Computer Science and Information Systems 20(1):277–306 https://doi.org/10.2298/CSIS210707057X

Formalization and Verification of Kafka Messaging Mechanism Using CSP

Junya Xu[1], Jiaqi Yin[2,⋆], Huibiao Zhu[1,⋆] and Lili Xiao[1]

[1] Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University, Shanghai, China
hbzhu@sei.ecnu.edu.cn
[2] Northwestern Polytechnical University, Xi'an, China
jqyin@nwpu.edu.cn

Abstract. Apache Kafka is an open source distributed messaging system based on the publish-subscribe model, which achieves low latency, high throughput and good load balancing. As a popular messaging system, the transmission of messages between applications is one of the core functions of Kafka. Therefore, the reliability and security of data in the process of message transmission in Kafka have become the focus of attention. The formal methods can analyze whether a model is highly credible. Therefore, it is significant to analyze Kafka messaging mechanism which describes the communication process and rules between each module entity in Kafka from the perspective of formal methods.

In this paper, we apply the process algebra CSP (Communicating Sequential Processes) and the model checking tool PAT (Process Analysis Toolkit) to analyze Kafka messaging mechanism. The results of verification show that the model caters for its specification and guarantees the reliability of messages in the normal communication process. Moreover, in order to further analyze the security of Kafka messaging mechanism, we add the intruder model and the authentication protocol Kerberos model and compare the verification results of Kafka messaging mechanism with or without the secure protocol Kerberos. The results show that the Kerberos protocol has improved the security of Kafka messaging mechanism in some aspects, but there are still some security loopholes.

Keywords: Distributed Messaging System, Kafka Messaging Mechanism, CSP, Formalization, Verification

1. Introduction

Since distributed messaging system provides an effcient and stable transmission channel for the realization of streaming calculation, data transmission and other functions, it has been widely used to capture and analyze large amounts of data in real time. A messaging system is responsible for the transmission of data among applications, and these applications only focus on the data rather than the datials of transmission. Data transmission in the communication process of distributed message system is based on a reliable message queue, which mainly has the following two modes: point-to-point and publish-subscribe [6,25]. In a point-to-point mode, a producer sends data to a queue and one or more consumers consume data from this queue in sequence. Each data can only be used once, i.e.,

⋆ Corresponding authors

when a consumer consumes a piece of data in this queue, this data is removed from the messaging queue. ActiveMQ, ZeroMQ and RabbitMQ [1,16] are well-known message queuing platforms. Unlike point-to-point, in a publish-subscribe mode, producers publish messages grouped into topics, while consumers can subscribe to one or more topics and consume all the data in these topic. In addition, the same piece of data can be consumed by multiple different groups of consumers, and the data will not be deleted immediately after consumption. Apache Kafka is a high-performance cross-language distributed messaging system based on publish-subscribe mode [8,24,26], which has been widely used by Internet companies such as Yahoo, Twitter, etc.

As a popular open source distributed messaging system, Kafka has the following advantages [8,26]. First, Kafka provides high throughput for both publishers and subscribers. It can produce about 250,000 messages (50 MB) per second and process 550,000 messages (110 MB) per second. Second, Kafka can be persisted. Messages are persisted to disks, so it can be used for bulk consumption, such as Extract-Transform-Load (ETL) and for real-time applications. At the same time, it prevents data loss by persisting data to disks and using replica mechanism. Third, it is easier for Kafka to expand outward. There are multiple producers, brokers and consumers, all of which are distributed. As a result, it can expand the machine without downtime. Fourth, the state in which messages are processed is maintained on the consumer side, not on the server side, so that it can be automatically balanced when message processing fails. Finally, Kafka supports both online and offline scenarios.

The formal methods are the research methods based on mathematical logic, which can verify and evaluate the reliability of the model through the specification, modeling and analysis of the model. Therefore, we consider to analyze whether the data in Kafka messaging mechanism is reliable from the perspective of formal methods. In this paper, we use the classical process algebra CSP [3,9,19] to give a formal model of Kafka messaging mechanism, and utilize the model checker PAT [10,13,17,20] to verify some important properties, including *Deadlock Freedom, Acknowledgement Mechanism, Parallelism, Sequentiality* and *Fault Tolerance*. Moreover, we introduce the intruder to simulate the attack behavior in the real network and introduce the authentication protocol Kerberos to improve the security of the Kafka messaging mechanism. We also model the intruder and Kerberos based on the original model to further analyze the security of Kafka.

The remainder of this paper is organized as follows. Section II gives a brief introduction to Kafka messaging system, the process algebra CSP and the model checking tool PAT. In section III, we model the core components in Kafka messaging system using CSP. At the same time, we adopt the model checking tool PAT to implement the constructed model and verify five properties. Moreover, we analyze the security of Kafka messaging mechanism by modeling the intruder and Kerberos and comparing the verification results of the constructed model with or without the Kerberos protocol in Section IV. Finally, we conclude this paper and make a discussion on the future work in Section V.

2. Background

In this section, we start with an overview of Kafka messaging system. At the same time, we also give a brief introduction to process algebra CSP and model checking tool PAT.

2.1. Kafka messaging system

In this paper, we focus on Kafka messaging mechanism which describes the process of communication among producers, consumers and other components as shown in Fig.1.

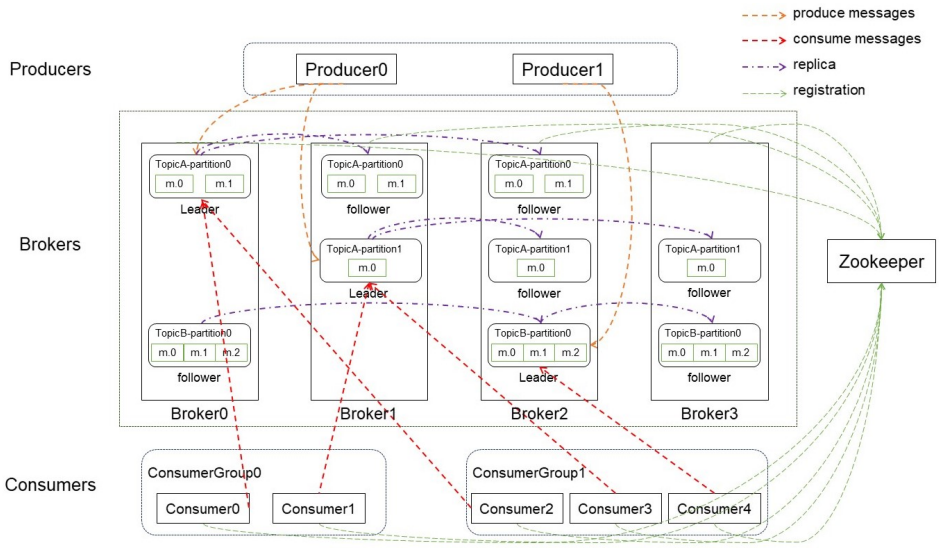


Fig. 1. The Kafka Messaging System

Before we introduce the Kafka messaging system, we first need to know a few common terms [16] in Kafka as follows:

- **ZooKeeper**: ZooKeeper helps Kafka store and manage the information of Kafka cluster.
- **Producer**: Producers consist of applications and are publishers of data, primarily sending messages to Brokers in Kafka.
- **Broker**: The Kafka messaging system contains one or more brokers that save data from producers and provide these data to consumers.
- **Topic**: Each message published to Kafka has a category called Topic. Producers and consumers only need to specify the topic of messages to publish and consume the data, regardless of which brokers data is stored on.
- **Partition**: A topic can be divided into multiple partitions, each of which is an ordered queue, and each message in a partition is assigned an ordered id. Kafka only guarantees that messages are sent to the consumer in the order of one partition, not the order of the whole of a topic.
- **Replica**: Replica is for backup to ensure that data is not lost when a broker in Kafka fails and Kafka continues to work. Each partition of a topic has several replicas including one leader and several followers.

