and accommodate diverse notification types without significant re-engineering.

Furthermore, the **data enrichment** process ensures that notifications are rich, personalized, and informative, enabling the system to deliver the right message to the right recipient at the right time. This approach is not only scalable but also flexible enough to handle the evolving needs of modern business systems, where real-time notifications are critical for customer engagement and satisfaction.

The system's architecture solves several key challenges faced in notification systems, including:

- **Handling high volumes of notifications efficiently**.
- **Guaranteeing reliable message delivery**.
- **Providing flexibility for adding new notification types** without downtime.
- **Ensuring that no notifications are lost** during system failures, and preventing duplicate notifications.

This architecture can serve as a model for designing real-time notification systems in various industries, from finance to e-commerce to social platforms, where timely and reliable notifications are essential.

References

1. RabbitMQ Quorum Queues and the Raft Consensus Algorithm. Source: RabbitMQ Documentation [Electronic Resource]. URL: https://www.rabbitmq.com/quorum-queues.html
2. ASP.Net Core Overview. Source: Microsoft Docs. [Electronic Resource]. URL: https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core
3. Raft Consensus Algorithm. Source: The Raft Consensus Algorithm Website. [Electronic Resource]. URL: https://raft.github.io/
4. Event-Driven Architecture for Scalable Applications. Source: Martin Fowler's Blog. [Electronic Resource]. URL: https://martinfowler.com/articles/201701-event-driven.html
5. Message Queuing in Distributed Systems. Source: IEEE Xplore Digital Library. [Electronic Resource]. URL: https://ieeexplore.ieee.org/document/9152165
6. Designing Scalable and Reliable Notification Systems. Source: High Scalability Blog. [Electronic Resource]. URL: http://highscalability.com/scalable-notification-systems
7. Using RabbitMQ for Message Queuing in Microservices. Source: DZone. [Electronic Resource]. URL: https://dzone.com/articles/rabbitmq-for-microservices
8. Data Enrichment in Event-Driven Systems. Source: O'Reilly Online Learning. [Electronic Resource]. URL: https://www.oreilly.com/library/view/event-driven-architecture/9781492062322/
9. Introduction to Distributed Systems Design. Source: InfoQ. [Electronic Resource]. URL: https://www.infoq.com/articles/introduction-to-distributed-systems-design/
10. Handling High Volumes of Notifications Efficiently. Source: AWS Architecture Blog. [Electronic Resource]. URL: https://aws.amazon.com/architecture/handling-high-volume-notifications/