Formalization and Verification of Go-based New Simple Queue System

Danyang **Wang**[1], Jiaqi **Yin**[2], Sini **Chen**[1] and Huibiao **Zhu**[1]

[1]*Shanghai Key Laboratory of Trustworthy Computing. East China Normal University, Shanghai, China*

[2]*Northwestern Polytechnical University, Xi'an, China*

Abstract

NSQ (New Simple Queue) is a real-time distributed messaging platform implemented by Go language. It's designed to operate at scale stably and efficiently handle billions of messages per day. Its decentralized topology guarantees fault tolerance, high availability, and reliable message delivery. Operationally, NSQ is elastic to configure and deploy. With the broad application of the NSQ message system, its security and stability have attracted extensive concentration. Therefore, it is crucial to conduct a rigorous analysis and verification of NSQ's properties. In this paper, we employ process algebra CSP (Communicating Sequential Processes) to model the core functional modules of the NSQ. In addition, we utilize the model checker PAT (Process Analysis Toolkit) to verify five properties of the model, including divergence freedom, reachability, scalability, availability, and flow controllability. The verification results demonstrate that the NSQ system satisfies all the above properties, proving that the system has high flexibility and robustness while providing credible and efficient message delivery.

Keywords

NSQ, Messaging System, Communicating Sequential Processes(CSP), Modeling, Verification

1. Introduction

In the rapidly evolving era of the Internet, the explosion of users and services creates severe challenges for network applications. Conventional monolithic and vertical service architectures can no longer deal with such a volume of data. Distributed services are gradually becoming the mainstream architecture. As a foundational segment in distributed message systems, middleware [1] is important in decoupling, asynchronous communication, traffic clipping, and other issues. It can improve the performance and stability of applications. Therefore, message queue as a critical middleware acquires more attention in the Internet field.

With the evolution of technology, message queues are gradually maturing, resulting in a series of outstanding middleware, including ActiveMQ [2], RabbitMQ [3], Kafka [4], and RocketMQ [5]. These services decouple complex systems and enable asynchronous operations to reduce response times, providing a better user experience. Although the introduction of middleware can significantly improve the performance of a system, we must consider its potential problems and challenges, such as reduced availability due to unstable message queues and data inconsistencies due to concurrent communication. The system needs to introduce additional mechanisms

to ensure high availability and reachability of messages, which increases system complexity. Therefore, excellent message middleware should have high message processing efficiency, robustness, stability, and scalability.

NSQ [6] has emerged from these excellent middlewares in recent years. It is a distributed messaging platform based on Go language [9] with outstanding performance, robustness, and usability. This messaging platform is a user-friendly middleware for real-time messaging services, capable of managing hundreds of millions of messages. In addition, NSQ is fitted to the current concurrent Internet ecosystem due to Go's native strengths in concurrency. Go is a programming language with concurrency features, and its concurrency model was developed based on the process communication concept of CSP (Communicating Sequential Processes) [10, 11]. This feature makes the Go-based NSQ distributed system well-suited to the producer-consumer concurrency problem. Therefore, it is becoming popular within businesses and has also attracted the attention of researchers.

NSQ is suitable for distributed applications and systems that require asynchronous messaging, such as Social Media, Gaming, and other industries that require high concurrency. Until now, existing studies primarily focus on comparing different message queues performance, operability, and other characteristics [8] or delve into practical applications of NSQ [7]. To the best of our knowledge, there has yet to be research about the verification of its properties, which are significant for users. And the fundamental attributes of the system still need to be proven.

Employing a formal verification approach to verify NSQ's fundamental properties offers rigorous proof and
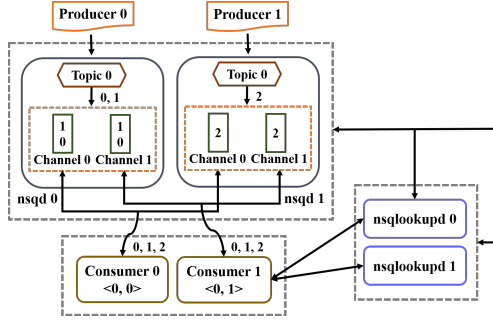
Figure 1: An Instance of the NSQ System

assurance, ensuring the system's correctness, reliability, and stability. This approach enhances confidence and credibility in the design, implementation, and deployment of the system, which is paramount for developers and users. Consequently, this paper bridges this research gap by adopting formal methods to analyze the NSQ system. We utilize process algebra CSP to formally model the core functional modules and basic workflow of the NSQ such as message publishing, subscription, registration, and querying. Subsequently, leveraging the model checker PAT [12], we verify five properties of the model, including *Divergence Freedom*, *Reachability*, *Scalability*, *Availability*, and *Flow Controllability*. Experimental results demonstrate that the NSQ distributed message platform can guarantee all these properties, proving that the system has outstanding flexibility and robustness.