- **Leader**: The leader is a 'master' replica of multiple replicas of each partition, and is the object on which the producer sends the data and the object on which the consumer consumes the data.
- **Follower**: The follower is the 'slave' replica of multiple replicas of each partition, and synchronizes data from the leader in real time to keep the data synchronized with the leader.
- **Consumer Group**: Each consumer group consists of multiple consumers subscribing to the same topic. In Kafka, the data of the same partition can only be consumed by one consumer within the group, but consumers in other groups still use this data.
- **Group Coordinator**: There is only one group coordinator for a consumer group. It needs to manage the load balancing among the consumers in this group, and sends partitioning strategy to all the consumers in that group.
- **Consumer**: It is the consumer of messages and pulls messages from the subscribed topic.

Based on the above concepts, we introduce Kafka messaging mechanism from the following three aspects: production of messages, rebalance of a consumer group and consumption of messages.

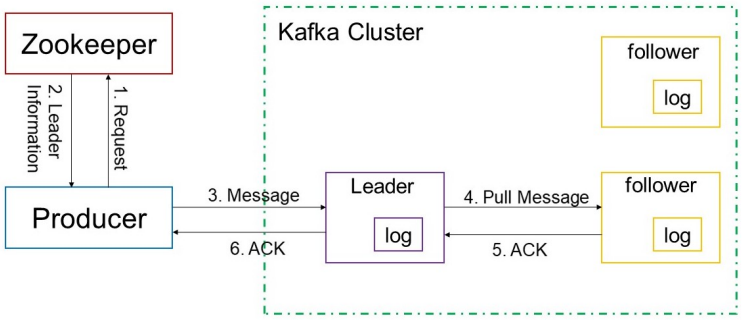


Fig. 2. The Production of Messages

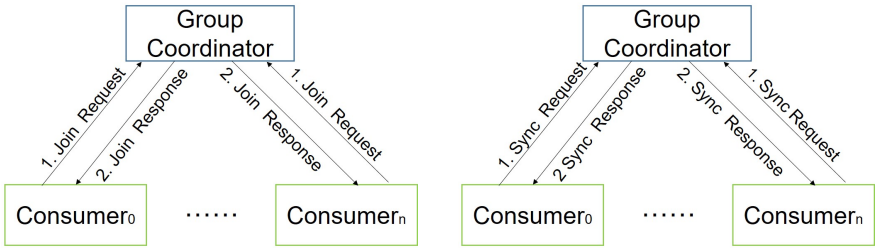
As shown in Fig.2, before publishing this message, a producer firstly needs to find the leader of a partition in this topic from ZooKeeper[18] and then sends the message to that leader. When a leader receives the message, it needs to write the message to a local log. Afterwards, followers of this partition need to pull this message from the leader and send 'ACK' to the leader after writing it to the local log. Finally, after receiving 'ACK' from all followers of this partition, the leader needs to send 'ACK' to the producer to state that the message has been delivered successfully.

In Kafka messaging mechanism, there are three types of values of 'ACK' sent by the leader in a partition to a producer: 'ACK=0', 'ACK=1' and 'ACK=all' [5]. In this paper, our model takes the third approach to ensure data security and reliability.

- 'ACK = 0' indicates that a producer does not care about the processing result of the message on the brokers. As long as it sends the message, it considers the message delivered successfully.

- 'ACK = 1' means that the message only needs to be written to the local log of the broker by the leader in this partition to return a successful commit.
- 'ACK = all' represents that before it is considered as a successful commit, the message not only needs to be stored by the leader, but also requires to be stored by all the followers of this leader .

The group coordinator of a consumer group need to manage all members in this group, and the process is called rebalance, which consists of two main steps: join and synchronization. In the process of consumers joining the group as shown in Fig.3 (a), all consumers in this group send 'JoinGroup Request' to the group coordinator and request to join. After receving all requests, the group coordinator selects one from members to play the role of leader and sends a reply message to each member of this group along with the member information of the cousumer group and the subscribed topic information to the leader. The leader needs to complete the partitioning strategy, which means that each consumer in this group should consume data from the corresponding partition of a topic. Fig.3 (b) shows the process of group synchronization. When the leader of consumers in this group completes the partitioning strategy, it encapsulates the strategy in the message named 'SyncGroup Request' to the group coordinator. Other consumers also send messages called 'SyncGroup Request'. After receiving all requests of synchronization, the group coordinator sends the partition strategy to all members of this group.



(a) Process of JoinGroup (b) Process of SyncGroup

Fig. 3. The Rebalance of a Consumer Group

Finally, we introduce the process by which consumers pull messages from the subscribed topic. In Kafka messaging system, consumers use the 'pull' mode to consume messages in sequence from the leader of the partition. Because the data transmission rate in the 'push' mode is determined by a server, but it is easy to cause problems like network congestion due to the lack of time for consumers to process the data.

It is worth noting that the message in the same partition of a topic can only be consumed by one consumer within the consumer group, but consumers in other consumer groups can still pull this messages. Moreover, when the number of partitions is greater than the number of consumers in a group, some of the consumers can pull messages from multiple partitions. In addition, we know that a topic can be divided into multiple partitions, and 'ordered' means that messages within each partition are sent to consumers in order, but there is no guarantee that messages within a topic are ordered.

2.2. CSP

Communicating Sequential Processes (CSP) [3,9] is the algebraic theory proposed by C.A.R. Hoare. The language is mainly designed to describe and analyze the behavior of concurrent systems and processes, which has been successfully applied in modeling and verifying various concurrent systems and protocols [7,11,27]. We give the following syntax of the CSP language used to describe the process in this paper, where P and Q are processes, a denotes the event and c represents the name of channel.

$$P, Q \ = \ Skip \mid Stop \mid a \rightarrow P \mid c?x \rightarrow P \mid c!x \rightarrow P \mid P \,\Box\, Q \mid$$
$$P \parallel Q \mid P \mid\mid\mid Q \mid P \lhd b \rhd Q \mid P; Q \mid P[\mid X \mid]Q$$

- $Skip$ represents that the process which does nothing but terminates successfully.
- $Stop$ denotes that the process does nothing and it is in the state of deadlock.
- $a \rightarrow P$ describes an object which first performs the event a and then behaves like P.
- $c?x \rightarrow P$ receives a message through channel c and stores the value in the variable x and then the behavior is like process P.
- $c!x \rightarrow P$ sends message x through channel c and then behaves like process P.
- $P \,\Box\, Q$ stands for the choice between process P and process Q, and this selection is decided by the environment.
- $P \parallel Q$ denotes that processes P and Q execute concurrently and are synchronized with the same communication events.
- $P \mid\mid\mid Q$ describes that processes P and Q run concurrently without barrier synchronization.
- $P \lhd b \rhd Q$ indicates if the Boolean condition b is true, the process behaves like P, otherwise like Q.
- $P; Q$ describes that processes P and Q execute in sequence.
- $P[\mid X \mid]Q$ denotes that the parallel composition of P and Q performs the concurrent events on the set X of channels.

2.3. PAT

Process Analysis Toolkit (PAT) [13,17], a toolset based on the process algebra CSP, is designed for applying model checking techniques for analysis of various systems and protocols. It supports to check for more properties [4,21], including deadlock freedom, reachability, complete LTL model checking, etc. Here we give some syntax of PAT used in this paper as follows.

- # $define\ V\ 0$
  It defines a global constant V with the initial value 0, and a global constant must be assigned an initial value in PAT.
- $var\ x\ =\ 1$
  It means that a variable x is defined with an initial value of 1. If the variable is not assigned an initial value, it defaults to 0.
- $channel\ c\ 0$
  This statement declares that c is the channel name and 0 is the buffer size. Notice that channel buffer size must be greater than or equal to 0. When the buffer size of a channel is equal to 0, it sends and receives messages synchronously.

