# Verification of RabbitMQ with Kerberos Using Timed Automata

**Article** *in* Mobile Networks and Applications · May 2022

DOI: 10.1007/s11036-022-01986-8

Huibiao Zhu
East China Normal University
**262** PUBLICATIONS **1,798** CITATIONS
SEE PROFILE

Phan Cong Vinh
Nguyen Tat Thanh University
**151** PUBLICATIONS **463** CITATIONS
SEE PROFILE

Mobile Networks and Applications (2022) 27:2049–2067
https://doi.org/10.1007/s11036-022-01986-8

# Verification of RabbitMQ with Kerberos Using Timed Automata

Ran Li[1] · Jiaqi Yin[1] · Huibiao Zhu[1] · Phan Cong Vinh[2]

Accepted: 4 April 2022 / Published online: 9 May 2022

**Abstract**
RabbitMQ, an implementation of Advanced Message Queuing Protocol (AMQP), is a very popular message middleware. It supports concurrency, guarantees sequential consistency, and enables independent applications and services to communicate. Consequently, it is of great significance to ensure the secure communication of RabbitMQ. Therefore, Kerberos, a network authentication protocol, is introduced to combine with RabbitMQ to address this security issue. In this paper, we apply formal methods to model and verify RabbitMQ with Kerberos. By utilizing UPPAAL, RabbitMQ is abstracted to timed automata. Further, we validate the constructed model with the simulator in UPPAAL. On this basis, we verify whether RabbitMQ meets some basic but essential properties, including *Reachability of Data*, *Concurrency*, *Sequence Consistency* and *Heartbeat Mechanism*. Additionally, the security property *Secure Communication* is verified as well. From the verification results via UPPAAL, it can be found that RabbitMQ can totally cater for these properties and it maintains secure communication under the umbrella of Kerberos.

**Keywords** Modeling · Verification · UPPAAL · RabbitMQ · AMQP · Kerberos

## 1 Introduction

With the rapid growth of information, powerful and stable communication between entities carries great meaning. The Message Queue (MQ) [1–3] is a component of efficient and reliable message middleware solutions that transfers messages between applications and services. With a message queue model, communication between processes can be extended in a distributed environment.

Currently, many popular open-source and commercially-supported message queue middlewares have seen wide adoption in enterprise companies, including RabbitMQ [4], Kafka [5], ActiveMQ [6], RocketMQ [7], etc. Among them, RabbitMQ is a message queue middleware that uses the Erlang language to implement Advanced Message Queuing Protocol (AMQP) [8, 9]. Also, it supports other protocols such as Message Queuing Telemetry Transport (MQTT) [10], Simple Text Orientated Messaging Protocol (STOMP) [11] and Extensible Messaging and Presence Protocol (XMPP) [12]. In this paper, we only focus on RabbitMQ based on the AMQP specification.

Due to its usability, expansibility, and high availability, RabbitMQ is growing rapidly. It has been widely applied in defense, telecommunications, manufacturing, Internet and cloud computing, and many additional market segments [4]. There are more than 500 commercial users, including JPMorgan, National Science Foundation, NASA, Red Hat, IBM, Google, AT&T, VMware, Mozilla, INETCO, etc [4]. The wide application reminds us that it is vital to ensure the security of communication in RabbitMQ.

Hence, we need to introduce an authentication protocol to RabbitMQ. In this paper, we combine Kerberos [13] with RabbitMQ. Kerberos, proposed by Massachusetts Institute of Technology (MIT), is a network authentication protocol designed to provide powerful authentication services for Client/Server applications based on timestamp and encrypted tickets [13].

This paper extends our work published at TrustCom 2020 [14]. In our previous work, we mainly focused on the reading and writing processes of RabbitMQ, and employed

✉ Jiaqi Yin
jqyin@stu.ecnu.edu.cn

✉ Huibiao Zhu
hbzhu@sei.ecnu.edu.cn

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

[2] Nguyen Tat Thanh University, 300A Nguyen Tat Thanh Street, Ward 13, District 4, Ho Chi Minh City, Vietnam

UPPAAL [15, 16] to formalize its architecture. We verified four important properties, including *Reachability of Data*, *Concurrency*, *Sequence Consistency* and *Message Acknowledgement*. Now, compared with our previous work [14], we omit some detailed processes of production and consumption for simplification, and combine the Kerberos authentication [13] with RabbitMQ. In addition, considering the network failure, it may take a long time to detect disrupted connections. Therefore, RabbitMQ provides a heartbeat feature to help the application layer observe the failure quickly [4]. In our model, we formalize the heartbeat mechanism between the client and the broker of RabbitMQ as well.

More specifically, in this paper, we abstract the entities of RabbitMQ to six types of timed automata (i.e., *AS*, *TGS*, *Producer*, *Consumer*, *Broker* and *Queue*) with the aid of UPPAAL. Then, we validate the correctness of our constructed model by simulation. The message sequence chart returned from UPPAAL agrees in the specification of RabbitMQ and Kerberos, which means the model is validated. Meanwhile, we verify the original properties and add some other properties for the sake of secure communication. As in our previous work [14], we verify *Reachability of Data*, *Concurrency*, *Sequence Consistency* in this paper. Since we add the heartbeat mechanism to this model, we check this property as well. Besides, instead of *Message Acknowledgement*, we verify *Secure Communication* in consideration of the existence of invalid clients. In the light of the verification results, we conclude that RabbitMQ satisfies these essential properties and it supports secure communication under the protection of Kerberos.

The remainder of this paper is organized as follows. In Section 2, we give an introduction to RabbitMQ and Kerberos. Additionally, we introduce UPPAAL briefly in this section. Section 3 is devoted to formalizing the model of RabbitMQ. Furthermore, Section 4 is about the validation and verification results of the model. Related work is discussed in Section 5. Finally, we conclude our work and propose some future work in Section 6.

## 2 Background

In this section, we briefly explain the architecture of RabbitMQ and the procedures of Kerberos authentication. Moreover, we also provide a short introduction to UPPAAL.

### 2.1 RabbitMQ

As a whole, RabbitMQ is a producer-consumer model, primarily responsible for receiving, storing and forwarding messages [17]. As shown in Fig. 1, it consists of three main components, i.e., Producer, Consumer and Broker.

- **Producer.** The producer creates the message and then publishes it to the broker in RabbitMQ.
- **Consumer.** The consumer connects to the broker and subscribes to the queue, then the consumer can consume messages.
- **Broker.** The broker is the service node of the messaging middleware. It routes messages to queues through the exchange. As presented in Fig. 1, there are three main types of exchanges [4].
  - **Direct.** This type of exchange sends messages to the queue according to the routing key.
  - **Topic.** This exchange transfers messages according to the topics.
  - **Fanout.** This kind of exchange transmits messages to all the queues that are bound with this exchange.

  In our paper, since how the exchange routes messages is not our concern, we assume that all the exchanges are direct. Further, we omit the details of this procedure in our formalized model in the next section.

### 2.2 Kerberos

Kerberos contains three main parts, including Client, Server and Key Distribution Center (KDC) [13].

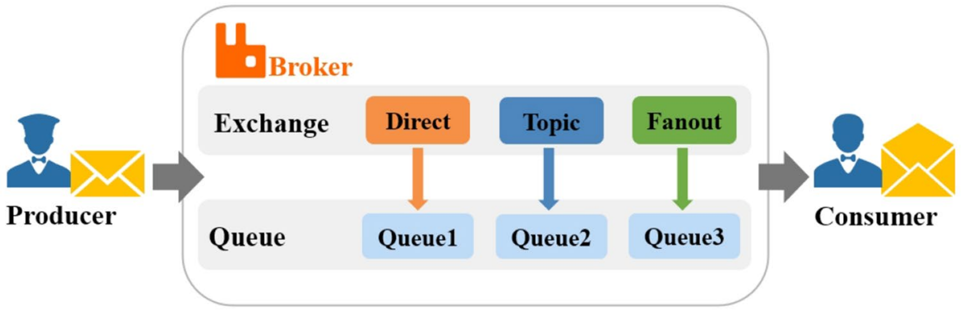


**Fig. 1** The architecture of RabbitMQ

– **Client and Server.** In the architecture of RabbitMQ, either the producer or the consumer needs to be authenticated through Kerberos before communicating with the broker. That is, the producer and the consumer act as the clients, and the broker is the server in Kerberos.
– **KDC.** It is the core component of Kerberos. KDC includes Authentication Service (AS) and Ticket Granting Service (TGS). AS is used to provide Ticket Granting Tickets (TGT), and TGS is responsible for providing service Tickets for the client.

The authentication flow of Kerberos is depicted in Fig. 2, and Table 1 lists the information of messages delivered during the authentication flow. Before accessing the services of the server, the client interacts with AS, TGS and the server in turn.

- **Client and AS.**
  – The client sends a request to AS, this request contains the client's ID and the target server's ID.
  – AS checks if the client's ID is in KDC's database. If exists, AS returns Client/TGS Session Key (CTSessionKey) encrypted with the client's secret key (CKey) and TGT encrypted with TGS's secret key (TKey) to the client. Otherwise, AS rejects this request.
- **Client and TGS.**

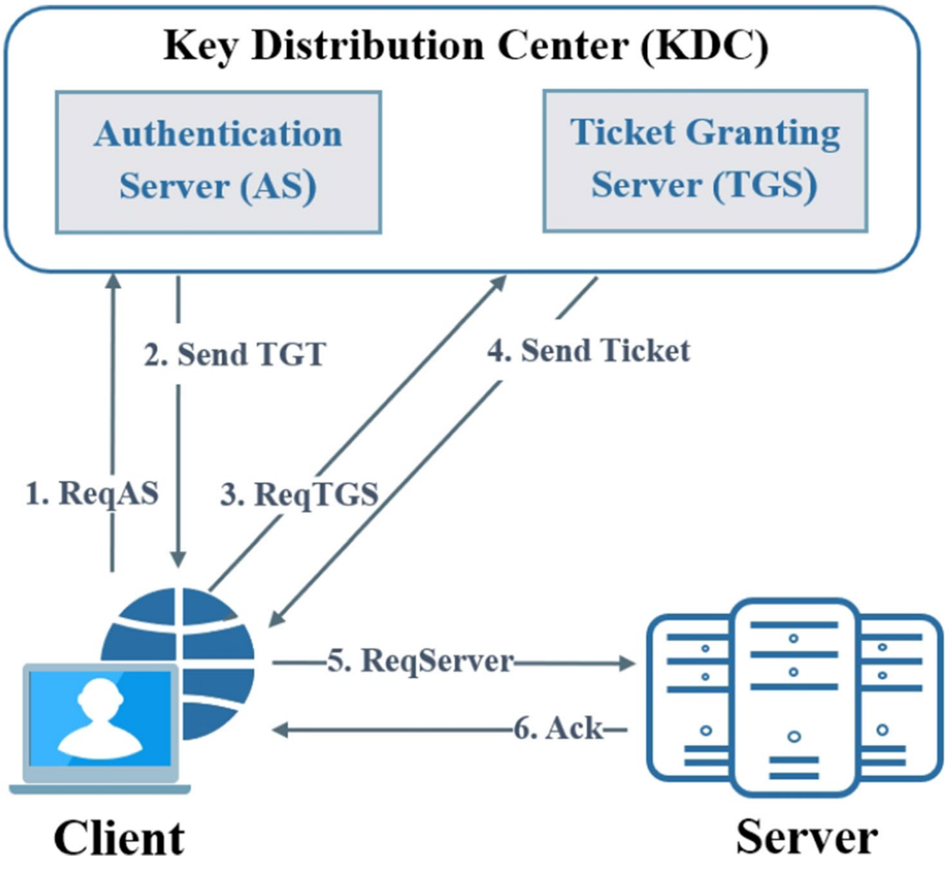

Key Distribution Center (KDC)
Authentication Server (AS)
Ticket Granting Server (TGS)
2. Send TGT
4. Send Ticket
1. ReqAS
3. ReqTGS
5. ReqServer
6. Ack
Client
Server
**Fig. 2** Authentication flows in Kerberos