The remainder of this paper is organized as follows. Section II briefly describes the NSQ system and process algebra CSP. In Section III, we use CSP to model four fundamental components in the NSQ message system. Furthermore, in Section IV, we employ the model checking tool PAT to implement the constructed models and verify five properties we defined. Finally, we summarize this paper and discuss future work in Section V.

2. Background

In this section we give a brief description of the NSQ's architecture and process algebraic language CSP.

2.1. NSQ - New Simple Queue

A typical architecture of the NSQ system is displayed in Fig. 1. Before furthering into the transmission mechanism of the NSQ, we should familiar with the following terms:

- **nsqd**: The NSQ daemon responsible for receiving and delivering messages. nsqd instances manage the actual message storage and distribution.
- **nsqlookupd**: The NSQ lookup daemon that manages topology information. It receives registration information and provides service discovery.

- **Topic**: It is a distinct stream of messages. An NSQ instance can have multiple Topics, each of which can have one or more Channels.
- **Channel**: It is a logical grouping of consumers subscribed to a given Topic. Each Channel receives a copy of all the messages for that Topic.

The Topic and Channel in the NSQ system are implemented by Go's channel data type. Go-chan builds on the idea of channel in CSP, it allows data transfers and synchronization operations between concurrent processes. A channel with cache space are also the natural way to express queue structure. Therefore, essentially NSQ's Topic/Channel is a buffered queue for message.

After learning the basic terms, we can introduce the NSQ system further from two core workflows: Message multi-cast and Message consumption.

2.1.1. Message Multi-cast

NSQ designs nsqd to handle multiple data streams concurrently. Each Topic can have one or more Channels. Topics multicast the received messages to Channels, and each Channel receives copies of messages. In practice, Channels map to downstream services that subscribe the Topics. Topics and Channels are not preconfigured but are created upon the first publication or subscription. Within nsqd, Topics and Channels independently buffer data to prevent lagging consumers from affecting other Channels. Messages are delivered to a randomly client when all clients are ready, achieving load balancing.

2.1.2. Message Consumption

Unlike many conventional message queues, NSQ maximizes performance and throughout by pushing data to the client instead of waiting for it to pull. This concept is called the RDY (Ready) state, constituting a form of client-side flow control. This RDY state is a pivotal performance parameter, allowing clients to modulate message by adjusting the RDY value. Once clients establish connections and subscriptions, they assert control over the flow of messages from nsqd by dynamically updating the RDY value.

2.2. CSP

Process Algebra CSP [10, 11] is a formal mathematical method that is widely applied in the design and verification of concurrent systems. This language has been successfully applied in modeling and verifying various concurrent systems and protocols [13, 14]. Parts of the CSP syntax used in this paper is defined as follows:

$$P, Q ::= SKIP \mid c?u \to P \mid c!v \to P \mid P \Box Q \mid P \| Q$$
$$\mid P \| \| Q \mid P [|X|] Q \mid P \triangleleft B \triangleright Q \mid P \, ; \, Q$$

- SKIP: The process terminates properly.
- $c?u \to P$: The process receives a value from channel c and assigns it to variable u, then starts P.

- $c!v \to P$: The process sends value v to channel c and then starts executing process P.

- $P \Box Q$: It depicts a general choice between process P and process Q.

- $P|||Q$: It illustrates interleaving. Processes P and Q run simultaneously and do not share any operations or variables.

- $P \triangleleft B \triangleright Q$: It portrays the execution of process P if the boolean expression b is true; otherwise, process Q will be executed.

3. Modeling

In this section, we construct the model of NSQ distributed architecture as illustrated in Fig. 1.

3.1. Sets, Messages and Channels

For a more detailed understanding of how the components within the NSQ system communicate and interact, we have laid out explanations for the fundamental building blocks used in the model: Sets, Messages, and Communication Channels.

Table 1

The correspondence between sets and constants/variables

Set	Constant / Variable
Module	P(Producer), C(Consumer), D(nsqd),LD(nsqlookupd)
ID	pid(producer ID), cid(consumer ID), did(nsqid), lid(nsqlookupd ID), tid(Topic ID), chid(Channel ID), msgid(message ID)
Command	FIN(Finish), REQ(Requeue), SUB(Subscribe), PUB(Publish), REGISTER(Register),MSG(Message),REP(Response), LOOKUPCHA(Lookup channel), LOOKUPD(Lookup nsqd)
Data	chList(registered channel list), dList(nsqd list with specific Topic)
ACK	OK, OUTTIME
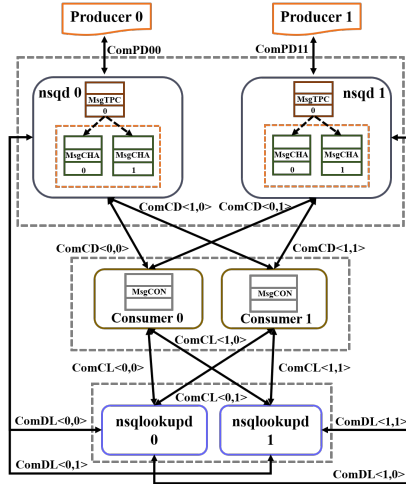
Figure 2: Channels of the NSQ System

Table 1 shows the definitions we defined for the relevant sets employed in the modeling process. The Module set contains all modules of the NSQ messaging system. The ID set consists of unique identifiers for each object within the system. Commands describe the instructions managing interactions within the NSQ, such as message publication (PUB) and subscription (SUB). The Data set indicates the topological information queried by components, and the Ack set is internal feedback.