- # $assert\ P\ deadlockfree$;
  This statement defines an assertion and it checks whether process P will enter a deadlock state or not.
- # $define\ goal\ x = false$;
  # $assert\ P\ reaches\ goal$;
  This first statement defines an assertion and the second statement checks whether process P will reach a state, where the property goal is satisfied or not.
- # $define\ goal\ x = false$;
  # $assert\ P \mid =\ goal$;
  This statement declares an assertion that checks whether process P always satisfies a state, where the property goal is satisfied or not.
- $|||i : \{0..N\}\ @P(i)$;
  This statement means that multiple processes run interspersed, specifically expressed as $P(0)$, $P(1)$, $P(2)$ ... $P(N)$.

3. Modeling

In this section, we use process algebra CSP to model the Kafka messaging mechanism which descirbes the process of communication between components in Kafka messaging system shown in Fig.1.

3.1. Sets, Messages and Channels

In order to model the process of message transmission and the behavior of components, such as producers and consumers, etc. In Kafka, we give the definitions of sets, messages and channels used in our model.

Table 1. The relationship between involved constants and pre-defined sets

Set	Constants
Module	Z(zookeeper), P(producer), PA(partition), F(follower), GC(groupcoordinator), C(consumer)
ID	TID(topic id), PID(producer id), LPAID(leader-partition id), FPAID(follower-partition id), CLeadID(leader-consumer id), CID(consumer id), GCID(GroupCoordinator ID)
Data	Data
Req	ReqData(request for data), reqTID(Request for the topic's id), reqLPAID(Request for the leader-partition's id), Join(request to join a group), Sync(request for group synchronization)
Ack	true/1(positive feedback), flase/0(negative feedback)

First, we give the definitions of some sets that are used in the model. **Module** set is composed of all modules in Kafka messaging system, including zookeeper, producers, consumers, groupcoordinators, leader-partitions and follower-partitions. **ID** set consists of unique identifier for each of the above module. **Req** set defines request information.

Table 2. The relationship between involved variables and pre-defined sets

Set	Constants
Module	z(zookeeper), p(producer), pa(partition), f(follower), gc(groupcoordinator), c(consumer)
ID	tid(topic id), lpaid(leader-partition id), fpaid(follower-partition id), cleadid(leader-consumer id), cid(consumer id), gcid(GroupCoordinator ID)
Data	data
Req	reqdata(request for data), reqtid(Request for the topic's id), reqlpaid(Request for the leader-partition's id), join(request to join a group), sync(request for group synchronization)
Ack	p_ack, c_ack, f_ack, sync_ack (positive feedback/negative feedback)

Data set includes the data transmitted between modules and **Ack** set contains feedback information.

In addition, we also give some constants based on the defined sets in Table I and some important variables we use in Table II respectively.

Based on the above sets, we describe the definition of the messages transmitted among components. In this paper, messages during communication are defined into the following three types:

$$\begin{aligned} MSG_{req} &= \{msg_{req}.A.B.content \mid A \in Module, B \in Module,\ content \in Req\} \\ MSG_{rep} &= \{msg_{rep}.A.B.content \mid A \in Module, B \in Module,\ content \in Ack\} \\ MSG_{data} &= \{msg_{data}.A.B.content \mid A \in Module, \\ &\qquad\qquad B \in Module,\ content \in \{Data,\ ID\}\} \end{aligned}$$

where, MSG_{req} is composed of the request messages transmitted between compents, MSG_{rep} represents the response messages and MSG_{data} means the data messages. A and B is sender and the receiver respectively, and $content$ represent content contained in each messagee.

Then, we define that MSG consists of the above three types of messages.

$$MSG = MSG_{req} \cup MSG_{rep} \cup MSG_{data}$$

Next, we define the channels used to simulate the communication among various modules. These channels of all modules use **COM_PATH** to represent in this paper:

- *ComZP*: the channels between zookeeper and producers. In a system, zookeeper may interact with multiple producers, and corresponding channels will also be generated. We use subscript p to distinguish each channel expressed as $ComZP_p$.
- *ComPL*: the channels between producers and leaders of the partitions. A produder can publish data on different topics to different leaders of topics by corresponding channels. We use the subscript i to distinguish each channel, which is described as $ComPL_i$.
- *ComLF*: the channels between followers and leaders of the partitions. A partition usually has one leader and multiple followers, so we use the subscript i and $ComLF_j$ to describe the multiple channels between leader and followers in a partition.

- *ComGC*: the channels between group coordinators and consumers. A consumer sends a request to the group coordinator, and the group coordinator publishes synchronization messages over this channel. Since a group coordinator manages multiple consumers within a consumper group, and there may be multiple consumper groups, we use the subscript m to distinguish them and denote them as $ComGC_m$.
- *ComLC*: the channels between consumers and leaders of the partitions. Each consumer can pull the data from mutiple leaders of partitions in different topics, and we use $ComLC_n$ to distinguish each channel.
- *ComFC*: the channels between consumers and followers of the partitions. When a broker on which the leader of a partition goes down, the consumer pulls data over the channel when one of followers becomes the leader of this partition. We use subscript l and $ComFC_j$ to describe the multiple channels.

3.2. Overall Modeling

According to the above description, the model of Kafka messaging mechanism includes six subprocesses, including $ZooKeeper$, $Producer$, $Consumer$, $GroupCoordinator$, $LPartition$ and $FPartition$. In order to facilitate the overall modeling and simulate the communication process of the model entity, we abstract the data transmission in Kafka. The overall model is shown in Fig.4.

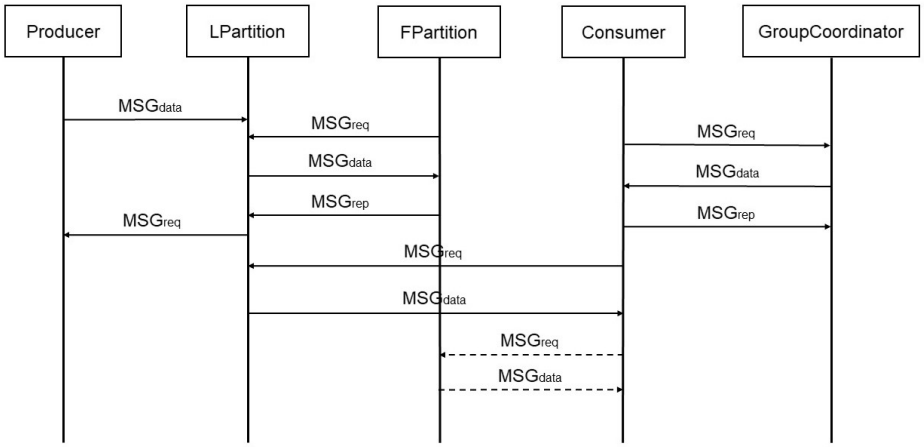


Fig. 4. The Communication Flow of Kafka Messaging Mechanism

Then, we formalize the whole model $System()$ as below:

$$
\begin{aligned}
System() =_{df}\ & |||_{\,pid\in PID,\ lpaid\in LPAID,\ fpaid\in FPAID,\ gcid\in GCID,\ cid\in CID} \\
& (\ ZooKeeper\ [|COM\_PATH|]\ Producer_{pid} \\
& \ [|COM\_PATH|]\ LPartition_{lpaid}\ [|COM\_PATH|]\ FPartitionr_{fpaid} \\
& \ [|COM\_PATH|]\ GroupCoordinator_{gcid}\ [|COM\_PATH|]\ Consumer_{cid}\ )
\end{aligned}
$$

where, the $System$ process is composed of the following processes: $ZooKeeper$, $Producer$, $LPartition$, $FPartition$, $GroupCoordinator$ and $Consumer$ concurrently using a set of channels COM_PATH. In addition, we define identifiers and a range of values for identifiers to distinguish each process. pid represents a producer's number and PID indicates the range of pid. Other characters, such as $lpaid$ and $fpaid$, have similar meanings. $[|COM\_PATH|]$ is the communication channel.