**Table 1** Messages delivered in Kerberos

| Notation | Sender | Receiver | Key | Content |
|---|---|---|---|---|
| MsgA | AS | Client | CKey | CTSessionKey |
| MsgB | AS/Client | Client/TGS | TKey | **TGT** (i.e., CTSessionKey, ClientID, ServerID, Lifetime) |
| MsgD | Client | TGS | CTSessionKey | **Authenticator** (i.e., ClientID, Time) |
| MsgE | TGS/Client | Client/Server | SKey | **Ticket** (i.e., CSSessionKey, ClientID, Lifetime) |
| MsgF | TGS | Client | CTSessionKey | CSSessionKey |
| MsgG | Client | Server | CSSessionKey | **Authenticator** (i.e., ClientID, Time) |

- The client uses its secret key to decrypt and gain CTSessionKey. Then, the client requests TGS for authentication. The client sends the encrypted TGT received from AS along with an Authenticator to TGS.
- KDS decrypts TGT with its own secret key and gets CTSessionKey. Next, it utilizes this key to decrypt the Authenticator. After authenticating TGT and the Authenticator, KDS returns client-to-server Ticket encrypted with the target server's secret key (SKey) and Client/Server Session Key (CSSessionKey) encrypted with the CTSessionKey to the client.

• **Client and Server.**

  - The client decrypts and gains the CTSessionKey. Afterwards, it sends the encrypted Ticket along with a new Authenticator to the target server.
  - The server decrypts and gains the CTSessionKey. Then, it decrypts and authenticates the received Authenticator. After successful authentication, the server can provide the corresponding services to the client.

## 2.3 UPPAAL

UPPAAL is an integrated tool environment used to model, simulate and verify real-time systems [15, 16]. It is composed of three major parts, i.e., description language, simulator and model-checker.

### 2.3.1 Description language

A system can be formalized as a network of several timed automata in parallel in UPPAAL. We define templates automata with a set of parameters. Templates automata mainly include locations and edges [16].

- **Location.** A location represents the state of the template and there are three kinds of locations as below.
  - **Initial Location.** Each template must have one and only one initial location, which denotes the initial state of the template.
  - **Urgent Location.** It freezes time and it is marked by *U* in a circle.
  - **Committed Location.** It is marked by *C* in a circle and it cannot delay. Moreover, the next transition must involve an outgoing edge of at least one of the committed locations [15, 16].
- **Edge.** A directed edge connects locations and it means a transition from the current location to the next location. The edges are associated with the following four components.
  - **Selections.** An edge associated with a selected label contains variables and these variables take values randomly in the range of their respective types.
  - **Guards.** A guard can return a Boolean value, and the edge will be enabled when the guard returns *true*.
  - **Synchronization.** The synchronization in UPPAAL is achieved through channels. An edge labeled with $c!$ synchronizes with another labeled $c?$.
  - **Updates.** When the edge is enabled, the expressions of the update label will be executed.

### 2.3.2 Simulator

At the early design or modeling stage, the simulator checks the possible dynamic performance of the system. Therefore, it can provide an inexpensive approach to fault detection before verification [15, 16]. Through simulation, UPPAAL returns the corresponding Message Sequence Chart. We can use it to validate the correctness of our model.

### 2.3.3 Model-Checker

UPPAAL can check reachability properties, safety properties and liveness properties of the system model and it characterizes these properties with a simplified version of TCTL (Timed Computational Tree Logic) [15, 16].

- **Reachability Property.** $E\langle\rangle\ p$ evaluates to true if $p$ can be satisfied by any reachable location.
- **Safety Property.** $A[]\ p$ is true means that $p$ should be true in all reachable locations.
- **Liveness Property.** $A\langle\rangle\ p$ is true, indicating that $p$ is eventually satisfied.

### 2.3.4 Case study

To get a better understanding of UPPAAL, we employ a build-in demo called *Vikings Cross Bridge* [15, 16] to illustrate the approach of formalization and verification. This example is adapted from [15].

In this example, there are four Vikings trying to cross a damaged bridge. For clarity, we assume that they intend to cross from *A* side to *B* side. The bridge can bear only two Vikings at the same time. Besides, the Vikings need to carry a torch when they are crossing the bridge, and they have only one torch. We assume that the Vikings need 5, 10, 20 and 25 minutes to walk across the bridge.

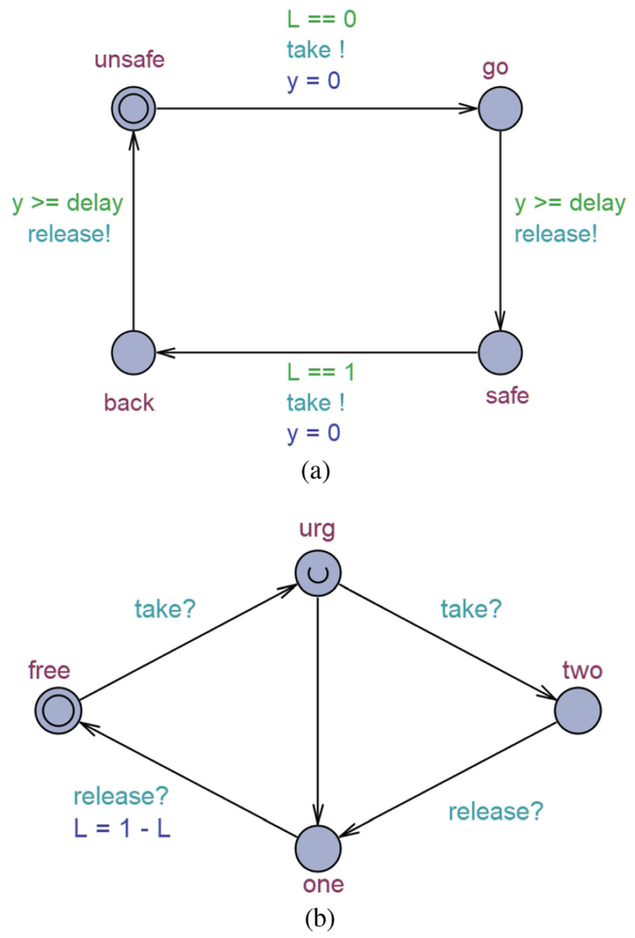


**Fig. 3** Example of *Vikings Cross Bridge* (Adapted from [15])

We abstract the behaviors of Vikings in Fig. 3(a), and its parameter is *delay* (i.e., time required to cross the bridge). *unsafe* is the initial location, representing the Viking has not crossed the bridge (i.e., at $A$). If the Viking gains the torch ($L == 0$, indicating the torch is at $A$) through the channel *take*, the local clock $y$ is set to 0 and the automaton reaches *go* (i.e., the Viking is crossing the bridge). When $y \geqslant delay$, it means that the Viking has reached the other side (i.e., at $B$). The torch can be released through the channel *release* and the automaton runs into *safe* location. Then, one Viking (who is at $B$ now) carries the torch ($L == 1$, meaning the torch is at $B$) and goes back to $A$. The process of coming back to $A$ is similar to the above process of coming to $B$.

Figure 3(b) demonstrates the states of the torch. The initial location *free* informs that the torch is free for use. Once a Viking attempts to take the torch through the channel *take*, *Torch* automaton reaches the urgent location *urg*. There are two conditions from this location. In one case, the Viking crosses the bridge with the torch, and the automaton moves to *one*. In the other case, another Viking also tries to get the torch and the automaton jumps to *two*. In the second case, one of the two Vikings must give up since there is only one torch. Then, the automaton runs into *one*. After reaching the other side, the torch is released and be free again.

After constructing the above two automata, we model this problem as a network of *Soldier* and *Torch* in parallel. The system is declared as below.

$Viking1 = Soldier(5);\ Viking2 = Soldier(10);$
$Viking3 = Soldier(20);\ Viking4 = Soldier(25);$
$system\ Viking1,\ Viking2,\ Viking3,\ Viking4,\ Torch;$

In order to verify whether the Vikings can get safe within 60 minutes, we define this property as below. Here, *time* is a global clock. We feed the formalized system to the model checker UPPAAL, and the verification result shows that this property is satisfied.

$$E <> \ Viking1.safe \ \&\& \ Viking2.safe \ \&\& \ Viking3.safe \ \&\& \ Viking4.safe \ \&\& \ time <= 60$$

## 3 Modeling rabbitMQ with Kerberos

In this section, we give the formalized model of RabbitMQ with Kerberos. We first abstract data structures of it, including channels, variables and functions. On this basis, we formalize the model in UPPAAL.

### 3.1 Abstracting data structures

#### 3.1.1 Channels

First, we define some channels to realize the synchronization between processes. Table 2 shows the channels and their functionalities.

Channels with the same functionality are described as a uniform form. The * character can match the producers' IDs and the consumers' IDs in our model. For example, *clientReqAs** represents the channels beginning with the word *clientReqAs* and is used for sending requests to AS.

#### 3.1.2 Variables

Next, some variables are introduced in our formalized model for describing the system behaviors. Their types and meanings are presented in Table 3.

We use the keyword *typedef* to define the types of clients, producers, consumers and the messages transmitted during the Kerberos authentication.

**Table 2** Channels in RabbitMQ with Kerberos

| Channel | Functionality |
|---|---|
| clientReqAs* | Sending requests to AS |
| clientReqTgs* | Requesting authentication from TGS |
| clientReqServer* | Requesting service from the server |
| asSendMsg* | Sending TGT to the client |
| tgsSendMsg* | Sending client-to-server Ticket to the client |
| serverSendMsg* | Sending an acknowledgement to the client |
| sendError* | Sending an error to the client |
| check* | Checking the heartbeat |
| ack* | Returning the heartbeat |
| open* | Opening the connection |
| dequeue* | Declaring a queue |
| publish* | Publishing messages to the broker |
| consume* | Requesting consumption from the broker |
| deliver* | Delivering messages to the consumer |
| restart* | Restarting the connection |
| new | Creating a new queue after declaration |
| add | Adding an element to the queue |
| hd | Gaining the header of the queue |
| rem | Removing elements from the queue |

In our model, we assume that there are four clients. When we check the fundamental properties, we assume that they are all valid. However, in order to check whether intruders can communicate in the system, we also introduce intruders to our model. Thus, when we verify the security property, we set that there is a valid producer whose ID is *0*, a valid consumer whose ID is *2*, a fake producer whose ID is *1* and a fake consumer whose ID is *3*.