Based on the above collections, we give the definition of the Message transferred between components:

$$
\begin{aligned}
MSG_{req} &= \{msg_{req}.A.B.Action.Content \mid \\
& \quad A \in Module, B \in Module, \\
& \quad Action \in Command, Content \in ID\} \\
MSG_{rep} &= \{msg_{rep}.A.B.Action.Content \mid \\
& \quad A \in Module, B \in Module, \\
& \quad Action \in Command, \\
& \quad Content \in \{ID, ACK, DATA\}\} \\
MSG_{data} &= \{msg_{data}.Content \mid Content \in ID\}
\end{aligned}
$$

MSG_{req} denotes the set of request messages, MSG_{rep} means the set of responses, and MSG_{data} represents the set of transmitted data.

Next, we define the Channels responsible for communication between the modules and refer to these Channels with the label COM_PATH.

- **ComCL**: channels between consumer and nsqlookupd.
- **ComPD**: channels between producer and nsqd.
- **ComDL**: channels between nsqd and nsqlookupd.
- **ComCD**: channels between consumer and nsqd.

We also define the channels used internally by components with the label MSG_PATH. These channels have cache space and are responsible for caching messages. Fig. 2 shows all the channels we have defined.

- **MsgTPC**: message cache channels of Topics.
- **MsgCHA**: message cache channels of Channels.
- **MsgCON**: message cache channels of consumers.

3.2. Overall Modeling

The NSQ system embodies an intricate workflow. Due to the page limit, we only present part of the core modeling codes in this section.

The whole $System()$ as below:

$$
System() =_{df}
$$
$$
|||_{pid \in PID, did \in DID, lid \in LID, cid \in CID}
$$
$$
\begin{pmatrix}
Producer_{pid}\,[|COM\_PATH|]\,nsqd_{did} \\
[|COM\_PATH|]\,nsqlookupd_{lid} \\
[|COM\_PATH|]\,Consumer_{cid}
\end{pmatrix}
$$

It describes the concurrent model where producers, nsqds, nsqlookupds, and consumers run in parallel and collaborate over the $[|COM\_PATH|]$ channels. The pid denotes the producer ID, and PID means the set of pid. Other characters such as did, lid are similar.

3.3. Producer

The producer is responsible for generating and sending messages to corresponding Topics. It communicates with the nsqd directly and publishes messages to the nsqd module through the $ComPD_i$ channel.

$$Producer_{pid}() =_{df}$$

$$
\left(
\begin{array}{l}
ComPD_i!msg_{req}.pid.did.PUB.tid.MSGID \rightarrow \\
ComPD_i?msg_{rep}.did.pid.REP.msgid.OK \rightarrow \\
updateMsgStates\{PmsgStates[msgid] == 1;\} ; \\
\left(
\begin{array}{l}
SKIP \lhd msgid == 1 \rhd \\
nextMsg\{MSGID++;\} \rightarrow Producer_{pid}()
\end{array}
\right)
\end{array}
\right)
$$
$$\lhd pid == 0 \rhd$$
$$
\left(
\begin{array}{l}
Producer_{pid}() \\
\lhd PmsgStates[0] == 1 \&\& PmsgStates[1] == 1 \rhd \\
\left(
\begin{array}{l}
ComPD_i!msg_{req}.pid.did.PUB.tid.2 \rightarrow \\
CoPD_i?msg_{rep}.did.pid.REP.2.OK \rightarrow \\
updateMsgStates\{PmsgStates[2] == 1;\} \\
\rightarrow SKIP
\end{array}
\right)
\end{array}
\right)
$$

We define two type $Producer_{id}$. $Producer_0$ sends messages with message id 0, 1 while $Producer_1$ with id 2. The producers publish three messages to simulate the practical operation of the NSQ. Furthermore, we restrict that $Producer_1$ must wait for $Producer_0$ to finish sending before it sends the message.

3.4. nsqd

The nsqd is daemon that receives, queues, and delivers messages to clients. It handles multiple streams of data at once through the unique design of Topic and Channel. We modeled three core functions of nsqd.

The entire nsqd process execute as flowing:

$$Nsqd_{did}() =_{df}$$
$$ExecLoop_{did}() |||msgPump_{did}()|||msgPush_{did}();$$

The $ExecLoop_{did}()$ is the main execution loop that drives the core functions of the NSQ daemon. It is responsible for constantly listens requests from other components and processes them according to predefined logic. We model four basic command handling logics, including REQ, SUB, PUB and FIN.