3.3. ZooKeeper

In Kafka, messages from the same Topic are divided into partitions and distributed over multiple brokers, and zookeeper needs to maintain a relationship between the partitions and the brokers. After receiving the request message from $Producer_{pid}$, $ZooKeeper$ process sends the information of these partitions to $Producer_{pid}$ by the channel $ComZP_p$. Each channel has its own id to prevent multiple processes of the same type from competing for resources on the same channel.

$$\begin{aligned} ZooKeeper() =_{df}\ & ComZP_p?msg_{req}.pid.reqtopid.reqlpaid \rightarrow \\ & ComZP_p!msg_{data}.PID.TID.LPAID \rightarrow \quad ZooKeeper() \end{aligned}$$

3.4. Producer

Producers, responsible for publishing data to partitions in a specified topic, are the important parts of Kafka messaging system. $Producer_{pid}$ process needs to find the leader of each partition from $ZooKeeper$ on the channel $ComZP_p$. At the same time, $Producer_{pid}$ provides a core parameter 'ack' to define the conditions for the message to be 'submitted'. In our model, it requires that the message has been stored not only by the leader-partition, but also by all follower-partitions of this leader.

$$\begin{aligned} Producer_{pid}() =_{df}\ & ComZP_p!msg_{req}.PID.reqTID.reqLPAID \rightarrow \\ & ComZP_p?msg_{data}.pid.tid.lpaid \rightarrow \\ & \begin{pmatrix} ComPL_i!msg_{data}.PID.LPAID.Data \\ \Box\ ComPL_i?msg_{rep}.p\_ack \end{pmatrix};\ Producer_{pid}() \end{aligned}$$

where, p_ack is a variable to describe a response message recevied from $LPartition_{lpaid}$. When its content is $P\_ack[lpaid] = 1$, it means that the data was successfully stored to the broker by all the replicas in this partition numbered $lpaid$.

3.5. LPartition

Each topic are divided into multiple partitions, but each partition has only one leader. As an important component of data storage, it needs to communicate with producers, consumers and followers of this partition. There are three types of channels that need to be used in this process: $ComPL_i$, $ComLC_n$, and $ComLF_j$.

First, it needs to accept the message from $Producer_{pid}$ process and then stores the message. Second, it needs to send the message which $Consumer_{cid}$ process needs. Finally, it also needs to send the message to $FPartition_{fpaid}$ of this partition to complete the copy of the message.

$$
\begin{aligned}
&LPartition_{lpaid}() =_{df} \\
&\left(\begin{array}{l}
\left(\begin{array}{l}
ComPL_i?msg_{data}.pid.lpaid.data \rightarrow \\
ComLF_j?msg_{req}.fpaid.tid.lpaid \rightarrow \\
ComLF_j!msg_{data}.Data \rightarrow \\
ComLF_j?msg_{rep}.fpaid.f\_ack \rightarrow \\
GetStateF(tid, lpaid, fpaid); \\
\left(\begin{array}{l}
(ComPL_i!msg_{rep}.P\_ack[lpaid]\ \{P\_ack[lpaid] = 1\}) \\
\lhd F\_ack == true\ \&\&\ lpaid \in LPAID \rhd\ SKIP
\end{array}\right)
\end{array}\right) \\
\Box \left(\begin{array}{l}
ComLC_n?msg_{req}.reqdata\ \{DataS[lpaid][cid][seq] = 1; \\
\qquad seq = Seq[lpaid];\ Seq[lpaid]++\} \rightarrow \\
ComLC_n!msg_{data}.Data
\end{array}\right)
\end{array}\right); \\
&LPartition_{lpaid}()
\end{aligned}
$$

In the above formula, $GetStateF(tid, lpaid, fpaid)$ is used to get the status of all followers to check whether they have all stored data; $P\_ack[lpaid] = 1$ indicates that the data is stored not only by the leader, but also by all followers of this partition numbered $lpaid$; $DataS[lpaid][cid][seq]$ records the state of the data with a sequence number of seq; and $Seq[lpaid]$ is the order of the data in $LPartition_{lpaid}$.

3.6. FPartition

The followers of each partition plays an important role in data security and data reliability which supports a copy mechanism. In detail, when process $LPartition_{lpaid}$ receives a message from $Producer_{pid}$, all processes $FPartition$ of this partition need to pull the message and store it. This is to ensure that when the broker of on which $LPartition_{lpaid}$ is located fails, $FPartition_{fpaid}$ of this partition can communicate with $Consumer_{cid}$ instead of it.

$$
\begin{aligned}
&FPartition_{fpaid}() =_{df} \\
&\left(\begin{array}{l}
\left(\begin{array}{l}
ComLF_j!msg_{req}.FPAID.TID.LPAID \rightarrow \\
ComLF_j?msg_{data}.data\{stateF[tid][lpaid][fpaid] = 1\} \rightarrow \\
ComLF_j!msg_{rep}.FPAID.F\_ack
\end{array}\right) \\
\Box \left(\begin{array}{l}
ComFC_l?msg_{req}.reqdata \rightarrow \\
ComFC_l!msg_{data}.Data
\end{array}\right)
\end{array}\right); \\
&FPartition_{fpaid}()
\end{aligned}
$$

where, the array $stateF[tid][lpaid][fpaid]$ indicates the state of $FPartition_{fpaid}$ of the $Lpartition_{lpaid}$ in $topic_{tip}$, i.e., whether data is received. After receiving the data successfully, the value of $stateF[tid][lpaid][fpaid]$ will change to 1.

3.7. GroupCoordinator

The group coordinator is responsible for managing all members of this group. After all consumers in this group making requests, $GroupCoordinator_{gcid}$ process selects a con-

sumer to take a leadership role and sends it group membership information and subscription information. In addition, process $GroupCoordinator_{gcid}$ notifies each $consumer_{cid}$ in the group of the partitioning strategy developed by the leader of consumers.

$$
\begin{aligned}
GroupCoordinator_{gcid}() =_{df}\ & ComGC_m?msg_{req}.join \rightarrow GetStateC(cid); \\
& \big(ConsumerL[cid] = 1 \lhd C\_ack == true \rhd SKIP\big); \\
& \left(\begin{array}{l}
\left(\begin{array}{l} ComGC_m!msg_{data}.Join.CLeadID.CID.TID.LPAID \rightarrow \\ ComGC_m?msg_{req}.sync.cid.tid.lpaid \rightarrow \\ ComGC_m!msg_{rep}.Sync\_Ack \end{array}\right) \\
\quad \lhd ConsumerL[cid] == 1 \rhd \\
\left(\begin{array}{l} ComGC_m!msg_{data}.Join.CLeadID.CID \rightarrow \\ ComGC_m?msg_{req}.sync \rightarrow ComGC_i!msg_{rep}.Sync\_Ack \end{array}\right)
\end{array}\right) \\
& ; GroupCoordinate_{gcid}()
\end{aligned}
$$

where, $ConsumerL[cid] = 1$ indicates that the consumer whose number is cid takes a leadership role in this consumer group and is responsible for the assignment of consumer and partition.