For messages delivered among clients (i.e., producers and consumers), brokers and entities of KDC (i.e., AS and TGS) are abstracted to struct types. Their encrypted keys and contents are listed in Table 1, and the detailed definition is given in the next subsection.

Among these variables, these with the same meaning are described as a uniform form. The * character in $msg *$ and $demsg *$ matches $A$, $B$, $D$, $E$, $F$ and $G$ in our model to represent the corresponding message which is presented in Table 1.

### 3.1.3 Functions

Besides, we explain the functions used in our model. Table 4 enumerates these functions and their functionalities. Similar to the definitions of variables, the * character matches different messages.

## 3.2 Modeling processes

### 3.2.1 Overall modeling

To model the architecture of RabbitMQ with Kerberos, we formalize it as a network of seven timed automata in parallel. The system declaration is as below.

**Table 3** Variables in RabbitMQ with Kerberos

| Variable | Type | Meaning |
|---|---|---|
| typedef int[0,3] client | Global | The client's (i.e., producer's or consumer's) ID |
| typedef int[0,1] producer | Global | The produce's ID |
| typedef int[2,3] consumer | Global | The consumer's ID |
| typedef struct msg* | Global | Messages transmitted in the processes of Kerberos authentication |
| bool pend/ cend/ bpend/ bcend | Global | Representing whether the producer/ consumer/ brokerp/ brokerc ends |
| const int CKey/ TKey/ SKey | Global | Client/ TGS/ Server's secret key stored in the database |
| const int CTSessionKey/ CSSessionKey | Global | Client-TGS/ Client-Server session key stored in the database |
| const int N | Global | The capacity of the queue |
| int cnt | Global | The current number of messages |
| int body | Global | The message contents |
| clock t | Global | Recording the global time |
| clock bpt/ bct | Local | Recording the time interval of the BrokerP/ BrokerC automaton |
| bool clientvalid | Local | Representing whether the client exists in the database |
| bool servervalid | Local | Denoting if the requested server is in the database |
| bool demsg* | Local | Indicating whether this message can be decrypted |
| bool serverack | Local | Implying if the server has authenticated the client |
| bool heart | Local | Denoting the heartbeat between the producer/ consumer and the broker |
| int lifetime | Local | The lifetime of the TGT/ Ticket |
| int tgssessionkey/ cssessionkey | Local | Client-TGS/ Client-Server session key received by the client |
| int ctsessionkey | Local | Client-TGS session key received by TGS |

**Table 4** Functions in RabbitMQ with Kerberos

| Function | Functionality |
|---|---|
| request | Sending initial request to AS |
| clientValid | Checking if the client's ID is in the database |
| serverValid | Checking if the server's ID is in the database |
| sendMsg* | Sending encrypted messages |
| deMsg* | Decrypting the encrypted messages |
| authen | Authenticating the identity of the client |
| declareQueue | Declaring a queue |
| publishMsg | Publishing messages to the broker |
| consumeMsg | Sending consumption requests to the broker |
| deliverMsg | Delivering messages to the consumer |
| checkHB | Checking heartbeat |
| reset | Resetting |

*system AS*, *TGS*, *Producer*, *Consumer*,
*BrokerP*, *BrokerC*, *Queue*;

- ***AS*** **and** ***TGS***. They realize the Kerberos protocol and their parameter are clients' IDs.
- ***Producer*** **and** ***Consumer***. They simulate the behaviors of producing and consuming messages. The parameters of them are producers' IDs and consumers' IDs respectively.
- ***BrokerP*** **and** ***BrokerC***. They describe the processes of the broker when receiving messages from the producer and sending messages to the consumer. Likewise, their parameters contain producers' IDs and consumers' IDs.
- ***Queue***. It contains the basic queue operations and indicates the states of the queue.

### 3.2.2 Queue automaton

The model of the queue is presented in Fig. 4. It uses an array of integers to handle it as a First In First Out (FIFO) queue.

- ***Init.*** It represents that the queue is waiting for commands from the broker. The *Queue* Automaton realizes the fundamental operations of the queue, including declaring a queue (through the channel *new*), adding a new element to the queue (through the channel *add*) and gaining the front element from the queue (through the channel *hd*).
- ***Shiftdown.*** The *Queue* automaton can also remove elements from the queue (through the channel *rem* and the *Shiftdown* location). This is an intermediate location used to compute a shift of the queue when the front element is removed from the queue. When an element is removed, all the remaining elements of the queue should be shifted by one position. We utilize the loop in this location and mark it as a committed location to avoid unnecessary interleavings.

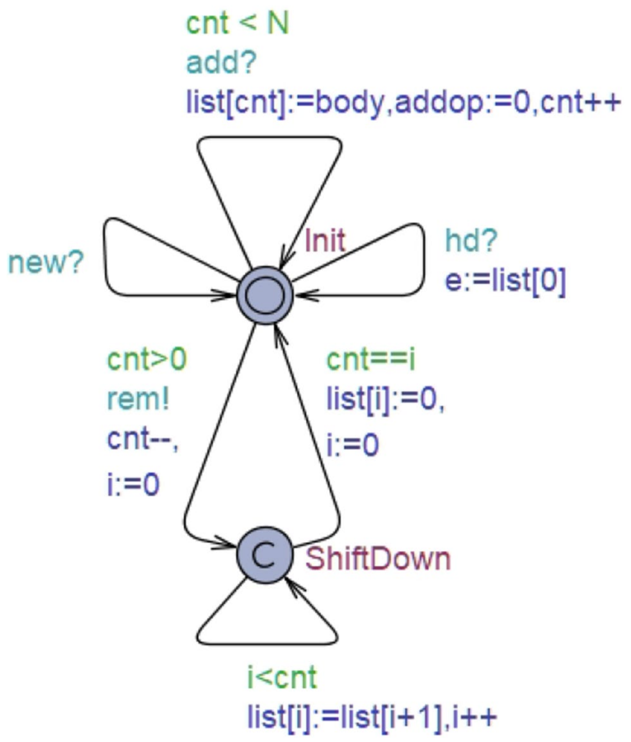


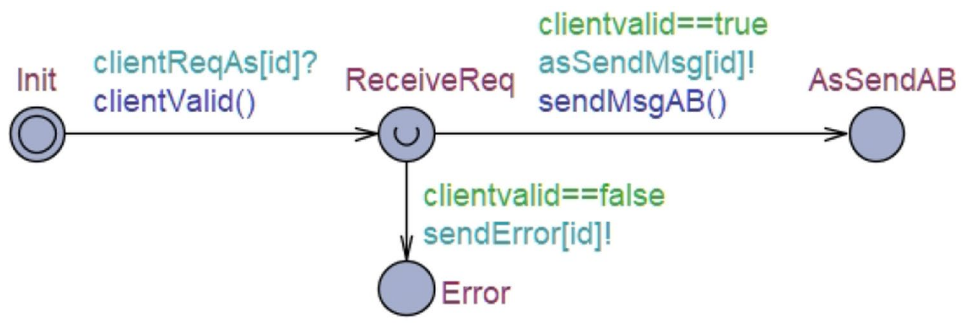
**Fig. 4** Queue automaton

### 3.2.3 AS automaton

The *AS* automaton is depicted in Fig. 5. It contains four locations and it is mainly responsible for authenticating the client's ID and granting TGT to the client.

- ***Init.*** It marks the initial state of AS.
- ***ReceiveReq.*** When AS receives the request from the client, it checks whether the client's ID exists in the database through the function *clientValid()* and then it reaches *ReceiveReq*. If exists, the variable *clientvalid* is set to *true*. Otherwise, *clientvalid* is set to *false*. Here, this location is an urgent one that freezes time. We design it because we assume that it takes no time to determine the validity of the client's ID after receiving the request.
- ***AsSendAB.*** If *clientvalid* is *true*, AS sends *MsgA* and *MsgB* (*MsgB*'s encrypted content is TGT) to the client through the function *sendMsgAB()*.

**Fig. 5** AS automaton



– ***Error.*** If *clientvalid* is *false*, the automaton runs into *Error* location which indicates decryption failure happens. With the channel *sendError*, AS notifies the producer of this error.

Here, we take *MsgB* as an example to illustrate the definitions of messages in our model. Other messages listed in Table 1 are abstracted similarly, and for brevity, we omit the details. Following the *C* notation, we use *typedef* to create a struct to record messages. As defined below, *msgB* consists of the five fields *key*, *sessionkey*, *client_id*, *server_id* and *life*. The first component key records the encrypted key of *MsgB*. The rest four fields represent the contents (i.e., CTSessionKey, ClientID, ServerID, Lifetime) of *MsgB* respectively.

```
typedef  struct{
            int key;
            int sessionkey;
            int client_id;
            int server_id;
            int life;
}msgB;
 msgB   MsgB[4];
```

Also, we take *sendMsgAB()* for instance to explain the functions of *sendMsg**. As shown below, parameters of *MsgA* and *MsgB* are assigned to the values stored in the database of AS or received from the client. Specifically, *MsgB* is encrypted by the client's secret key *CKey* which is stored in the database. *CTSessionKey* and *lifetime* in the encrypted contents are generated by AS, other contents are received from the request of the client.

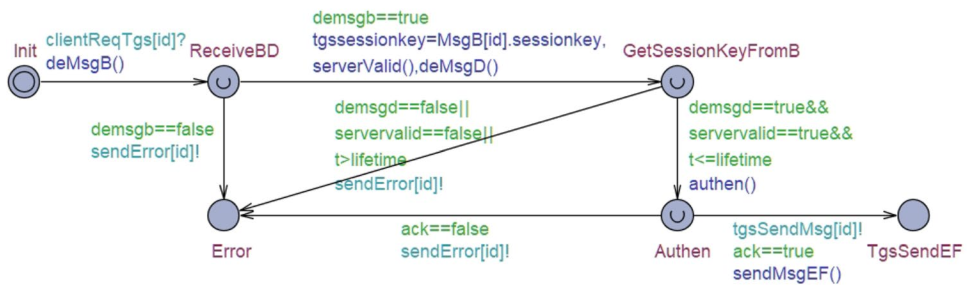
```
void sendMsgAB(){
        MsgA[id].key := CKey;
        MsgA[id].sessionkey := CTSessionKey;
        MsgB[id].key := TKey;
        MsgB[id].sessionkey := CTSessionKey;
        MsgB[id].client_id := id;
        MsgB[id].server_id := serverid;
        MsgB[id].life := lifetime;}
```

### 3.2.4 TGS automaton

As shown in Fig. 6, TGS automaton is in charge of verifying Authenticator and providing Ticket for the client. It is composed of the following six locations.

