The Art of Reliability and Asynchrony: Building Robust Distributed System Communication via Outbox

Nitin Gupta

Dublin, CA, USA
Email: nitingupta.sri7@gmail.com (N.G.)
Manuscript received August 20, 2025; accepted October 7, 2025; published November 26, 2025.

***Abstract*—Modern distributed systems offer significant benefits for scalability, resiliency and flexibility but these benefits come with complex challenges of inter-service communication, reliable messaging and data integrity. A critical challenge comes when application need to persist data and send message as an atomic operation, a failure in one can cause data inconsistency. This paper presents the Outbox pattern as a robust architecture paradigm to address these challenges. The Outbox pattern achieves this by saving both the data and message as an atomic transaction in a local database, an asynchronous process runs later to send a message to another system by reading the database for stored message. This approach enhances resiliency, data integrity and promotes loose coupling along with independent scalability. It also presents various use cases that can be applied to different industries like banking, fintech, healthcare.**

***Keywords*—architecture pattern, cloud native, distributed system, reliable messaging, robust design**

I. Introduction

Modern digital system has grown exponentially to meet the increasing demand of IT workloads, which has led the organizations to incline more towards the distributed system. These distributed systems not only come with great benefits of individual scalability, fault tolerance, improved reliability and flexibility but also introduces challenges for inter service communication, message delivery, network dependency, data consistency and data integrity. Often these challenges when overlooked or not handled in efficient manner, eradicates the benefits of distributed systems and led to complex architecture and non-reliable digital system.

In distributed system ensuring reliable message delivery is a great concern, and with all the challenges it can lead to message lost or duplicated. There are various messaging patterns and technologies that helps to deliver messages exactly once, but it comes with its own complexity [1]. A common problem comes in application when data persistent and message publication need to be done as an atomic operation. When these two actions are not tightly coupled and a failure in one can led to inconsistent data or non-reliable communication. Consider a scenario when database commit is successful, but message publishing fails then two system will have diverge state as the consumer expect a message and maybe doing some actions for business logic.

This paper provides detail on architecture paradigm that provides robust and efficient design where reliable messaging in critical to ensure data integrity and consistency. The Outbox Pattern helps to perform two actions related to business data and message as an atomic operation, in traditional system these were handles as a two-phase commit which introduces overhead and reduced availability [2]. The outbox pattern ensures that message get persisted alongside the business data as apart of single local transaction thus ensuring the operation is successful or failure. The message delivery will happen eventually as it persists in the system and can be retried even there is failure due network error or system unavailability.

The outbox pattern ensures reliable message delivery in loosely coupled and message-driven architectures. This pattern helps to reduce the error handling and removes the complexity associated with distributed coordination in complex software system [3]. The paper outlines various use case and different industries like- banking, healthcare, ecommerce where pattern can be applied to deliver efficient and robust software solutions.

II. Architecture and Design

Asynchronous and reliable messaging is a critical aspect of communication in distributed system for inter service communication or communication with external system. The outbox design leverages the transactional capability of the database by making sure that message get persisted in the same datastore along with business data as an atomic operation. A separate asynchronous process in the background runs or polls the database for the updated messages and publishes to the event bus. The reliability and robustness of the design can be highlighted through various critical aspects:

(1) Data Integrity: The outbox pattern helps to achieve reliable messaging by delivering messages in correct order and accurately to maintain data consistency across systems. This ensures there is no data corruption, and all system view has same state visibility.
(2) Loose Coupling: The pattern helps to communicate asynchronously with different system thus reducing tight coupling. It helps systems to scale independently and provides flexibility for system maintenance [4].
(3) Fault Tolerance: It helps the system to be more resilient in case of network issues or unexpected events and help to recover gracefully. Outbox pattern ensures messages are not lost during such events and consistency is maintained in different systems. Since messages are saved so during any network failure these is no risk of message loss, and it can be replayed.
(4) Scalability: As the pattern helps to communicate asynchronously this allows systems to be loosely coupled. This helps to scale different systems independently to enhance performance and availability.

As shown in Fig 1 there are few key components in outbox design pattern that ensures reliable message delivery, through persisting in database and sending to event bus by maintaining data consistency across system. These

components are:

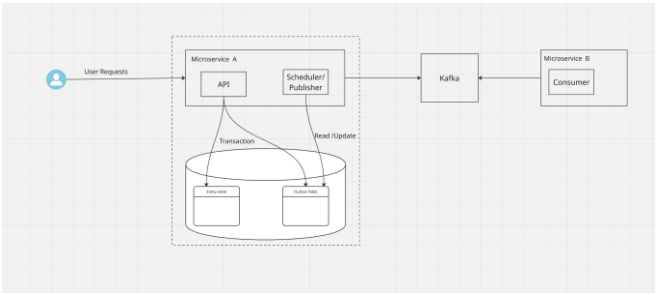

Fig. 1. Outbox design pattern.