3.8. Consumer

Consumers is the core part in data consumption. First, it needs to join a consumer group by sending a request to $GroupCoordinator_{gcid}$. After joining and synchronizing the consumer group, it can pull messages from the assigned $LPartition$ according to the partitioning strategy. In addition, the $Consumer_{cid}$ can also pull messages from the follower-partitions of $LPartition$ in order to ensure that after the leader-partition crashes, consumers will still have access to the information they need.

$$
\begin{aligned}
& Consumer_{cid}() =_{df} ComGC_i!msg_{req}.Join \rightarrow \\
& \left(\begin{array}{l}
\left(\begin{array}{l} ComGC_m?msg_{data}.join.cleadid.cid.tid.lpaid\{consumer[lpaid][cid] = 1\} \\ \rightarrow ComGC_m!msg_{req}.Sync.CID.TID.LPAID \\ \rightarrow ComGC_m?msg_{rep}.sync\_ack\{consumerS[cid] = 1\} \end{array}\right) \\
\Box \left(\begin{array}{l} ComGC_m?msg_{data}.join.cleadid.cid \rightarrow ComGC_m!msg_{req}.Sync \\ \rightarrow ComGC_m?msg_{rep}.sync\_ack\{consumerS[cid] = 1\} \end{array}\right)
\end{array}\right) \\
& ; GetSync(cid); \\
& \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l} ComLC_n!msg_{req}.ReqData \rightarrow \\ ComLC_n?msg_{data}.data\{Data[lpaid][cid] = 1\} \end{array}\right) \\
\Box \left(\begin{array}{l} ComFC_l!msg_{req}.ReqData \rightarrow \\ ComFC_l?msg_{data}.data\{DataF[lpaid][fpaid][cid] = 1\} \end{array}\right)
\end{array}\right) \\
\lhd S\_Ack == true \ \&\& \ consumer[lpaid][cid] == 1 \rhd \\
SKIP
\end{array}\right) \\
& ; Consumer_{cid}()
\end{aligned}
$$

In the above formula, $consumer[lpaid][cid] = 1$ represents that $consumer_{cid}$ establishes a connection to $LPartition_{lpaid}$, that is, this consumer can pull messages from

the leader of partition numbered $lpaid$. $consumerS[cid]$ indicates whether $consumer_{cid}$ completes the group synchronization. $GetSync(cid)$ is a function to get the state of process $consumer_{cid}$ synchronization. $Data[lpaid][cid]$ indicates the transmission of data between $Consumer_{cid}$ and $LPartition_{lpaid}$, where $Data[lpaid][cid] = 1$ indicates success. In addition, we use $DataF[lpaid][fpaid][cid]$ to define whether $consumer_{cid}$ can pull data from $FPartition_{fpaid}$ of $LPartition_{lpaid}$.

4. Architecture Verification

In order to evaluate the correctness and reliability of Kafka in the normal communication process, we use the model checking tool PAT to verify the properties of the constructed formal model, including *Deadlock Freedom, Acknowledgement Mechanism, Parallelism, Sequentiality* and *Fault Tolerance*. Here, we give the detailed verification procedure:

4.1. Deadlock Freedom

We need to ensure that each process in the system we build can communicate and interact with each other smoothly, and that the whole system does not stop due to one process get into a deadlock state. PAT provides a primitive assertion to describe this situation:

$$\#assert\ System()\ deadlockfree;$$

The *Deadlock Freedom* property is used to verify whether our system is in the deadlock state.

4.2. Acknowledgement Mechanism

In order to avoid losing data, there is an ack mechanism designed to describe a scenario that followers copy data from the corresponding leader-partition in Kafka. Data reliability is important for data storage, thus we give the definition of the property and the assertion:

$$\#define\ Acknowledgement\_Mechanism\ \ (\ P\_ack[1] == 1\ \&\&\ P\_ack[2] == 1\ );$$
$$\#assert\ System()\ reaches\ \ Acknowledgement\_Mechanism;$$

If all the final values of the variable $P\_ack[lpaid]$ are changed from 0 to 1, we will say that the property *Acknowledgement Mechanism* is satisfied.

4.3. Parallelism

According to the partitioning strategy in our model, a partition of the same topic can only send data to one consumer of the same consumer group. In addition, we should ensure that when a consumer pulls data from the corresponding partitions, it will not affect the data of other consumers. Then we define the assertion as follows:

$$\begin{aligned}&\#define\ Parallelism\ (\ Data[0][0] == 1\ \&\&Data[0][1] == 0\\ &\qquad\qquad\qquad\qquad \&\&\ Data[1][0] == 0\ \&\&\ Data[1][1] == 1\ );\\ &\#assert\ System()\ reaches\ Parallelism;\end{aligned}$$

If our system satisfies the property *Parallelism*, then according to the partitioning strategy, $consumer_{cid}$ can connect to the corresponding channels of $partition_{lpaid}$ in a topic respectively.

4.4. Sequentiality

In Kafka, the data pulled by consumers and sent by corresponding partitions are all in order. In the above implementation section, we adopts $DataS[LPA][C][Seq]$ to represent whether the data is sent, where if its value is equal to 1, it means the data sent successfully, otherwise, not. The definitions and assertion are as follows:

$$
\begin{aligned}
&\#define\ channel1\_Seq \\
&\quad ((DataS[0][0][0] == 1 \&\& DataS[0][0][1] == 0) \\
&\quad || (DataS[0][0][0] == 1 \&\& DataS[0][0][1] == 1) \\
&\quad || (DataS[0][0][0] == 0 \&\& DataS[0][0][1] == 0)); \\
&\#define\ channel2\_Seq \\
&\quad ((DataS[1][1][0] == 1 \&\& DataS[1][1][1] == 0) \\
&\quad || (DataS[1][1][0] == 1 \&\& DataS[1][1][1] == 1) \\
&\quad || (DataS[1][1][0] == 0 \&\& DataS[1][1][1] == 0)); \\
&\#define\ Sequentiality\ (channel1\_Seq\ \&\&\ channel2\_Seq); \\
&\#assert\ System() \mid =\ Sequentiality;
\end{aligned}
$$

Since we randomly set the number of data to 2, there are three cases where the system satisfies this property. The first shows none of the messages are sent, the second indicates that the previous message is sent and the subsequent message is not sent, and the third expresses that all the messages are sent successfully.

4.5. Fault Tolerance

The *Fault Tolerance* property describes that a consumer still gets the needed message when a partition taking a leadership role breaks down. The replica mechanism enables Kafka messaging system to own this property, and the assertion is defined as follow:

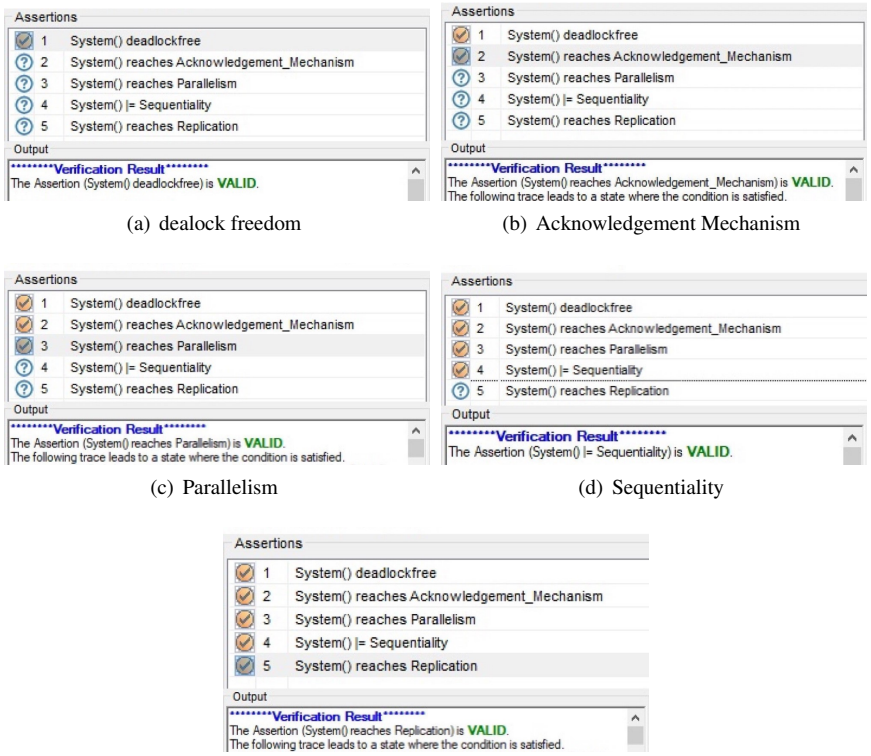
$$
\begin{aligned}
&\#define\ Replication \\
&\quad (DataF[0][0][0] == 1\ \&\&\ DataF[0][1][0] == 1 \\
&\quad \&\&\ DataF[1][0][1] == 1\ \&\&\ DataF[1][1][1] == 1); \\
&\#assert\ System()\ reaches\ Replication;
\end{aligned}
$$

Based on the partitioning strategy assumed in this paper, if processes $consumer_0$ and $consumer_1$ can respectively pull data from the followers of $partition_0$ and $partition_1$, it means the property *Fault Tolerance* is satisfied.