– ***Init.*** It represents the initial state of TGS.
– ***ReceiveBD.*** After the client has communicated with AS, it sends a request which includes *MsgB* and *MsgD* to TGS. The *TGS* automaton sets the local clock *tgst* to *0*, attempts to decrypt *MsgB* with its own secret key, and moves from the initial state to *ReceiveBD*. Since we assume that decryption also takes no time, we mark this location as an urgent one.
– ***GetSessionKeyFromB.*** If *MsgB* is decrypted successfully, TGS gains *tgssessionkey*. Then, it checks if the

**Fig. 6** TGS automaton



server is valid and decrypts *MsgD* with the received key. Similarly, it is an urgent location.

- ***Authen.*** After checking the contents of *MsgB* and decrypting *MsgD*, it verifies the Authenticator through the function *authen()* and gets here. We see authentication of the Authenticator as an instantaneous action. Thus, this location is urgent as well.
- ***TgsSendEF.*** If the Authenticator is verified, TGS sends *MsgE* (whose encrypted content is Ticket) and *MsgF* to the client. Then, the automaton runs into this location.
- ***Error.*** If an accident (such as decryption failure, unsuccessful authentication and timeout) occurs, the automaton fails to the *Error* location and uses the channel *sendError* to inform the client.

Here, we present the function *deMsgB()* as an example to show the definitions of the functions *deMsg**.

```
void deMsgB(){
    if(MsgB[id].key == TKey&&tgskey == TKey)
    demsgb := true;
    else   demsgb := false;}
```

As defined above, if TGS's own secret key *tgskey* equals *TKey* and the encrypted key of *MsgB* also equals *TKey*, the variable *demsgb* is assigned to *true*. Otherwise, *MsgB* cannot be decrypted and *demsgb* is set to *false*.

### 3.2.5 Producer automaton

Figure 7 presents the Producer Automaton. It realizes the conduction of Kerberos protocol and the production of messages.

- **Conducting Kerberos Protocol.**
  Different from the *Producer* automaton in [14], the producer must pass the authentication of Kerberos protocol before producing messages to the broker. The fol-

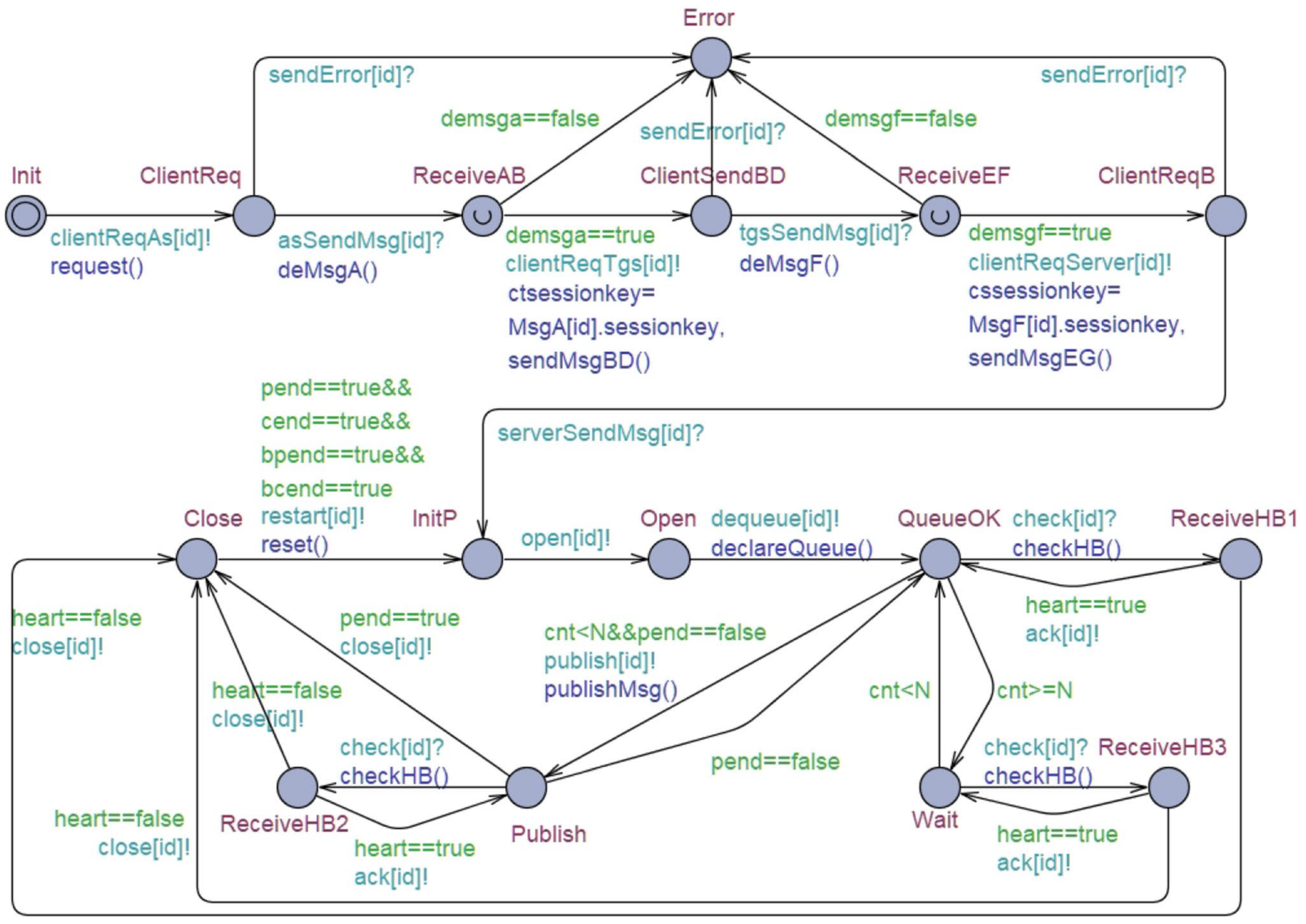


**Fig. 7** Producer automaton

lowing seven locations are used to conduct the Kerberos protocol.

- ***Init.*** It represents the initial state of the producer.
- ***ClientReq.*** The producer first sends a request (which includes its own ID and the corresponding server's ID) to AS through the function *request()*. Consequently, it transforms from *Init* to *ClientReq* location.
- ***ReceiveAB.*** The producer waits for messages sent from AS. Once it receives *MsgA* and *MsgB*, it applies its own secret key to decrypt *MsgA* and moves to this location.
- ***ClienrSendBD.*** If *MsgA* is decrypted (i.e., *demsga* equals to *true*), the producer attains *ctssessionkey* from *MsgA* which supports its later communication with TGS. Next, it sends *MsgB* and *MsgD* to TGS and the automaton gets here.
- ***ReceiveEF.*** When the producer gets *MsgE* and *MsgF* from TGS, it decrypts *MsgF* with *ctssessionkey* which is gained from AS and reaches *ReceiveEF*.
- ***ClientReqB.*** The producer obtains *cssessionkey* from *MsgF* and sends *MsgE* and *MsgG* to the broker. The automaton arrives at *ClientReqB* which indicates the producer applies for interaction with the broker.
- ***Error.*** It jumps to *Error* location if abortive decryption occurs.

- **Producing Messages.**
  After authentication, the producer starts to produce messages to the broker. The automaton simulates the processes of production by the rest seven locations.

  - ***InitP.*** It indicates that the producer can prepare to produce messages if the broker returns an ack to the producer.
  - ***Open.*** It means the producer opens the connection through the channel *open*.
  - ***QueueOK.*** With the channel *dequeue* and the function *declareQueue()*, the producer declares the queue and the automaton arrives at this location.
  - ***Publish.*** If the number of current messages *cnt* is less than *N*, i.e., the queue is not full, it reaches this location where the producer publishes messages.
  - ***Wait.*** If the number of current messages *cnt* equals to or greater than *N*, i.e., the queue is full, the producer reaches here, which means that the producer needs to wait until the queue is not full.
  - ***Close.*** If the producer ends the communication or the heartbeat timeout happens, the automaton jumps to *Close* location. Further, it can utilize the function *reset()* to move to *InitP* location and begin production again.
  - ***ReceiveHB*.*** We check the heartbeat after the establishment of the connection between the producer and the broker. If the time interval is greater than the maximal time interval, the producer receives the check of heartbeat with the function *checkHB()* and then it will jump to *ReceiveHB** location. Here, the character * matches *1*, *2* and *3*. If the variable *heart* is *true*, it continues the following procedures.

Here, we use the *publishMsg()* function to publish messages to the broker. In this function, we neglect the real contents of the messages but we focus on the order of these messages. We use the variable *body* to mark the order. The initial value of *body* is set to *1* and it increases progressively with the increase of messages.

### 3.2.6 Consumer automaton

As illustrated in Fig. 8, Consumer Automaton formalizes the behaviors of the consumer. It takes charge of conducting Kerberos protocol and consuming messages.

- **Conducting Kerberos Protocol.**
  The names and meanings of locations that defined to conduct the Kerberos protocol are not be repeated here, because they stay the same as those in the *Producer* Automaton.
- **Consuming Messages.**

  - ***InitC.*** If the broker returns an ack to the consumer, the consumer arrives at this location.
  - ***Open.*** It means the connection has been opened.
  - ***Consume.*** Through the channel *consume* and the function *consumeMsg()*, the consumer sends consumption requests to the broker, and it jumps to *Consume* location.
  - ***Deliver.*** This location indicates that the broker delivers messages to the consumer and the corresponding function is executed in the *BrokerC* Automaton which is presented later.
  - ***Wait.*** If there is no message available to the consumer, the consumer reaches to *Wait*.
  - ***Close.*** Similarly, the automaton jumps to here when the consumer ends the communication or the heartbeat timeout happens.
  - ***ReceiveHB*.*** The *Consumer* automaton also supports the heartbeat mechanism like that in the *Producer* automaton.