(1) Outbox Table: This database entity is used to store messages in the same database where application data is stored.
(2) Transaction Management: It helps to save application data and messages in same transaction and helps to maintain consistency between application state and message state.
(3) Publisher: A publisher can be a background job that runs periodically in the service to check for database updates in outbox table or any event driven strategy that react to database change. It reads the messages from table and publishes to event bus like – Kafka and updates the status in table
(4) Consumer: A consumer is other async process running in other service which consumes the message and perform relevant action
(5) Retry and Error: Retry logic and retry count helps to retry the messages which were not published due to failures. The number of retries can be customized and once it exhausts this message can be send to deal letter topic from there further action can be done depending on the business need. This ensure messages are not lost due to failures.

III. Methodology

When working with sensitive information that require reliable and asynchronous communication in different systems, the Outbox design pattern is well suited for distributed system architecture. This pattern ensures data integrity and consistency by saving application data and message as an atomic operation in local database. This operation of saving to database is performed as a single transaction so it is either success or rolled back from outbox entity along with application entity [5]. Since messages are stored in database, a separate asynchronous process runs that publishes these messages.

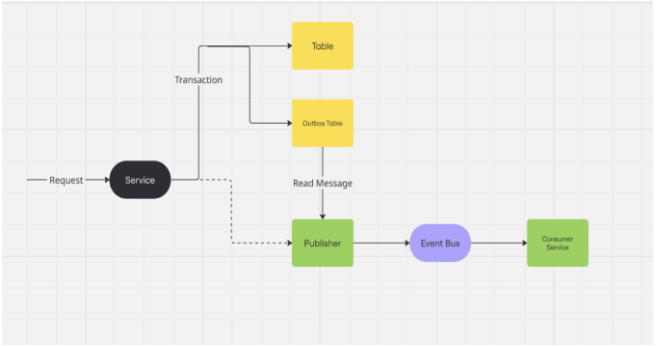

Fig. 2 Outbox pattern data flow.

There are various ways this process can be designed which can vary from use case, some of the strategies for this process are:

(1) Scheduling–A scheduler job is implemented which runs at regular interval to check outbox entity in database for updates and retrieves the pending messages that needs to be published.
(2) Event Driven–A listener is configured on outbox entity which publishes message for any change in outbox entity

The pattern finds significant use in modern digital system where reliable messaging is essential due to various compliance related to audit, tracing and data consistency [6]. Some of the real-world system where this can be applied are:

A. Audit and Compliance

In banking or fintech system every transaction performed by system needs to be recorded for auditing and compliance. Example – A user initiates the fund transfer, a system needs to perform various operations like – Debit/ Credit money, send notification to the users and notify record in Datawarehouse for audit and compliance or to auditing service. The Outbox pattern guarantees that all these are done as part of transaction i.e. debit\credit and send message persisted in database and an asynchronous process runs in background, to notify user along with the message delivery to auditing service [7]. This ensures the data consistency by delivering the messages and making visible the same state of data to different system [8].

B. Distributed System

In Distributed system computing each service needs to communicate with each other to maintain consistent state across the system. This communication needs to be reliable and loosely coupled to take advantage of distributed system [9]. In any ecommerce application when order is placed in order service a notification needs to be sent to inventory, billing and shipping service to act accordingly. This coordination can be complicated but outbox pattern helps here to solve this problem by managing the atomicity of saving the order and message. Outbox pattern ensures notification is delivered to other services, to maintain consistent order state across the application.

C. Loose Coupling

To fully harness the benefits of distributed system it should be loosely coupled so it can be scaled independently. Outbox patterns help to achieve this by segregating the process of message sending to reliable asynchronous process so each system can work independently without tying them for message delivery. As in healthcare industry when patient book an appointment a notification needs to send to the patient for appointment confirmation. Here outbox pattern plays an important role of sending appointment notification to the patient along with booking in the system.

D. Rate Control and Throttling

Enterprise application connects with various third-party systems to provide end to end solution or to provide extended functionality. But the use of this third party sometime comes with the cost, to work efficiently with optimum cost Outbox pattern plays a significant role. Example–third party system

has defined a cost tier with rate limiting, in case of high load scenarios outbox pattern ensures that this rate limit is not exceeded with denial of request and customer experience is not affected.

IV. Evaluation and Considerations

The outbox design pattern is a critical architecture solution for reliable messaging. It provides a robust distributed system by handling challenges for data consistency and preventing data loss. After evaluating the outbox pattern in different use case across various industry domains it has surpass with key benefits from other approaches.

(1) Atomic–Since application data and message are stored as a part of same transaction in local database the atomicity is maintained along with data consistency.
(2) Reliable–The outbox pattern ensures that messages are delivered, that allows for reliable messaging. If database transaction is successful then message publishing is guaranteed, in case of failure delivery can be retried.
(3) Simple–Outbox pattern removes the need for two phase commit or complex transaction management mechanism for message delivery. This not only removes the complexity in the digital system but also allows them to be loosely coupled.
(4) Scalable–As systems are loosely coupled through asynchronous messaging they can be scaled independently which provides flexibility in digital application.