4.6. Verification and Results

Based on the above definitions and assertions, we implement the code in PAT and it searches the state space of the system until it finds a counter example or runs out of state space. At the end, we get the results of verification shown in Fig.5.

From Fig.5, we can see that the five properties are all valid, which means the pattern of the distributed messaging system can guarantee the correctness and reliability of communications.



(a) dealock freedom (b) Acknowledgement Mechanism

(c) Parallelism (d) Sequentiality

(e) Fault Tolerance

Fig. 5. Verification Results in $System()$

- The propertry *Deadlock Freedom* means that our model does not run into a deadlock state.
- The propertry *Acknowledgement Mechanism* represents that each replica stores the message published by the producer to ensure the reliability of the message delivery.
- The propertry *Parallelism* ensures that there is no interference between consumers within the same consumer group.
- The propertry *Sequentiality* indicates that the data consumption of each partition in Kafka is orderly.
- The propertry *Fault Tolerance* describes that Kafka messaging system is robust and does not crash even if the leader of a partition fails.

5. Security Verification

In this section, we introduce the intruder model to judge whether the messaging mechanism can guarantee its reliability and security. Meanwhile, we introduce the Kerberos protocol to improve the security of data transmission and further analyze Kafka messaging mechanism by comparing the verification results.

In an insecure network environment, some intruders may attack the process of data transmission, resulting in data leakage and other problems, which are described:

- **Camouflage**: An intruder can either send bogus messages to a broker by pretending to be a producer, or send request for consuming data to a broker pretend to be a consumer. The broker is unable to determine whether the entity sending messages has a legitimate identity, and will store the data recevied from the intruder disguised as the producer as the normal data, or will transmit the data stored on the broker to the intruder disguised as the consumer, resulting in data inauthenticity or data leakage and other problems.
- **Interception**: When a producer transmits a message containing real data to the broker, an intruder can intercept the message and discard it or tamper with it so that the broker does not receive the real message. In other case, when the broker sends data to a consumer, an intruder can also intercept the message so that the real consumer cannot receive the message and is in a waiting state, and the real data is stolen by the intruder, resulting in data leakage.

5.1. Intruder

We also deem an intruder as a process that can pretend to be a producer or a consumer to intercept messages on channels *ComPL* and *ComCL*, as well as to use fake channels to send bogus or tampered messages to the broker. Here, we introduce these channels that an intruder might use:

$$INTER\_PATH =_{df} FakePL \cup InterceptPL \cup FakeCL \cup InterceptCL \cup InterceptCF$$

Then, we define the set $Fact$, which represents the fact that an intruder might acquire:

$$Fact =_{df} Producer \cup Consumer \cup MSG$$

Next, we define the rule to express how the intruder can deduce new facts from what it has known, shown as follows:

$$F \mapsto f \wedge F \subseteq F' \Rightarrow F' \mapsto f$$

where, set F denotes the facts the intruder has known, and f is the fact deduced from set F. $F \mapsto f$ represents that fact f can be deduced from the set F.

Also, we define the funtion $Info$, which indicates how an intruder obtains a new fact from an already obtained message:

$$Info(msg.A.B.content) =_{df} \{ A.B.content \}$$

In addition, we declare a channel $Deduce$ used for deducing new facts:

$$Channel\ Deduce :\ Fact.P(Fact)$$

Based on the above description, an intruder can eavesdrop and intercept message transmitted between processes on the normal channels to obtain new facts, and can also

interfere with the communication by sending false messages. We first present a model of an intruder masquerading as a producer process:

$$
\begin{aligned}
&FakePro(F) =_{df} \ \Box_{m\in MSG} InterceptPL!m \rightarrow FakePro(F \cup Info(m)) \\
&\quad \Box\Box_{m\in MSG\cap Info(m)\subset F} FakePL!m \rightarrow FakePro(F) \\
&\quad \Box\Box_{f\in Fact, f\notin F, F\mapsto f} Init\{datac\_leakage = flase\} \rightarrow Deduce.f.F \rightarrow \\
&\left(\begin{array}{l}(DataP\_Leaking\_Success\{datap\_leakage = true\} \rightarrow FakePro(F \cup \{f\})) \\ \lhd\ f == Data \ \rhd \\ (DataP\_Leaking\_Success\{datap\_leakage = false\} \rightarrow FakePro(F \cup \{f\}))\end{array}\right)
\end{aligned}
$$

Similarly, we also present an intruder model masquerading as a consumer:

$$
\begin{aligned}
&FakeCon(F) =_{df} \ \Box_{m\in MSG} InterceptCL!m \rightarrow FakeCon(F \cup Info(m)) \\
&\quad \Box\Box_{m\in MSG\cap Info(m)\subset F} FakeCL!m \rightarrow FakeCon(F) \\
&\quad \Box\Box_{f\in Fact, f\notin F, F\mapsto f} Init\{datac\_leakage = flase\} \rightarrow Deduce.f.F \rightarrow \\
&\left(\begin{array}{l}(DataC\_Leaking\_Success\{datac\_leakage = true\} \rightarrow FakeCon(F \cup \{f\})) \\ \lhd\ f == Data \ \rhd \\ (DataC\_Leaking\_Success\{datac\_leakage = false\} \rightarrow FakeCon(F \cup \{f\}))\end{array}\right)
\end{aligned}
$$

5.2. Updated Model

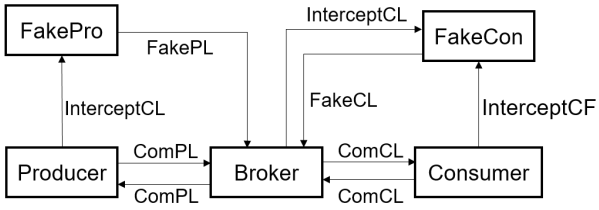


Fig. 6. Channels of Kafka Messaging Mechanism with Intruders

After modeling the intruder, we consider adding intruders to the existing system model as shown in Fig.6. In this system model, we only extract one producer process, one broker process and one consumer process. Therefore, we need to add the intruder process and the channels used on the basis of the original process interaction, so that the intruder can complete the communication interaction with the normal process.

$$
\begin{aligned}
&System\_I =_{df} \ System\_FakingP \ || \ System\_FakingC \\
&System\_FakingP =_{df} \ (\ Producer^{'} \ [|COM\_PATH|] \ Broker^{'} \\
&\qquad [|COM\_PATH|] \ Consumer^{'} \ [|INTR\_PATH|] \ FakePro\ ) \\
&System\_FakingC =_{df} \ (\ Producer^{'} \ [|COM\_PATH|] \ Broker^{'} \\
&\qquad [|COM\_PATH|] \ Consumer^{'} \ [|INTR\_PATH|] \ FakeCon\ )
\end{aligned}
$$