### 3.2.7 Broker automata

Since the producer publishes messages to the broker and the consumer consumes messages from the broker, to help

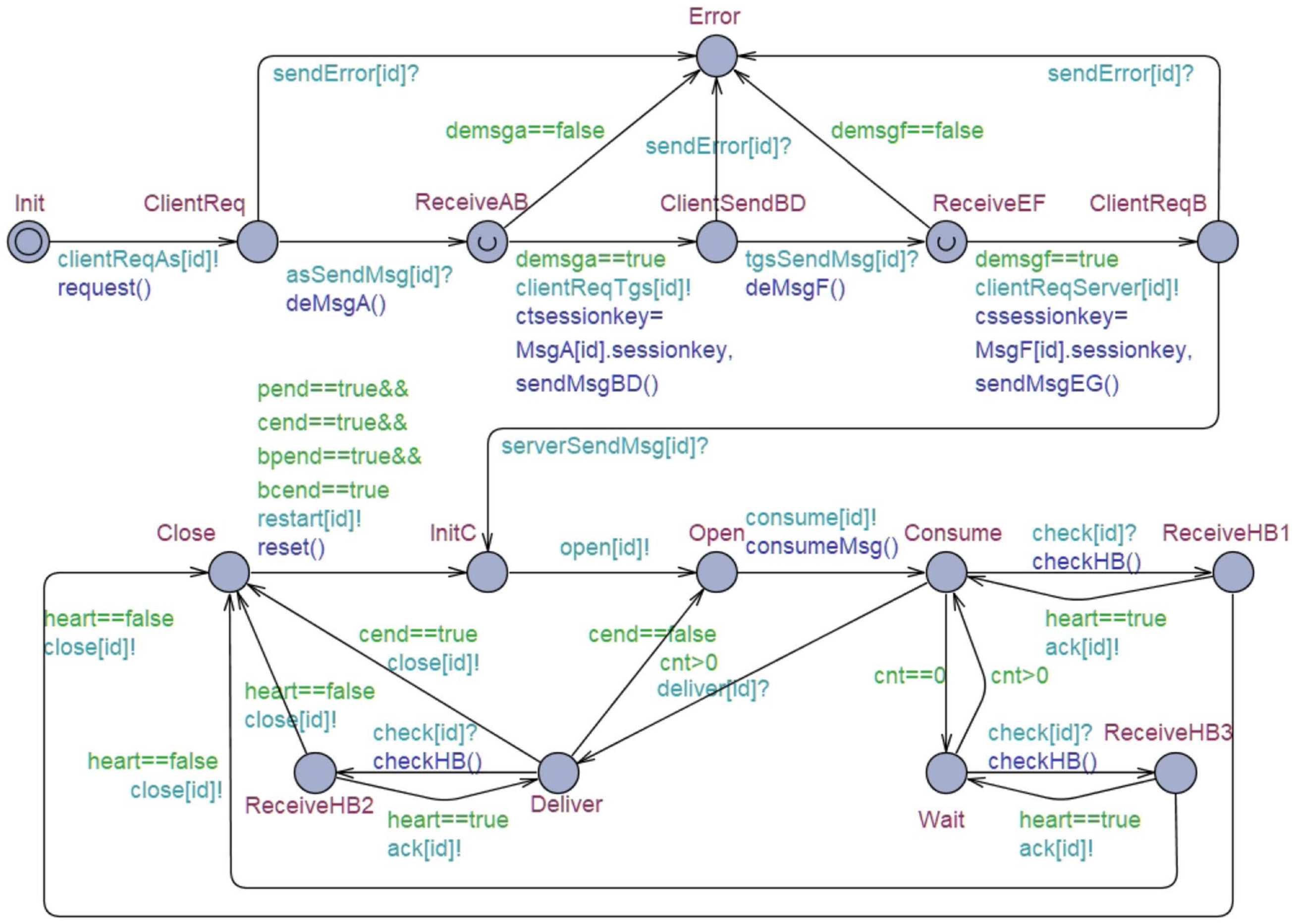


**Fig. 8** Consumer automaton

describe this model, we divide the broker model into two automata: *BrokerP* Automaton and *BrokerC* Automaton.

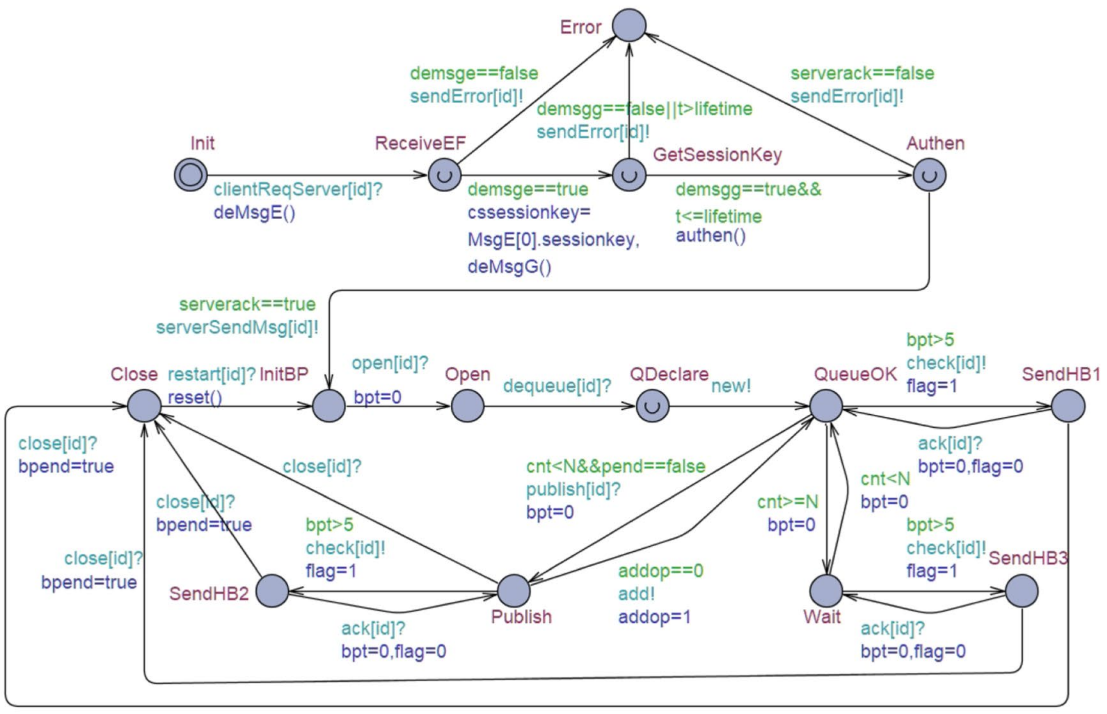
**(1) BrokerP Automaton.** The *BrokerP* Automaton is displayed in Fig. 9(a) and it simulates the broker when the producer requests to publish messages.

- **Authenticating.**
  Before providing services to the producer, the broker first authenticates the producer's identification through the Kerberos protocol. There are five locations formalized to realize the flows of authentication.

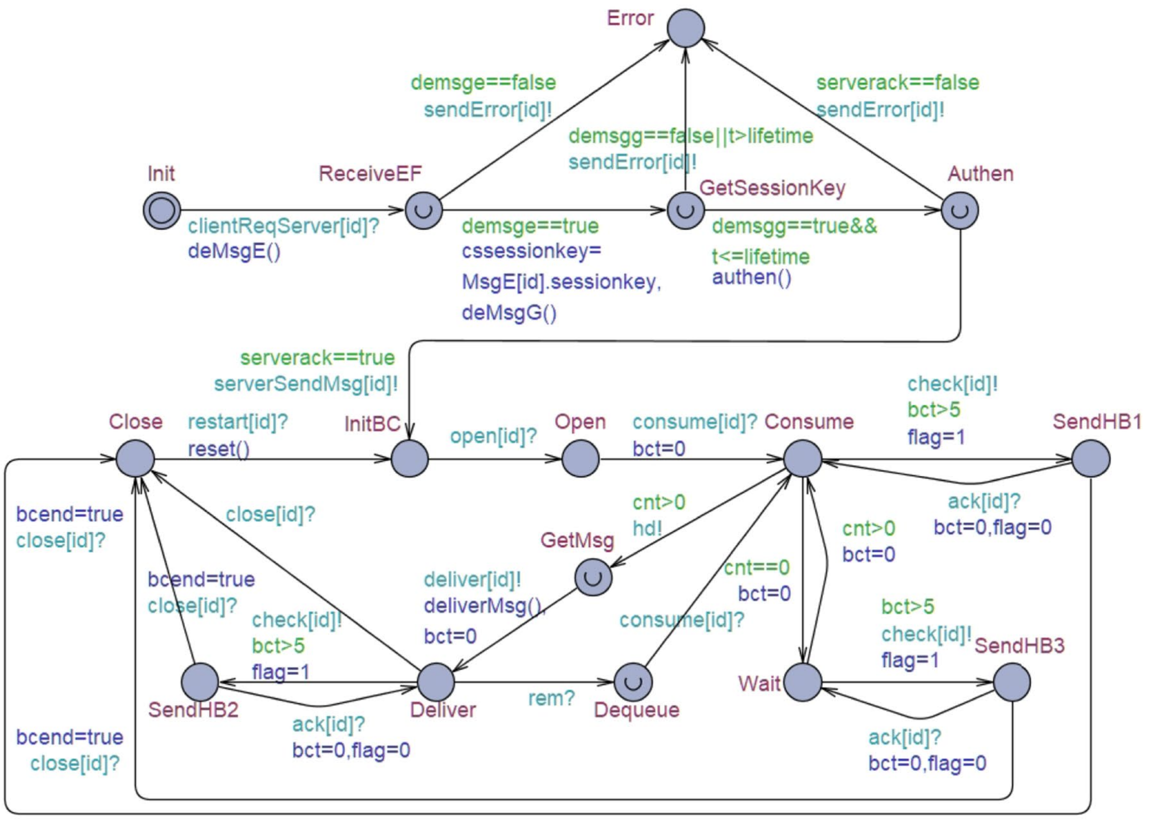
  - ***Init.*** It is the initial location of *BrokerP* automaton.
  - ***ReceiveEF.*** Once the broker receives messages from the producer, it first uses its own secret key to decrypt *MsgE* and reaches this location.
  - ***GetSessionKey.*** If the broker cracks *MsgE* successfully, it can get the *cssessionkey* and adopts this key to decrypt *MsgG*. Then, it jumps to here.
  - ***Authen.*** After decrypting *MsgG* and checking lifetime successfully, the broker uses the function *authen()* to authenticate the producer and the automaton is at this location.
  - ***Error.*** This location indicates there is an error such as failed decryption and overtime tickets.

- **Handling Production.**
  If the producer has been authenticated, the broker applies the channel *serverSendMsg* to indicate this successful authentication and then the producer can produce messages to the broker. The rest locations simulate the behaviors of the broker when the producer sends messages.

  - ***InitBP.*** It indicates that the producer can prepare to produce messages if the broker returns an ack to the producer.
  - ***QueueOK.*** After the producer declares a queue with the channel *dequeue*, the *BrokerP* automaton moves to *QDeclare* location. Next, it communicates with the *Queue* automaton through the channel *new* and it moves to this location. The time



(a)



(b)

**Fig. 9** Broker automata

spent on declaring queue is not considered in our model, so we set *QDeclare* as an urgent location.