Multicasting and delivery of messages is a core function of nsqd. The relationship between Topics and Channels is established through multicast, ensuring that each Channel receives a copy of all messages associated with a given Topic. This logic is implemented by the $msgPump_{did}()$ process.

$msgPush_{did}()$ is responsible for pushing messages to clients by load balancing strategy. In the NSQ messaging system, this strategy is achieved by employing a random distribution strategy, wherein messages are randomly dispatched to clients subscribed to the same Channel. $Rdy_{cid}[did]$ signifies the number of messages $consumer_{id}$ can process from a specific $nsqd_{id}$. We use the $pushStates_{did}[cid]$ array to mark whether $nsqd_{did}$ is in the state of pushing messages to $consumer_{cid}$. nsqd only sends messages to clients who can process messages. We model this process using the *General Choice* in CSP.

$$ExecLoop_{did}() =_{df}$$

$$
\left(
\begin{array}{l}
ComPD_i?msg_{req}.pid.did.PUB.tid.msgid \rightarrow \\
\left(
\begin{array}{l}
creatTopic(did, tid) \\
\lhd DTStates_{did}[tid] == 0 \rhd SKIP
\end{array}
\right) ; \\
MsgTPC_j!msg_{data}.tid.msgid \rightarrow \\
ComPD_i!msg_{rep}.did.pid.REP.msgid.OK \rightarrow \\
SKIP
\end{array}
\right)
$$
$$\square$$
$$
\left(
\begin{array}{l}
ComCD_k?msg_{req}.cid.did.FIN.tid.chid.msgid \rightarrow \\
updateMsg\{DMStates_{did,tid,chid}[msgid] = -1\} \rightarrow \\
SKIP
\end{array}
\right)
$$
$$\square$$
$$
\left(
\begin{array}{l}
ComCD_k?msg_{req}.cid.did.REQ.tid.chid.msgid \rightarrow \\
updateMsg\{DMStates_{did,tid,chid}[msgid]++\} \rightarrow \\
MsgCHA_l!msg_{data}.tid.chid.msgid \rightarrow \\
SKIP
\end{array}
\right)
$$
$$\square$$
$$
\left(
\begin{array}{l}
ComCD_k?msg_{req}.cid.did.SUB.tid.chid \rightarrow \\
\left(
\begin{array}{l}
createTopic(did, tid); \\
updateChannel\{ \\
\quad DChStates_{did,tid}[chid] = 1;\} \rightarrow \\
Notify(did, tid, cid)
\end{array}
\right) \\
\lhd DTStates_{did}[tid] == 0 \rhd \\
\left(
\left(
\begin{array}{l}
updateChannel\{ \\
\quad DChStates_{did,tid}[chid] = 1;\} \rightarrow \\
Notify(did, tid, cid)
\end{array}
\right)
\right) \\
\lhd DChStates_{did,tid}[chid] == 0 \rhd SKIP \\
addClient\{TCh2C_{did,tid,chid}[cid] = 1;]\} \rightarrow \\
pumpMsg\{startMsgPump_{did}[tid] = 1;\} \rightarrow \\
ComCD_k!msg_{rep}.did.cid.REP.SUB.OK \rightarrow \\
SKIP
\end{array}
\right)
$$
$$ExecLoop_{did}();$$

$$msgPush_{did}() =_{df}$$

$$
\square_{cid}
\left(
\begin{array}{l}
\left(
\begin{array}{l}
MsgCHA_i?msg_{data}.tid.chid.msgid \\
\{pushStates_{did}[cid] = 1;\} \rightarrow \\
ComCD_j!msg_{rep}.did.cid. \\
\quad MSG.tid.chid.msgid \\
\{pushStates_{did}[cid] = 0;\} \rightarrow \\
SKIP
\end{array}
\right) \\
\lhd
\left(
\begin{array}{l}
TCh2C_{did,tid,chid}[cid] == 1 \\
\& \ Rdy_{cid}[did] > 0
\end{array}
\right) \rhd \\
SKIP
\end{array}
\right) ;
$$

$$msgPush_{did}();$$

3.5. nsqlookupd

The nsqlookupd daemon manages the system's topology information. nsqlookupd provides discovery and registration services, which decouple consumers from producers. The formal modeling of nsqlookupd is as follows.

$$nsqlookupd_{lid}() =_{df}$$
$$Lookup_{lid}() \ ||| \ Register_{lid}() \ ||| \ ErrorHandler_{lid}();$$

The $Register_{lid}()$ process handles the registration requests sent by nsqd through the $ComCD_i$ channel, and record nsqd instance by $LDStates_{lid}[did]$. $LTStates_{lid}[tid]$ stores all the registered Topics on the nsqlookupd, and $T2D_{lid}[tid][did]$ holds the corresponding nsqd addresses for each Topic. Similarly, $LChStates_{lid,tid}[chid]$ and $TC2D_{lid}[tid][chid][did]$ serve same functions for Channels.