Updated Producer Next, we need to update the $Producer$ process so that an intruder can send fake data to a broker, and intercept data sent by a producer to a broker. Therefore, we need to add channels $FakePL$ and $InterceptPL$ to replace the original normal communication channel. We use the rename operation in CSP to update the communication channels, where $\{|c|\}$ describes the set of all events that occur on channel c:

$$\begin{aligned} Producer'() =_{df}\ & Producer()[[\\ & ComPL?\{|ComPL|\} \leftarrow ComPL?\{|ComPL|\}, \\ & ComPL!\{|ComPL|\} \leftarrow ComPL!\{|ComPL|\}, \\ & ComPL!\{|ComPL|\} \leftarrow InterceptPL!\{|ComPL|\}]] \end{aligned}$$

Updated Broker Then, we need to update the $Broker$ process so that an intruder can transmit messages to a broker by disguising a producer, or it can also transmit the message for requesting data to a broker by disguising a consumer and steal the data transmitted by a broker. We added the renamed channel to update the communication channel to complete the communication behavior of the intruder.

$$\begin{aligned} Broker'() =_{df}\ & Broker()[[\\ & ComPL?\{|ComPL|\} \leftarrow ComPL?\{|ComPL|\}, \\ & ComPL?\{|ComPL|\} \leftarrow FakePL?\{|ComPL|\}, \\ & ComPL!\{|ComPL|\} \leftarrow ComPL!\{|ComPL|\}, \\ & ComCL?\{|ComCL|\} \leftarrow ComCL?\{|ComCL|\}, \\ & ComCL?\{|ComCL|\} \leftarrow InterceptCL?\{|ComCL|\}, \\ & ComCL!\{|ComCL|\} \leftarrow ComCL!\{|ComCL|\}, \\ & ComCL!\{|ComCL|\} \leftarrow FakeCL!\{|ComCL|\}]] \end{aligned}$$

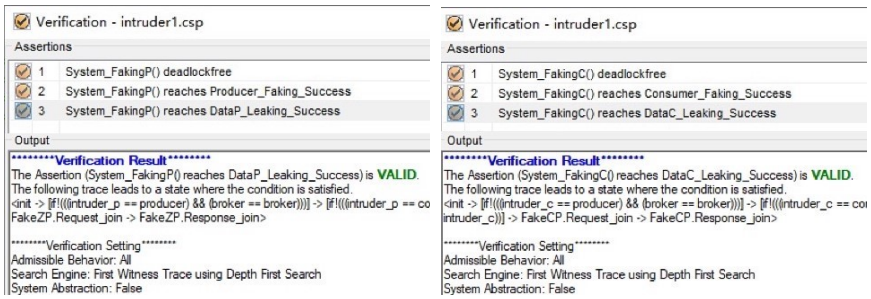
Updated Consumer Similarly, an intruder can disguise as a real consumer to intercept and tamper with a request message from a consumer over the normal channel to a broker. The updated $Consumer$ process describes as follow:

$$\begin{aligned} Consumer'() =_{df}\ & Consumer()[[\\ & ComCL?\{|ComCL|\} \leftarrow ComCL?\{|ComCL|\}, \\ & ComCL!\{|ComCL|\} \leftarrow ComCL!\{|ComCL|\}, \\ & ComCL!\{|ComCL|\} \leftarrow InterceptCF!\{|ComCL|\}]] \end{aligned}$$

5.3. Verification Results of Model with Intruder

We implement the updated model and the intruder model in PAT and get the results of verification shown in Fig.7.

From Fig.7 (a), we can see the three properties are all valid, which means that an intruder can disguise a producer and successfully intercept the data message transmitted by



(a) Camouflage Producer (b) Camouflage Consumer

Fig. 7. Verification Results in $System\_I()$

a real producer in the normal transmission process, thus causing the data leakage problem. Similarly, we see the results shown in Fig.7 (b) that three properties are all valid, which means that an intruder can disguise a consumer to intercept the data transmitted by a broker to a consumer, thus leading to the problem of data leakage.

5.4. Kerberos

After the 0.9.0.0 version of Kafka messaging system, security mechanism was introduced. This paper analyzes the security of Kafka messaging mechanism by adding the modeling of Kerberos authentication protocol and comparing the validation results of the messaging mechanism model with no security mechanism.

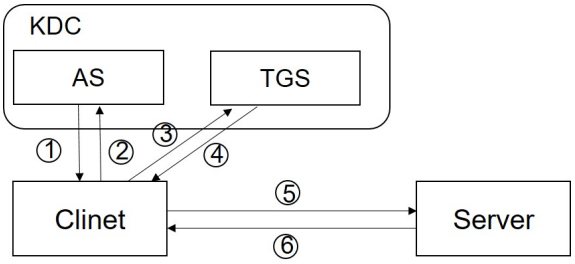


Fig. 8. Channels of Kafka Messaging Mechanism with Intruders

Kerberos [2,22] is a protocol based on key encryption technology by MIT, which provides authentication of identity for applications of client and server. There are three main modules: client, server and key distribution center(KDC). KDC is composed of authentication server (AS) and ticket granting server (TGS). Fig.8 describes the authentication process of Kerberos protocol in detail.

– In the first step, the client needs to prove his identity to the AS in the KDC.

- In the second step, AS determines whether the user's identity is valid. If it is valid, AS will generate the session key between the client and TGT[3]. Then it encrypts the session key and TGT with the client's public key and transmits it to the client.
- Third, after receving the encrypted message transmitted by AS, the client decrypts the ciphertext message with its private key to obtain TGT and session key. Then, the client transmits TGT to TGS together with Auth[4].
- Step 4, Similarly, TGS uses its private key to decrypt TGT to obtain the information of the client and session key between the client and TGS, and uses the session key to decrypt Auth to obtain the identity information provided by the client. At this time, TGS checks whether the timestamp is expired and verifies the consistency of the client's information by comparison, as well as encrypts the session key between the client and the server. When the information is verified successfully, TGS returns the encrypted session key and ST[5].
- Step 5, the client decrypts the message with the session key between the client and TGS to obtain the session key of the client and the server, and uses this session key to encrypt Auth$'$[6]. Then the ciphertext message ST and Auth$'$ are sent to the server.
- The sixth step, the server obtains the identity information by decrypting ST with its private key and the identity information by decrypting Auth$'$ with the session between the client and the server, respectively. After the verification is passed, the server uses the session key between the client and the server to encrypt the server's identity information and timestamp, and transmits the ciphertext message to the client. Finally, the client determines whether the server identity is consistent. If consistent, the connection is successful, that is, the client can access the server.

Kerberos Model First, we need to add some definitions for other sets: the set **Key** contains the long-term key **MK** and the short-term key **SK**, the set **Time** contains the definition of discrete time.

In addition, we also defined the encryption function **E** and decryption function **D**, which are specifically expressed as follows:

$$E(k, msg);\ D(k, emsg)$$

Where, function **E** uses key **k** to encrypt the message **msg**, which is represented as **emsg**, and function **D** uses key **k** to decrypt the encrypted message **emsg**. Therefore, we can draw the following conclusion:

$$D(k, E(k, msg)) = msg$$

Next, we define some of the channels used by the processes when modeling the Kerberos protocol, as follows:

$$Time,\ ComUA,\ ComTA,\ ComSA$$

[3] TGT: the ticket provided to TGS when applying for the server ticket encrypted the public key of TGS, containing the client's identity, time stamp and the session key between the user and TGS.

[4] Auth: authentication information of the client encrypted by the session key between the client and TGS, including user name, timestamp.

[5] ST: the ticket to access the server encrypted with the server's public key, containing the client's identity, timestamp, and session key between the server and the client.