- ***Publish.*** If the queue is not full, the broker moves to *Publish* location. If the message has not been added into the queue (i.e., *addop* equals *0*), the *BrokerP* automaton informs the *Queue* automaton to add this message through the channel *add* and then sets *addop* to *1*.
- ***ReceiveHB*.*** As listed in Table 3, the variable *bpt* is a local clock defined to record the time interval between the producer and the broker. When the automaton reaches *Open* location, it implies that the connection between the producer and the broker has been established. Hence, we set *bpt* to *0* and check the heartbeat in the subsequent locations except for the urgent location *QDeclare*. In our model, we assume the maximal time interval is *5* seconds. Thus, the broker checks the heartbeat, if the time interval exceeds *5* seconds. If the broker sends a heartbeat through the channel *check*, the variable *flag* is assigned to *1*. Then, *flag* and *bpt* are reset to *0* when the broker receives a response from the producer through the channel *ack*.

**(2) BrokerC Automaton.** Figure 9(b) presents the *BrokerC* Automaton. Similar to the *BrokerP* automaton, it can authenticate the consumer's ID and simulate the behaviors when consumption occurs.

- **Authenticating.**
  The details of authenticating are the same as those in the *BrokerP* automaton, so we omit them here.
- **Handling Consumption.**
  Analogously, the broker handles the consumption after authenticating the consumer. The rest locations realize the procedures. Most of them have the same meanings as those in the *Consumer* automaton, and we focus on locations involved with consumption.

  - ***InitBC.*** It implies the broker has authenticated the consumer.
  - ***GetMsg.*** After the broker receives the request of consumption, it contacts the queue with the channel *hd* and runs into the *GetMsg* location which means that the consumer can fetch the message from the queue. Here, *GetMsg* is an urgent location that costs no time.
  - ***Dequeue.*** After getting the message and delivering it to the consumer, the broker reaches this location which indicates that the message has been removed from the queue.

## 4 Validation and verification

In this section, we first validate our model with the assistance of the simulator in UPPAAL. Then, we employ the model checker of UPPAAL to verify several properties of the constructed model.

### 4.1 Model validation

We validate the correctness of our model by simulation, and Fig. 10 illustrates the Message Sequence Charts returned from UPPAAL. As shown in Fig. 10(a), we take the producer as an example to explain the flows of authentication in our model. Figure 10(b) simulates the interactions among the producer, broker and queue when the producer produces messages. Figure 10(c) reflects the procedures of consumption among the consumer, broker and queue.

We can find that all the communication among entities is consistent with the specification of RabbitMQ and Kerberos protocol. Hence, the correctness of our modeling is validated.

### 4.2 Properties verification

With the model checker in UPPAAL, we verify several important properties of RabbitMQ with Kerberos. We first check four fundamental properties of the model under the assumption that all clients are valid. Next, we introduce invalid clients to verify whether the model ensures secure communication.

#### 4.2.1 Fundamental properties

We verify four basic properties of the constructed model, including *Reachability of Data*, *Concurrency*, *Sequence Consistency* and *Heartbeat Mechanism*.

**Property 1: Reachability of Data.**

The property refers to the reachability of messages. The first formula denotes that for every producer, if it reaches *Publish* location, the message should be added into the queue. Similarly, the second formula indicates that for every consumer, once it reaches *Deliver* location, the message should be consumed. The two-dimensional array *cmsg[i][k]* stores the *k* th message that the consumer *i* received.

$$E\langle\rangle\ forall(p : producer)\ Producer(p).Publish\ imply\ cnt! = 0$$

$$E\langle\rangle\ forall(c : consumer)\ Consumer(c).Deliver\ imply\ cmsg[c][0]! = 0$$

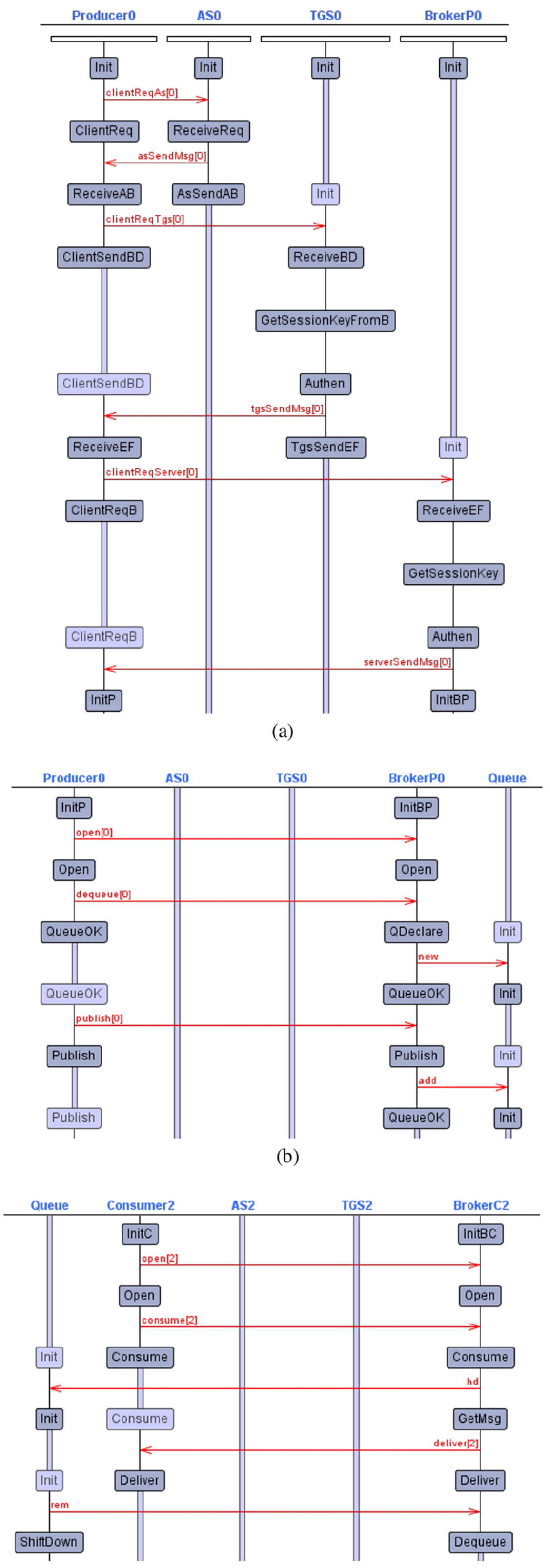


**Fig. 10** Message sequence charts

**Property 2: Concurrency.**

The property represents that RabbitMQ supports concurrent operations, i.e., the producer can produce messages for many consumers. Since we assume there are *Consumer(2)* and *Consumer(3)* in our model, if there is a path that satisfies *Consumer(2)* and *Consumer(3)* can both consume messages successfully, the property is satisfied.

$$E\langle\rangle\ cmsg[2][0]! = 0\&\&cmsg[3][0]! = 0$$

**Property 3: Sequence Consistency.**

The property means that the order in which the messages are consumed by the consumer and those produced by the producer is the same. The first formula describes that the producer generates messages in ascending order. While the second formula shows that the consumer always consumes messages in ascending order as well. If the two formulas are both true, the property of sequence consistency is fulfilled.

$$\begin{aligned}&A\langle\rangle\ forall(i : id\_q)\,forall(j : id\_q)\\&\quad i < j\&\&Queue.list[j]! = 0\\&\quad imply\ Queue.list[i] < Queue.list[j]\\&A\langle\rangle\ forall(c : consumer)\,forall(i : id\_q)\,forall(j : id\_q)\\&\quad cmsg[c][i]! = 0\&\&cmsg[c][j]! = 0\&\&i < j\\&\quad imply\ cmsg[c][i] < cmsg[c][j]\end{aligned}$$

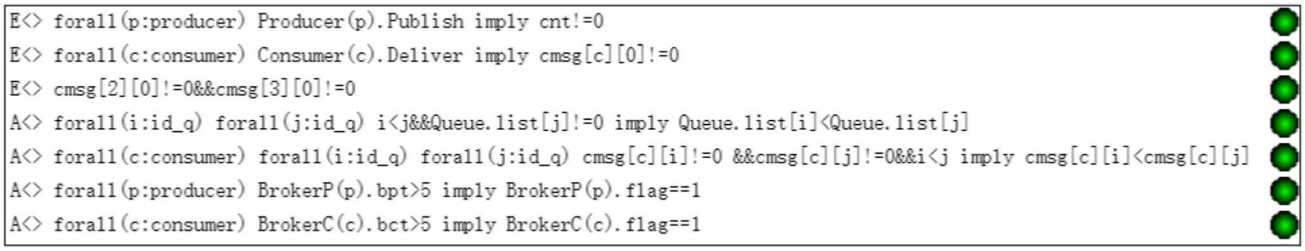
**Property 4: Heartbeat Mechanism.**

This property is designed to check the heartbeat mechanism. When the time interval exceeds *5* seconds, the broker checks the heartbeat of the producer and the consumer. The variables *bpt* and *bct* are the local clocks in *BrokerP* automaton and *BrokerC* automaton, and the variable *flag* is defined to mark the sending of the heartbeat.

$$\begin{aligned}&A\langle\rangle\ forall(p : producer)\ BrokerP(p).bpt > 5\\&\qquad imply\ BrokerP(p).flag == 1\\&A\langle\rangle\ forall(c : consumer)\ BrokerC(c).bct > 5\\&\qquad imply\ BrokerC(c).flag == 1\end{aligned}$$

**Verification Results.**

UPPAAL 4.1.24 runs on the machine with *Core i7* processor and *32G* memory. By feeding the constructed model to UPPAAL, we verify the above four properties. The verification results are shown in Fig. 11. According to it, we can find RabbitMQ can transmit messages among the producer, the broker and the consumer. Also, it supports concurrency, guarantees sequential consistency and realizes the heartbeat mechanism.



```
E<> forall(p:producer) Producer(p).Publish imply cnt!=0
E<> forall(c:consumer) Consumer(c).Deliver imply cmsg[c][0]!=0
E<> cmsg[2][0]!=0&&cmsg[3][0]!=0
A<> forall(i:id_q) forall(j:id_q) i<j&&Queue.list[j]!=0 imply Queue.list[i]<Queue.list[j]
A<> forall(c:consumer) forall(i:id_q) forall(j:id_q) cmsg[c][i]!=0 &&cmsg[c][j]!=0&&i<j imply cmsg[c][i]<cmsg[c][j]
A<> forall(p:producer) BrokerP(p).bpt>5 imply BrokerP(p).flag==1
A<> forall(c:consumer) BrokerC(c).bct>5 imply BrokerC(c).flag==1
```

**Fig. 11** Verification results of fundamental properties

### 4.2.2 Security property

In our model, we set *Producer(1)* and *Consumer(3)* as invalid clients to verify that RabbitMQ with Kerberos is secure in the presence of intruders.

**Property 5: Secure Communication.**

Considering secure communication, we check the following three assertions. The first two imply that the invalid clients will encounter errors during the authentication. The last assertion states that the two automata can never reach *Publish* location and *Deliver* location, and it indicates that any invalid producer or consumer cannot produce or consume messages.

$E\langle\rangle \ Producer(1).Error$
$E\langle\rangle \ Consumer(3).Error$
$A[] \ !(Producer(1).Publish \ || \ Consumer(3).Deliver)$

**Verification Results.**

For this property, two scenarios are set up to check it. We give the detailed verification results in Fig. 12.