$$Register_{lid}() =_{df}$$
$$ComDL_i?msg_{req}.did.lid.REGISTER.tid.chid \rightarrow$$
$$addnsqd\{LDStates_{lid}[did] = 1;\} \rightarrow$$

$$\left(\begin{array}{l} \left(\begin{array}{l} registerTopic\{ \\ \quad LTStates_{lid}[tid] = 1; \\ \quad T2D_{lid}[tid][did] = 1;\} \rightarrow SKIP \end{array} \right) \\ \lhd chid == -1 \rhd \\ \left(\begin{array}{l} registerTopicAndChan\{ \\ \quad LTStates_{lid}[did] = 1; \\ \quad LChStates_{lid,tid}[chid] = 1; \\ \quad T2D_{lid}[tid][did] = 1; \\ \quad TC2D_{lid}[tid][chid][did] = 1;\} \rightarrow \end{array} \right) \\ SKIP \end{array} \right) ;$$

$$Register_{lid}();$$

$Lookup_{lid}()$ formalizes nsqlookupd's responses to queries from consumers and nsqd instances using General Choice. $lookpnsqd(lid, tid)$ provides all the stored nsqd address information associated with a specific Topic in nsqlookupd. Similarly, the $lookupChannel(lid, tid)$ returns Channels list under the specified Topic.

$$Lookup_{lid}() =_{df}$$

$$\left(\begin{array}{l} ComCL_i?msg_{req}.cid.lid.LOOKUPD.tid \rightarrow \\ lookupnsqd(lid, tid); \\ ComCL_i!msg_{rep}.lid.cid.REP.tid.dlist \rightarrow \\ SKIP \end{array} \right)$$

$$\square$$

$$\left(\begin{array}{l} ComDL_i?msg_{req}.did.lid.LOOKUPCHA.tid \rightarrow \\ lookupChannel(lid, tid); \\ ComDLi!msg_{rep}.lid.did.REP.tid.chlist \rightarrow \\ SKIP \end{array} \right) ;$$

$$Lookup_{lid}();$$

We also modeled the response of nsqlookupd to connection errors. When nsqlookupd encounters connection timeouts with nsqd, it will receive $OUTTIME$

signal through $ComDL_i$ and then remove all information associated with the corresponding nsqd from its records. This process ensures that the information stored on nsqlookupd remains consistently available.

3.6. Consumer

When a consumer is initiated, it queries nsqlookupd for the addresses of nsqd instances associated with the target Topics. Upon receiving the addresses, it subscribes to all of these instances. Only after these can the consumer activate processes for message retrieval and processing.

Therefore, the modeling of consumer is as follows:

$$Consumer_{cid,tid,chid}() =_{df}$$
$$ConnToLookupds_{cid,tid}();$$
$$(Handler_{cid}() \ ||| \ ReadLoop_{cid}());$$

$$ConnToLookupds_{cid,tid}() =_{df}$$
$$LOOP(lid : 0..LD) :$$
$$\quad SKIP \lhd addrLookupd[lid] == 0 \rhd$$

$$\left(\begin{array}{l} addLD\{CLStates_{cid}[lid] = 1;\} \rightarrow \\ count\{totalLD = countLD(lid, cid);\}; \\ \left(\begin{array}{l} \left(\begin{array}{l} ComCL_i!msg_{req}. \\ \quad cid.lid.LOOKUPD.tid \rightarrow \\ ComCL_i?msg_{rep}. \\ \quad lid.cid.REP.tid.dlist \rightarrow \\ LOOP(did : 0..D) : \\ \left(\begin{array}{l} ConnToNsqd_{cid,did,tid}() \\ \lhd dlist[did] == 1 \rhd SKIP \end{array} \right) ; \end{array} \right) \\ \lhd totalLD == 1 \rhd SKIP \end{array} \right) \end{array} \right) ;$$

$$ConnToNsqd_{cid,did,tid}() =_{df}$$
$$SKIP \lhd CDStates_{cid}[did] == 1 \rhd$$

$$\left(\begin{array}{l} ComCD_i!msg_{req}.cid.did.SUB.tid.c2ch[cid] \rightarrow \\ ComCD_i?msg_{rep}.did.cid.REP.SUB.OK \rightarrow \\ addnsqd\{CDStates_{cid}[did] = 1;\} \rightarrow \\ updateRDY\{Rdy_{cid}[did] = 1\} \rightarrow SKIP \end{array} \right) ;$$

The above formula models the process of a consumer connecting to nsqdlookupds and nsqds. The consumer sends a SUB request to the nsqd through $ComCD_i$ channel. It records connection information in $CDStates_{cid}[did]$ and updates the Rdy_{cid} value of $nsqd_{id}$. In the formula, the value of Rdy_{cid} is set to 1, indicating the consumer's readiness to process one message from $nsqd_i$.