The outbox pattern proves to be great architecture in distributed world, but key considerations need to be done while solving the problem of reliable messaging. Database Bottleneck – Since the pattern relies on database it can become a bottleneck thus reducing the system performance and scalability, so evaluation of database selection and throughput needs to be done for optimal results. Storage – As messages are also getting stored, a strategy needs to be developed when these can be removed otherwise it can burden the storage system. Message Duplication – Since outbox pattern retries message in case of fault consumer system needs to handle message idempotency as they can receive same message more than once [10].

V. The Catalyst

The whole system robustness of reliable messaging i.e. message needs to be delivered to the sub-system is driven by the asynchronous delivery system and retry in case of failure. An efficient retry system with n number of retries makes system more reliable and efficient, an extra guard around the system can be placed by creating a dead letter queue which holds the messages that are not delivered even with n retries. The dead letter queue can be audited for failed message and re-run, so data consistency is maintained in whole eco-system.

The system efficiency and robustness can further be enhanced by creating different retry strategies depending on the use case like – Fixed delay retry where retry will be done predefined fixed interval or exponential retry system which will give more breather to the system to recover before next retry. These capabilities help to recover the digital system automatically without manual intervention.

Fault Tolerance: It helps the system to be more resilient in case of network issues or unexpected events and help to recover gracefully. Outbox pattern ensures messages are not lost during such events and consistency is maintained in different systems. Since messages are saved so during any network failure these is no risk of message loss, and it can be

VI. Conclusion

This design has demonstrated a reliable and robust message delivery system by leveraging the database atomicity in a single transaction and removing the complicated process like two phase commits which is essential for distributed system design [11]. It provides the flexibility and loose coupling along the subsystem so each system can be scaled and developed independently. It not only removes the dependency on complex architecture but also provides a way where systems can communicate reliably, which is essential for certain domain like banking, fintech, healthcare and ecommerce for their digital applications [11].

The design helps to shield the application against the fault and help with self-healing capability through retry. Apart from providing as a reliable messaging system it can also be used for rate control when calling the downstream where system has strict rate limiting enforced. As illustrated through practical use cases where the design can be applied like – auditing, communication and loose coupling the design helps to achieve overall system availability without degrading the performance.

As needed in distributed system for system to perform independently without causing impact or degrade the performance of other system, they need to be loosely coupled. The outbox design pattern helps to achieve all these goals with further enhancing the reliability of the digital world. Though the use of database can be tricky in case of very high load scenarios put proper design and consideration as shown in the paper can help to overcome that. Overall, the design when implemented thoughtfully can be a great solution to various proper in modern digital systems.

Conflict of Interest

The author declares no conflict of interest.

References

[1] L Magnoni, "Modern messaging for distributed sytems," *Journal of Physics: Conference Series*, vol. 608, 16th International workshop on Advanced Computing and Analysis Techniques in physics research (ACAT2014) 1–5 September 2014, published under licence by IOP Publishing Ltd, Prague, Czech Republic.
[2] G. Samaras, K. Britton, A. Citron *et al.*, "Two-phase commit optimizations in a commercial distributed environment," *Distrib Parallel Databases*, vol. 3, pp. 325–360, 1995.
[3] D. Malkhi and M. K. Reiter, "An architecture for survivable coordination in large distributed systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 2, pp. 187–202, 2000, doi: 10.1109/69.842262.
[4] M. A. Bauer *et al.*, "A distributed system architecture for a distributed application environment," *IBM Systems Journal*, vol. 33, no. 3, pp. 399–425, 1994, doi: 10.1147/sj.333.0399.
[5] B. C. Desai and B. S. Boutros, "Performance of a two-phase commit protocol," *Information and Software Technology*. vol. 38, no. 9, 1996, pp. 581–599.
[6] C. Richardson, *Microservices Patterns: With examples in Java*, 2018.
[7] E. Losiewicz-Dniestrzanska, "Monitoring of compliance risk in the bank," *Procedia Economics and Finance*, vol. 26, 2015, pp. 800–805.

[8] P. Kanhere and H. K. Khanuja, "A methodology for outlier detection in audit logs for financial transactions," in *Proc. 2015 International Conference on Computing Communication Control and Automation*, Pune, India, 2015, pp. 837–840, doi: 10.1109/ICCUBEA.2015.167.

[9] R. W. Watson, "Chapter 2. Distributed system architecture model," *Lecture Notes in Computer Science*, vol 105. Springer, 1981, Berlin, Heidelberg. https://doi.org/10.1007/3-540-10571-9_2

[10] Reliable Messages and Connection Establishment, Butler Lampson in Distributed Systems, ed. S. Mullender, published by ACM Press/Addison-Wesley, 1993

[11] K. Anita, P. K. Birman, T. A. Joseph, "Reliable communication in the presence of failures," *ACM Transactions on Computer Systems (TOCS)*, vol. 5, no. 1, pp. 47–76, 1987.