– **Scenario 1.** The fake client attempts to participate in the communication with an invalid ID. We achieve this by defining the function *clientValid()*, i.e., we assume only *0* and *2* are valid IDs.
– **Scenario 2.** The fake client uses a valid ID to join the communication. We achieve this by changing the judgment condition of the function *clientValid()*. That is, we assume all the IDs are valid.

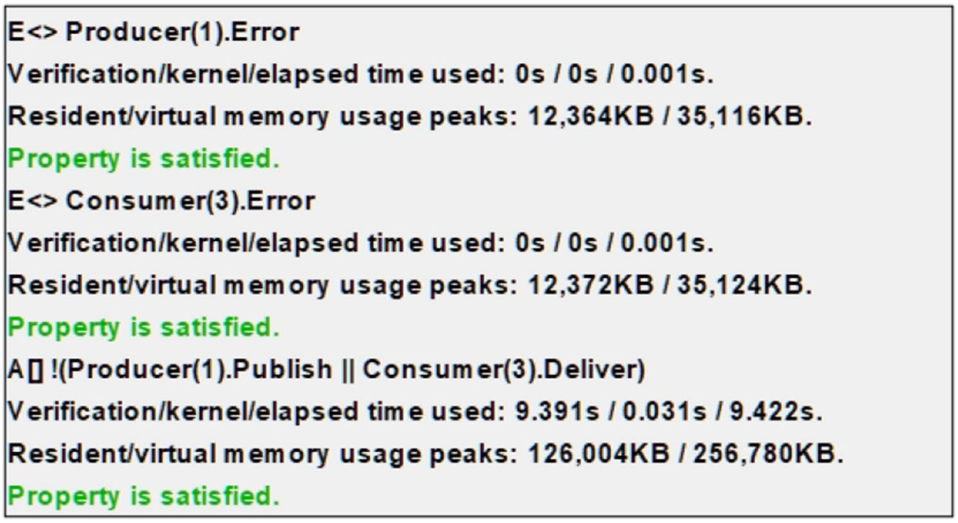
It can be found that no matter in which scenario, RabbitMQ ensures secure communication with the authentication of Kerberos. However, the verification costs more time in Scenario 2 than in Scenario 1. It makes sense, as shown in Fig. 13, the simulation traces returned from UPPAAL indicate that AS rejects the invalid request at the very beginning in Scenario 1. While, in Scenario 2, the fake client gains the encrypted messages from AS, but fails in decryption for lack of the relevant secret key.

## 5 Related work

Due to the information explosion, the message queue is an efficient solution to deliver messages in the distributed system [1]. Many message queue middlewares have been proposed and widely used, including RabbitMQ [4], Kafka [5], ActiveMQ [6], RocketMQ [7], etc. Some research about the application of these middlewares has been carried out. Matic et al. introduced RabbitMQ to smart home systems and focused on monitoring and auto-scaling the state of it [18]. Prabhu et al. applied Object Knowledge Model (OKM) and Kafka to extend knowledge management (KM) models [19]. Ofenloch et al. presented a flexible distributed simulation environment for Cyber Physical Systems (CPS) with the help of ZeroMQ [20]. Besides, an integration model of ZeroMQ and blockchain was proposed in [21] to guarantee personal privacy protection and data security for E-Voting. It is the widespread adoption that persuades us to focus on message queue middlewares. In this paper, we choose RabbitMQ which is a popular and typical one to analyze properties.

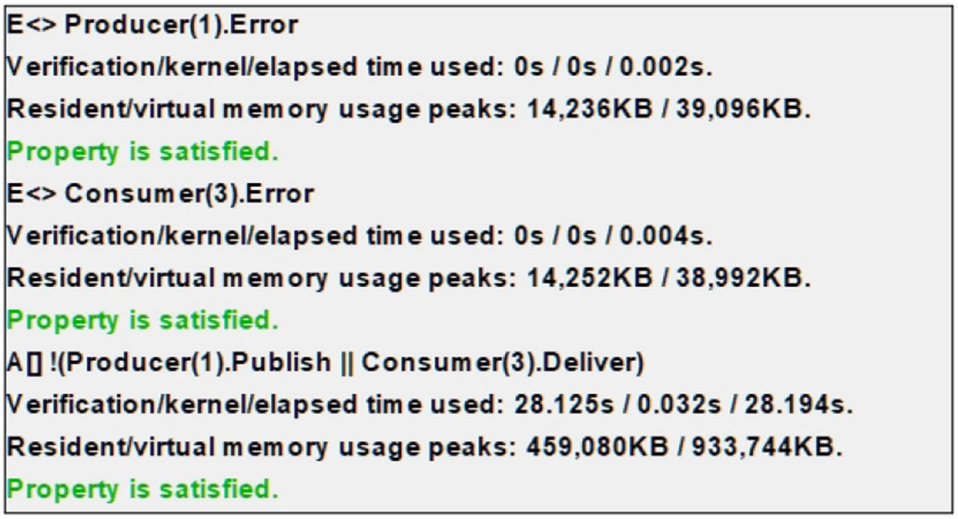
Meanwhile, there are many research works on the comparison of RabbitMQ and other message queue middlewares. Hong et al. evaluated the performance of RabbitMQ and REST API in different circumstances [22]. Dobbelaere et al. proposed a qualitative and quantitative comparison framework of RabbitMQ and Kafka [17]. Estrada et al. investigated the scalability of RabbitMQ and ZeroMQ [23]. Ionescu analyzed the performance of RabbitMQ and ActiveMQ [24]. The above works have made great contributions to the analysis of RabbitMQ, and they are mainly based on experiments for performance evaluation. To our best knowledge, there is nearly no work on the analysis of RabbitMQ using formal methods. Therefore, our paper applies formal methods to construct a model of RabbitMQ.

Formal methods have been applied widely, since they can help researchers model and verify systems from a mathematical point of view. Rodríguez et al. undertook a series



```
E<> Producer(1).Error
Verification/kernel/elapsed time used: 0s / 0s / 0.001s.
Resident/virtual memory usage peaks: 12,364KB / 35,116KB.
Property is satisfied.
E<> Consumer(3).Error
Verification/kernel/elapsed time used: 0s / 0s / 0.001s.
Resident/virtual memory usage peaks: 12,372KB / 35,124KB.
Property is satisfied.
A[] !(Producer(1).Publish || Consumer(3).Deliver)
Verification/kernel/elapsed time used: 9.391s / 0.031s / 9.422s.
Resident/virtual memory usage peaks: 126,004KB / 256,780KB.
Property is satisfied.
```

(a)



```
E<> Producer(1).Error
Verification/kernel/elapsed time used: 0s / 0s / 0.002s.
Resident/virtual memory usage peaks: 14,236KB / 39,096KB.
Property is satisfied.
E<> Consumer(3).Error
Verification/kernel/elapsed time used: 0s / 0s / 0.004s.
Resident/virtual memory usage peaks: 14,252KB / 38,992KB.
Property is satisfied.
A[] !(Producer(1).Publish || Consumer(3).Deliver)
Verification/kernel/elapsed time used: 28.125s / 0.032s / 28.194s.
Resident/virtual memory usage peaks: 459,080KB / 933,744KB.
Property is satisfied.
```

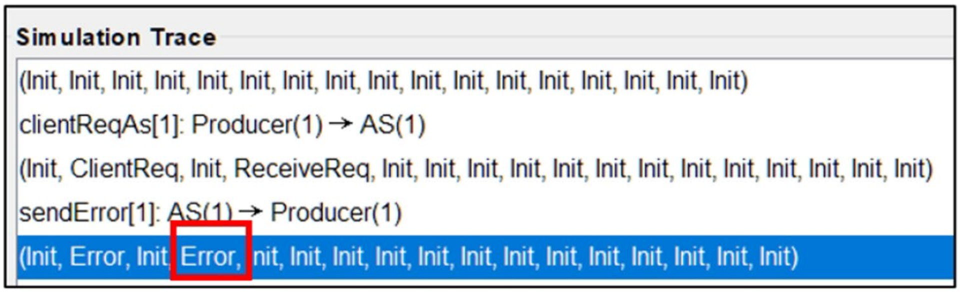
(b)

**Fig. 12** Detailed verification results of security property

of research on the Message Queuing Telemetry Transport (MQTT) protocol using Coloured Petri Nets (CPNs) [25–28]. Xu et al. formalized and verified the Producer-Consumer communication in Kafka with the process algebra Communicating Sequential Processes (CSP) [29]. Different from these methods, we employed UPPAAL [15, 16] to explore some inherent properties of RabbitMQ in our previous work [14]. Many additional works using the same formal methods also exist, due to the convenience and powerful capabilities of UPPAAL. Lin et al. utilized UPPAAL to model and verify Real-Time Publish and Subscribe (RTPS) protocol [30]. Fei et al. took the analysis of the Named-data Link State Routing (NLSR) protocol [31] and Sun et al. modeled and verified Common Knowledge Base (CKB) blockchain consensus protocol [32].

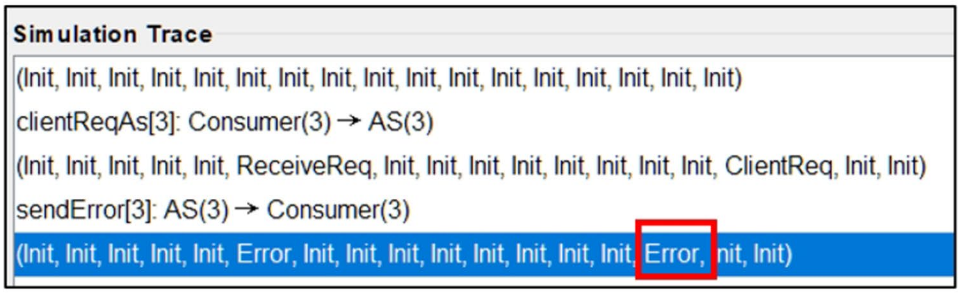
Our previous work focused on the fundamental properties of RabbitMQ [14]. Nevertheless, security property should not be ignored either. The most recent research described the security vulnerabilities of RabbitMQ by protocol fuzzing [33]. Hence, our paper now takes the security issue into consideration. Designed by Massachusetts Institute of Technology (MIT), Kerberos is a network authentication protocol that uses symmetric cryptography [13]. Li et al. proposed a scheme for secure offline delivery services based on Kerberos [34]. Xu et al. explored the identity authentication security of Hadoop Distributed File System (HDFS) which is combined with Kerberos [35]. Inspired by them, we build a combination of RabbitMQ and Kerberos to address the security problem in our paper. With the aid of UPPAAL, we construct the model of the combination of RabbitMQ and