$$readLoop_{cid}() =_{df}$$
$$ComCD_i?msg_{rep}.did.cid.$$
$$\quad MSG.tid.chid.msgid\{Rdy_{cid}[did] --;\} \rightarrow$$

$$\left(\begin{array}{l} \left(\begin{array}{l} updateRDY\{Rdy_{cid}[did] ++;\} \rightarrow \\ ReadLoop_{cid,did,tid}() \end{array} \right) \\ \lhd Attempts_{cid}[msgid] == -1 \rhd \\ \left(\begin{array}{l} MsgCON_j!msg_{data}.did.tid.chid.msgid\{ \\ \quad msg2d_{cid}[msgid] = did;\} \rightarrow \\ ReadLoop_{cid,did,tid}() \end{array} \right) \end{array} \right) ;$$

After completing the subscription, the consumer maintains a TCP connection with nsqd to be ready to receive messages. The diminishing of $Rdy_{cid}[did]$ value implies a decrease in the amount of messages consumers can handle. $Attempts_{cid}[msgid]$ keeps track of the message attempts number. -1 signifies successful processing and will release $Rdy_{cid}[did]$. Otherwise, the message is cached in the $MsgCON_i$ channel for further processing.

$Handler_{cid}()$ is the message-handling module of the consumer process. In our experiment, we use nondeterministic to model the message-processing behavior. We also model aborting re-queuing when the message attempts exceed the maximum value. $MaxAttempts$ defines the maximum number of message attempts allowed by the system.

$$Handler_{cid}() =_{df}$$

$$MsgCON_i?msg_{data}.did.tid.chid.msgid \rightarrow$$

$$\left(\begin{array}{c} ComCD_j!msg_{req}.cid.did.FIN.tid.chid.msgid\{ \\ Attempts_{cid}[msgid] = -1; \} \rightarrow \\ updateRDY\{Rdy_{cid}[did] + +; \} \rightarrow SKIP \end{array} \right)$$

$$\lhd Attempts_{cid}[msgid] > MaxAttempts \rhd$$

$$\left(\begin{array}{c} \left(\begin{array}{c} ComCD_j!msg_{req}. \\ cid.did.FIN.tid.chid.msgid\{ \\ Attempts_{cid}[msgid] = -1; \} \rightarrow \\ updateRDY\{Rdy_{cid}[did] + +; \} \rightarrow SKIP \end{array} \right) \\ \sqcap \\ \left(\begin{array}{c} ComCD_j!msg_{req}. \\ cid.did.REQ.tid.chid.msgid\{ \\ Attempts_{cid}[msgid] + +; \} \rightarrow \\ updateRDY\{Rdy_{cid}[did] + +; \} \rightarrow SKIP \end{array} \right) \end{array} \right)$$

$$Handler_{cid}();$$

4. Verification

In this section, we use the model-checking tool PAT to realize the formal model constructed in section III, and verify its properties. At the same time, the results of properties verification are also shown at the end.

4.1. Implementation

This part presents details of the modeling implementation with the PAT tool, mainly concerning the definition of constants, array variables and channels.

$$\#define\ P\ 2; \qquad \#define\ C\ 2;$$
$$\#define\ D\ 2; \qquad \#define\ LD\ 2;$$
$$\#define\ T\ 1; \qquad \#define\ CHA\ 2;$$
$$\#define\ MsgNum\ 3;$$

We define constants as above to materialize the architecture of the NSQ system in Fig. 1. P, D, LD, and C represent the number of producer, nsqd, nsqlookupd, and consumer. T and CHA denote that each nsqd has one

Topic and associated two Channels. $MsgNum$ defines the number of messages.

$$var\ Rdy[C][D] = [0,0,0,0];$$
$$var\ pushStates[D][C] = [0,0,0,0];$$
$$var\ LDStates[LD][D] = [0,0,0,0];$$
$$var\ Attempts[C][MsgNum] = [0,0,0,0,0,0];$$

In addition, We define some arrays to store system information, which assists us in confirming the status of processes. $Rdy[C][D]$ is used to record the number of messages the consumer can process. $pushStates[D][C]$ marks whether the nsqd is in the state of pushing messages to the consumer. $LDStates[LD][D]$ logs information about registered instances of nsqd on nsqlookupd. $Attempts[C][MsgNum]$ tracks the status of messages processed on the consumer.

Furthermore, we have implemented the relevant channels in PAT based on the definitions provided earlier. We use multidimensional arrays to store channels between different entities is to avoid resource contention.

$$channel\ ComPD[P][D]\ 0;$$
$$channel\ ComCD[C][D]\ 0;$$
$$channel\ MsgTPC[D][T]\ MsgNum;$$
$$channel\ MsgCHA[D][T][CHA]\ MsgNum;$$

The channel definitions can be categorized into two types: COM_PATH are used for inter-component communication, where the channel size is set to 0 to achieve process synchronization. Cache channels MSG_PATH are used within components, where the channel size is set to $MsgNum$. These channels are utilized for process synchronization and message buffering.

Given that the NSQ message system operates with multiple producers, nsqds, nsqlookupds, and consumers, we employ a combination of interleaving and loop functions to realize the system's implementation. The comprehensive definition of the NSQ system is presented as follows. $|||i : \{0..N\}@P(i);$ statement means that multiple processes $P(i)$ run interspersed in the PAT.

$$System() =$$
$$|||pid : \{0..P-1\}; did : \{0..D-1\}; ldid : \{0..LD-1\};$$
$$cid : \{0..C-1\}; tid : \{0..T-1\}; chid : \{0..CHA-1\}$$
$$@ \left(\begin{array}{c} Producer(pid, tid) \parallel nsqd(did) \parallel \\ nsqlookupd(lid) \parallel Consumer(cid, tid, chid) \end{array} \right)$$

4.2. Properties Verification

In this section, we verify the properties of the constructed model with the model checker PAT. These properties present the flexibility and robustness of NSQ distributed messaging platform.

4.2.1. Divergence Freedom

In NSQ system, if messages can always flow and be handled in the correct way as they should, avoiding invalid or infinite loops, then we think the system is divergence free. It is crucial for message systems because the correctness and stability of the system depends on the correct handling and delivery of messages.

PAT provides the primitive to verify the divergence freedom of the system:

$$\#assert \ System() \ divergence free;$$

4.2.2. Reachability

Data reachability is the basic property of message queue. NSQ ensures at least one delivery of a message using the FIN and REQ, but it does not guarantee data order. In our experiment, we track the attempts of messages with $Attempts[C][MsgNum]$, where the value of -1 indicates the message is finished. Therefor, the definitions of reachability and assertions are as follows:

$\#define \ Reachability\{$
$\quad Attempts[0][0] == -1 \ \&\& \ Attempts[1][0] == -1$
$\&\& \ Attempts[0][1] == -1 \ \&\& \ Attempts[1][1] == -1$
$\&\& \ Attempts[0][2] == -1 \ \&\& \ Attempts[1][2] == -1\};$
$\#assert \ System() \ \models<> \ Reachability;$

As the model we constructed has two consumers subscribing to different Channels under the same Topic, each consumer will receive a copy of all messages sent by producers and finish them all eventually. Symbol $<>$ means that the system can finally reach $Reachability$ state.

4.2.3. Scalability

The NSQ system realizes a distributed decentralized architecture with nsqlookupd, which shows scalability. nsqlookupd manages the topological information of the system and allows nsqd instances to be added for horizontal scaling. In our experiments, the $LDStates[LD][D]$ is initially set to 0, denoting that no nsqd instances are available. When the value changes to 1, it indicates that nsqd instances were dynamically added, demonstrating the system's scalability.

$\#define \ Scalability\{$
$\quad LDStates[0][0] == 1 \ \&\& \ LDStates[0][1] == 1$
$\&\& \ LDStates[1][0] == 1 \ \&\& \ LDStates[1][1] == 1\};$
$\#assert \ System() \ \models<> \ Scalability;$

4.2.4. Availability

nsqlookupd serves as a distributed directory service that supports fault tolerance and redundancy. It maintains information about the available components of the system forever, and when instances of nsqd are abnormal, it deletes all information about the corresponding instances. We defined a new system to verify the high availability of NSQ. An $OUTTIME$ event of $nsqd_0$ is added to the original system, which will be triggered when all messages are finished.

$$System2() = System()|||$$
$$\left(\begin{array}{l} |||lid : \{0..LD - 1\} \\ @ \left(\begin{array}{l} [reachability] \\ ComDL_i!msg_{req}.0.lid. \\ ERROR.OUTTIME \to SKIP; \end{array} \right) \end{array} \right)$$

The above formula describes the new system, and we verify in the PAT whether nsqlookupd maintains the list of available nsqd. The definition and assertion are as follows:

$\#define \ Availability\{$
$\quad LDStates[0][0] == 0 \ \&\& \ LDStates[0][1] == 1$
$\&\& \ LDStates[1][0] == 0 \ \&\& \ LDStates[1][1] == 1\};$
$\#assert \ System2() \ \models<> \ Availability;$

4.2.5. Flow Controllability

NSQ can dynamically adjust messages' processing rate by changing the consumer's RDY value. To verify this property, we need to demonstrate that nsqd can push messages only if the consumer's RDY is greater than 0. Therefore, we introduce the $pushStates[D][C]$ array to store nsqds' status, which indicate whether $nsqd_{did}$ is pushing data to $Consumer_{cid}$. Combined with the $Rdy[C][D]$ array, we give the following definition and assertion.

$\quad \#define \ Ready00 \ \{Rdy[0][0] > 0\};$
$\quad \#define \ Ready \cdots$
$\quad \#define \ pushStop00 \ \{pushStates[0][0] = 0\};$
$\quad \#define \ pushStop \cdots$
$\quad \#assert \ System() \ \models$
$\qquad (pushStop00 \ \mathbf{U} \ Ready00)$
$\quad \&\&(pushStop01 \ \mathbf{U} \ Ready01)$
$\quad \&\&(pushStop10 \ \mathbf{U} \ Ready10)$
$\quad \&\&(pushStop11 \ \mathbf{U} \ Ready11);$

Our model has four message subscription connections as show in Fig.2. $pushStop00$ defines the state when $nsqd_0$ stops pushing messages to the $consumer_0$, and $Ready00$ defines the state in which the $consumer_0$ is ready to receive messages from $nsqd_0$. The rest of definitions are similar. We use the Untill($\mathbf{U}$) syntax from Linear Timing Logic (LTL) to describe the event that *the nsqd stops pushing messages until the Rdy of the corresponding consumer is larger than zero*. This formula verifies if the system can realize flow control.