**Fig. 13** Simulation traces of the security property



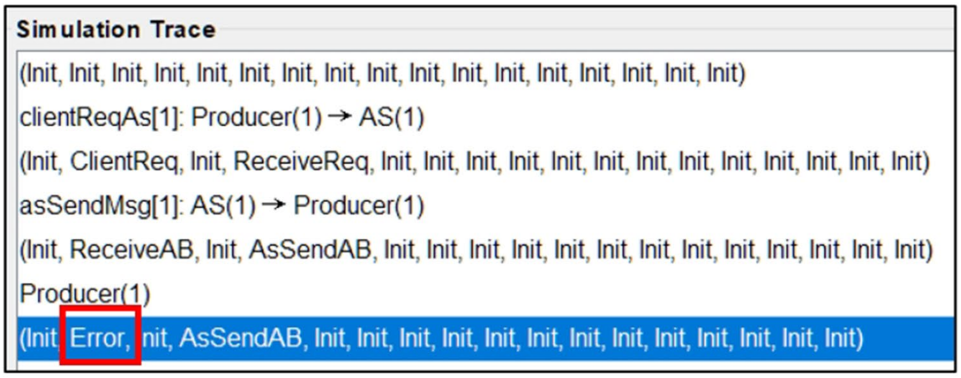
```
Simulation Trace
(Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init)
clientReqAs[1]: Producer(1) → AS(1)
(Init, ClientReq, Init, ReceiveReq, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init)
sendError[1]: AS(1) → Producer(1)
(Init, Error, Init, Error, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init)
```

(a)



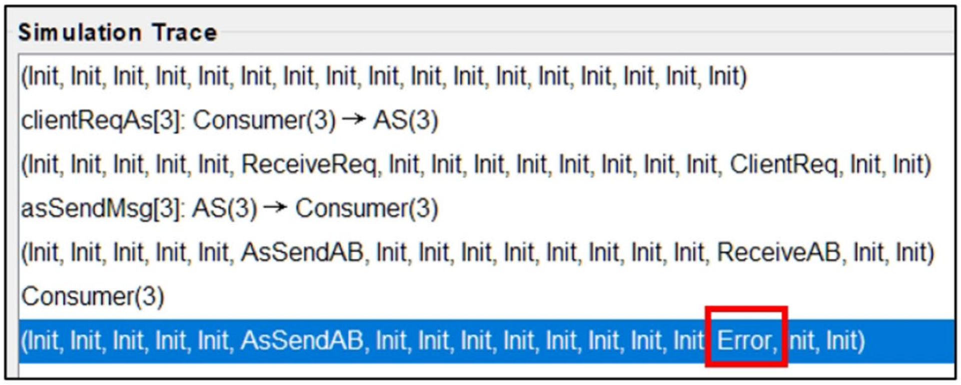
```
Simulation Trace
(Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init)
clientReqAs[3]: Consumer(3) → AS(3)
(Init, Init, Init, Init, Init, ReceiveReq, Init, Init, Init, Init, Init, Init, Init, Init, ClientReq, Init, Init)
sendError[3]: AS(3) → Consumer(3)
(Init, Init, Init, Init, Init, Error, Init, Init, Init, Init, Init, Init, Init, Init, Error, Init, Init)
```

(b)



```
Simulation Trace
(Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init)
clientReqAs[1]: Producer(1) → AS(1)
(Init, ClientReq, Init, ReceiveReq, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init)
asSendMsg[1]: AS(1) → Producer(1)
(Init, ReceiveAB, Init, AsSendAB, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init)
Producer(1)
(Init, Error, Init, AsSendAB, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init)
```

(c)



```
Simulation Trace
(Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init, Init)
clientReqAs[3]: Consumer(3) → AS(3)
(Init, Init, Init, Init, Init, ReceiveReq, Init, Init, Init, Init, Init, Init, Init, Init, ClientReq, Init, Init)
asSendMsg[3]: AS(3) → Consumer(3)
(Init, Init, Init, Init, Init, AsSendAB, Init, Init, Init, Init, Init, Init, Init, Init, ReceiveAB, Init, Init)
Consumer(3)
(Init, Init, Init, Init, Init, AsSendAB, Init, Init, Init, Init, Init, Init, Init, Init, Error, Init, Init)
```

(d)

Kerberos, and verify four vital properties of it. The verification results imply that RabbitMQ satisfies all these properties, owing to the participation of Kerberos.

## 6 Conclusion and future work

RabbitMQ is a message queue middleware that uses the Erlang language to implement AMQP, which is rapidly gaining widespread adoption. To ensure communication security, we have combined RabbitMQ with the network authentication protocol Kerberos.

In this paper, we have formalized the architecture of RabbitMQ with Kerberos by abstracting it to timed automata. By implementing the constructed model in UPPAAL, we have validated its correctness by simulation. Further, we have utilized the model checker to verify five properties of this model, including *Reachability of Data*, *Concurrency*, *Sequence Consistency*, *Heartbeat Mechanism* and *Secure Communication*. The verification results show that RabbitMQ meets the first four basic properties. Additionally, it also caters for security property due to the combination of Kerberos.

Actually, there exist many other attacks in real-world communication. We will introduce other threats into our model and analyze them with formal methods in the future.

**Data Availability** Not Applicable.

## Declarations

**Competing interests** We have no competing interests to declare that are relevant to the content of this article. This article does not involve ethics issues.

## References

1. Kinoshita M, Konoura H, Koike T, Leibnitz K, Murata M (2017) High throughput dequeuing technique in distributed message queues for IoT. J Inf Process 25:199–208
2. Jiang Y, Liu Q, Qin C, Su J, Liu Q (2019) Message-oriented middleware: A review. In: BigCom. IEEE, pp 88–97
3. Liu Y, Zhang L J, Xing C (2020) Review for message-oriented middleware. In: Internet of Things - ICIOT 2020. Springer International Publishing, Cham, pp 152–159
4. VMware (2022) RabbitMQ. https://www.rabbitmq.com/. Accessed 6 March 2022
5. Apache (2022a) Kafka. https://kafka.apache.org/. Accessed 6 March 2022
6. Apache (2022b) ActiveMQ. https://activemq.apache.org/. Accessed 6 March 2022
7. Apache (2022c) RocketMQ. https://rocketmq.apache.org/. Accessed 6 March 2022
8. OASIS (2022) AMQP. https://www.amqp.org/. Accessed 6 March 2022
9. Appel S, Sachs K, Buchmann AP (2010) Towards benchmarking of AMQP. In: DEBS. ACM, pp 99–100
10. Hunkeler U, Truong HL, Stanford-Clark AJ (2008) MQTT-S - A publish/subscribe protocol for wireless sensor networks. In: COMSWARE. IEEE, pp 791–798
11. Mimouni SE, Bouhdadi M (2015) Formal modeling of the simple text oriented messaging protocol using event-b method. In: AICCSA. IEEE Computer Society, pp 1–4
12. Osinski T, Dandoush A (2018) XMPP as a scalable multi-tenants isolation solution for onos-based software-defined cloud networks. In: CNSM. IEEE Computer Society, pp 300–303
13. Neuman BC, Ts'o TY (1994) Kerberos: an authentication service for computer networks. IEEE Commun Mag 32(9):33–38
14. Li R, Yin J, Zhu H (2020) Modeling and analysis of RabbitMQ using UPPAAL. In: TrustCom. IEEE, pp 79–86
15. UPPAAL (2022) http://uppaal.org. Accessed 6 March 2022
16. Behrmann G, David A, Larsen KG (2004) A tutorial on UPPAAL. In: SFM, Springer, Lecture Notes in Computer Science, vol 3185, pp 200–236
17. Dobbelaere P, Esmaili KS (2017) Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry paper. In: DEBS. ACM, pp 227–238
18. Matic M, Ivanovic S, Antic M, Papp I (2019) Health monitoring and auto-scaling RabbitMQ queues within the smart home system. In: ICCE-Berlin. IEEE, pp 380–384
19. Prabhu C, Gandhi RV, Jain AK, Lalka VS, Thottempudi SG, Rao PP (2019) A novel approach to extend KM models with Object Knowledge Model (OKM) and Kafka for big data and semantic web with greater semantics. In: CISIS, Springer, Advances in Intelligent Systems and Computing, vol 993, pp 544–554
20. Ofenloch A, Greif F (2018) A flexible distributed simulation environment for Cyber-Physical Systems using ZeroMQ. J Commun 13(6):333–337
21. Chaisawat S, Vorakulpipat C (2021) Towards achieving personal privacy protection and data security on integrated E-Voting model of blockchain and message queue. Secur Commun Networks 2021:8338,616:1–8338,616:14
22. Hong XJ, Yang HS, Kim YH (2018) Performance analysis of RESTful API and RabbitMQ for microservice web application. In: ICTC. IEEE, pp 257–259
23. Estrada N, Astudillo H (2015) Comparing scalability of message queue system: ZeroMQ vs RabbitMQ. In: CLEI. IEEE, pp 1–6
24. Ionescu VM (2015) The analysis of the performance of RabbitMQ and ActiveMQ. In: RoEduNet. IEEE, pp 132–137
25. Rodríguez A, Kristensen LM, Rutle A (2021) Verification of the MQTT IoT protocol using property-specific CTL sweep-line algorithms. Trans Petri Nets Other Model Concurr 15:165–183
26. Rodríguez A, Kristensen L M, Rutle A (2019a) Formal modelling and incremental verification of the MQTT IoT protocol. Trans Petri Nets Other Model Concurr 14:126–145
27. Rodríguez A, Kristensen LM, Rutle A (2019b) On CTL model checking of the MQTT IoT protocol using the sweep-line method. In: PNSE@Petri Nets/ACSD, CEUR-WS.org, CEUR Workshop Proceedings, vol 2424, pp 57–72
28. Rodríguez A, Kristensen LM, Rutle A (2018) On modelling and validation of the MQTT IoT protocol for M2M communication. In: PNSE@Petri Nets/ACSD, CEUR-WS.org, CEUR Workshop Proceedings, vol 2138, pp 99–118
29. Xu J, Yin J, Zhu H, Xiao L (2021) Modeling and verifying producer-consumer communication in Kafka using CSP. In: ECBS. ACM, pp 9:1–9:10

30. Lin Q, Wang S, Zhan B, Gu B (2020) Modelling and verification of real-time publish and subscribe protocol using UPPAAL and simulink/stateflow. J Comput Sci Technol 35(6):1324–1342
31. Fei Y, Zhu H, Li X (2018) Modeling and verification of NLSR protocol using UPPAAL. In: TASE. IEEE Computer Society, pp 108–115
32. Sun M, Lu Y, Feng Y, Zhang Q, Liu S (2021) Modeling and verifying the CKB blockchain consensus protocol. Mathematics 9(22)
33. Kwon S, Son S, Choi Y, Lee J (2021) Protocol fuzzing to find security vulnerabilities of RabbitMQ. Concurr Comput Pract Exp 33(23)
34. Li H, Niu Y, Yi J, Li H (2018) Securing offline delivery services by using Kerberos authentication. IEEE Access 6:40,735–40,746
35. Xu C, Zhu H, Xie W (2017) Modeling and verifying identity authentication security of HDFS using CSP. In: APSEC. IEEE Computer Society, pp 259–268