	Divergence Freedom	Reachability	Scalability	Availability	Flow Controllability
Result	VALID	VALID	VALID	VALID	VALID
Visited States	324159	730077	5143	730225	42775
Total Transitions	1153000	2634052	10565	2634218	120917
Time Used	361. 275s	116.7222s	0.5011s	127.0521s	5.3045s
Estimated Memory Used	81164.15KB	59856.97KB	12643.18KB	58307.32KB	18067.96KB

Figure 3: Verification Results of the NSQ System

4.3. Verification and Results

Depending on the definitions and assertions provided above, we use model checker PAT to verify five properties of the NSQ system, including *Divergence Freedom*, *Reachability*, *Scalability*, *Availability*, and *Flow Controllability*. The model checker PAT verifies properties by searching for counterexamples in the system's state space or reaching the limits of state exploration.

We present a summary of the verification statistics in Fig. 3, including *Visited States*, *Total Transitions*, *Time Used*, and *Estimated Memory Used*.

The verification results of all five properties indicate that the NSQ message queue satisfies all the above properties, proving that the system has high flexibility and robustness while providing credible delivery of messages.

5. Conclusion and Future Work

In this paper, we focus on the core functionalities of the NSQ message platform, including message publishing, registration, subscription, and querying. With CSP, we formalized critical components of the NSQ architecture, such as producers, consumers, nsqd, and nsqlookupd. Using the model checker PAT, we conducted a rigorous analysis of the constructed NSQ model, verifying five fundamental properties: *Divergence Freedom*, *Reachability*, *Scalability*, *Availability*, and *Flow Controllability*. These properties underscore NSQ's capacity to handle real-time distributed message delivery at scale, confirming its high flexibility and robustness while ensuring dependable message transmission.

Nonetheless, besides the robustness of message queues, the security of data is extremely important for users. In the future, we will continue to enhance the formalized modeling and verification of NSQ by refining workflows. We will also delving into the system's security aspects to advance our research outcomes continually.

References

[1] Bernstein, P. A. (1996). Middleware: a model for distributed system services. Communications of the ACM, 39(2), 86-98.

[2] Snyder, B., Bosnanac, D., & Davies, R. (2011). ActiveMQ in action (Vol. 47). Greenwich Conn.: Manning.

[3] Rostanski, M., Grochla, K., & Seman, A. (2014, September). Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ. In 2014 federated conference on computer science and information systems (pp. 879-884). IEEE.

[4] Wang, G., Koshy, J., Subramanian, S., Paramasivam, K., Zadeh, M., Narkhede, N., ... & Stein, J. (2015). Building a replicated logging system with Apache Kafka. Proceedings of the VLDB Endowment, 8(12), 1654-1655.

[5] Yue, M., Ruiyang, Y., Jianwei, S., & Kaifeng, Y. (2017, October). A MQTT protocol message push server based on RocketMQ. In 2017 10th International Conference on Intelligent Computation Technology and Automation (ICICTA) (pp. 295-298). IEEE.

[6] NSQ: A realtime distributed messaging platform, https://nsq.io/

[7] Lai, X., Wang, H., Zhao, J., Zhang, F., Zhao, C., & Wu, G. (2020, May). HBase Connection Dynamic Keeping Method Based on Reactor Pattern. In Journal of Physics: Conference Series (Vol. 1544, No. 1, p. 012122). IOP Publishing.

[8] Raje, S. N. (2019). Performance Comparison of Message Queue Methods (Doctoral dissertation, University of Nevada, Las Vegas).

[9] Togashi, N., & Klyuev, V. (2014, April). Concurrency in Go and Java: performance analysis. In 2014 4th IEEE international conference on information science and technology (pp. 213-216). IEEE.

[10] Brookes, S. D., Hoare, C. A. R, & Roscoe, A. W. (1984). A theory of communicating sequential processes. Journal of the ACM (JACM), 31(3), 560-599.

[11] Hoare, C. A. R. (1985). Communicating sequential processes (Vol. 178). Englewood Cliffs: Prentice-hall.

[12] PAT: Process Analysis Toolkit. An Model Checker and Refinement Checker for Concurrent and Real-time System. https://pat.comp.nus.edu.sg/

[13] Xiao, L., Zhu, H., Xu, Q., & Vinh, P. C. (2022). Modeling and verifying pso memory model using CSP. Mobile Networks and Applications, 27(5), 2068-2083.

[14] Xu, J., Yin, J., Zhu, H., & Xiao, L. (2023). Formalization and verification of Kafka messaging mechanism using CSP. Computer Science and Information Systems, 20(1), 277-306.