[6] Auth$'$: authentication information of the client encrypted by the session key between the client and the server

Based on the definitions of sets, functions and channels, we model the Kerberos protocol. First, we need to define a process $Clock$ to synchronize time of the whole system. $Clock$ process represents a discrete increase in time t, and when receiving a request message, it responds with a message containing the current time t.

$$Clock() =_{df} tick -> \{t = t + 1\} -> Clock() \\ \Box\, time\,?\, request -> time\,!\, t -> Clock();$$

To facilitate the modeling of the Kerberos protocol, we will model steps 1-6 in Fig.8 respectively. The Step 1 and Step 2 in Fig.8 are the interactions between the client and AS. Process $USER\_AS$ needs to send a request to process $Clock$ and takes the returned time t as the initial time. Next, process UER_AS sends a message including its name and initial time to AS and expects a reply from AS. If the authentication fails, process UER_AS will stops. If the anthentication is successful, process UER_AS will decrypt the message with its private key MK_{user}^{-1} to get TGT and session key between TGS and client SK_{TGS}.

$$USER\_AS(user) =_{df} time\,!\, Request \rightarrow time\,?\, t\,\{\, starttime = t\,\}\,; \\ ComUA\,!\, name.starttime\{name = user\} \rightarrow \\ \left(\begin{array}{l} ComUA\,?\, name\_fail \rightarrow Stop \\ \Box \left(\begin{array}{l} ComUA\,?\, x \\ \{\, TGT = D(MK_{user}^{-1}, x);\ SK_{TGS} = D(MK_{user}^{-1}, x)\} \rightarrow Skip \end{array} \right) \end{array} \right)$$

In the communication between the client and AS, process AS will judge whether the username is valid after receiving the message sent by process UER_AS. If not, a failed reponse is returned to UER_AS. If successful, AS generates SK_{TGS}, as well as uses the client's public key MK_{user} to encrypt SK_{TGS} and TGT, and sends the encrypted message to UER_AS.

$$AS() =_{df} ComUA?\, x\{name = getname(x);\ name\_faking = \neg(valid(name))\}\,; \\ \left(\begin{array}{l} ComUA\,!\, Name\_Fail \rightarrow Skip \\ \lhd\, name\_invalid == true \,\rhd \\ \left(\begin{array}{l} ComUA\,!\, E(MK_{user},\ (TGT,\ SK_{TGS})) \\ \{TGT = E(MK_{TGS}, (username, starttime, lifetime, SK_{TGS}))\} \rightarrow Skip \end{array} \right) \end{array} \right);$$

In the above formula, the function $getname()$ is used to get the client's name, and the function $valid(name)$ is used to determine whether the name is valid.

Steps 3 and 4 in Fig.8 are the interactions between the client and TGS. $USER\_TGS$ process sends the TGT and $Auth$ to process TGS, and expects to receive a reply from TGS. The first case is when the process stops because the ticket expires due to a timeout. In the second case, when the identity information in TGT does not match the identity information $Auth$, the process terminates. The last one is that the authentication succeeds, and process $USER\_TGS$ receives ST and the session key SK_{Server} between the client and the server.

$$USER\_TGS(user, server) =_{df} \; ComTA\,!\,TGT.Auth$$
$$\{Auth = E(SK_{TGS}, (username, starttime, lifetime))\} \rightarrow$$
$$\left(\begin{array}{l} ComTA\,?\,Timeout \rightarrow Stop \\ \Box \left(\begin{array}{l} ComTA\,?\,invaild \rightarrow Stop \\ \Box(ComTA\,?\,y\,\{\, SK_{Server} = D(SK_{TGS}, y)\} \rightarrow Skip\,) \end{array} \right) \end{array} \right)$$

After receiving the message from process $USER\_TGS$, process TGS first sends a request message to process $Clock$ and obtains the time t of the current system. Then, it uses the private key MK_{TGS}^{-1} to decrypt TGT to get SK_{TGS} as well as the client's identity information, initial time and life time provided by AS. Next, TGS decrypts $Auth$ transmitted by process $USER\_TGS$ with SK_{TGS}, obtains the client's information, initial time and life time given by client, and judges whether the timeout is determined by comparing the life time of the ticket with the current time. If the time runs out, TGS will directly send the message named $Timeout$ to $USER\_TGS$. If the ticket is within the valid time, TGS compares the information provided by AS and the client, and an invalid reply message is sent if it does not match. If it is consistent, TGS generates ST and Sk_{Server} encrypted with SK_{TGS}, and sends them to process $USER\_TGS$.

$$TGS() =_{df} \; dComTA\,?\,y \rightarrow time\,!\,Request \rightarrow time\,?\,t\,\{\, nowtime = t\,\} \rightarrow$$
$$TGT = get(y) \rightarrow msg_{TGT} = D(MK_{TGS}^{-1},\, TGT) \rightarrow$$
$$SK_{TGS} = get(msg_{TGT}) \rightarrow a = getname(msg_{TGT}) \rightarrow$$
$$starttime_t = get(msg_{TGT}) \rightarrow lifetime_t = get(msg_{TGT}) \rightarrow$$
$$Auth = get(y) \rightarrow msg_{Auth} = D(SK_{TGS},\, Auth) \rightarrow b = getname(msg_{Auth}) \rightarrow$$
$$starttime_a = get(msg_{Auth}) \rightarrow lifetime_a = get(msg_{Auth}) \rightarrow$$
$$\left(\begin{array}{l} ComTA\,!\,Timeout \rightarrow Stop \\ \lhd\, nowtime - starttime_t > lifetime_t \,||\, nowtime - starttime_a > lifetime_a \rhd \\ \left(\begin{array}{l} ComTA\,!\,invaild \rightarrow Stop \quad \lhd\; a \neq b \rhd \\ ComTA\,!\,E(SK_{TGS},\, Sk_{Server}).ST \\ \{ST = E(MK_{Server}, (username, starttime, lifetime, SK_{Server}))\} \rightarrow Skip \end{array} \right) \end{array} \right);$$

Finally, the interactions between the client and the server correspond to steps 5 and 6 in Fig.8. Process $USER\_Server$ transmits $Auth^{'}$ and the ticket ST to process $Server$. If process $USER\_Server$ receives an invalid message from $Server$, the process stops. Otherwise, $USER\_Server$ decrypts the message by using Sk_{Server} to get the information of the server, and determines whether it is valid. If it is valid, the authentication is successful, otherwise process $USER\_Server$ terminates.

$$USER\_Server(user, server) =_{df} \; ComSA\,!\,ST.Auth^{'}$$
$$\{Auth^{'} = E(SK_{Server}, (username, starttime, lifetime))\} \rightarrow$$
$$\left(\begin{array}{l} ComSA\,?\,invaild \rightarrow Stop \\ \Box \left(\begin{array}{l} ComSA\,?\,z\,\{\, server\_name = D(Sk_{Server},\, z)\} \rightarrow \\ Server\_faking\_success = \neg(valid(server\_name)) \rightarrow \\ (Stop \lhd\; Server\_faking\_success == true \;\rhd Skip) \end{array} \right) \end{array} \right);$$

Firstly, process $Server$ uses the private key MK^{-1}_{Server} to decrypt ST to get the client's name provided by TGS and the session key SK_{Server}. Then, $Server$ decrypts $Auth^{'}$ using SK_{Server} to obtain the client's name provided by $USER\_Server$. If this name matches the name provided by TGS, process $Server$ sends a message containing the identity information of the server encrypted with SK_{Server} to $USER\_Server$, otherwise it sends an invalid message reply.

$$
\begin{aligned}
&Server(server) =_{df}\ ComSA\,?\,z \ \rightarrow\ ST = get(z) \ \rightarrow \\
&\quad msg_{ST} = D(MK^{-1}_{Server},\ ST) \rightarrow\ \ c = getname(msg_{ST}) \rightarrow \\
&\quad SK_{Server} = D(MK^{-1}_{Server},\ ST) \rightarrow\ Auth^{'} = get(z) \ \rightarrow \\
&\quad msg_{Auth'} = D(SK_{Server},\ Auth^{'}) \rightarrow\ d = getname(msg_{ST}) \ \rightarrow \\
&\begin{pmatrix} ComSA\,!\,invaild \ \rightarrow Stop \quad \lhd\ e \neq f \rhd \\ ComSA\,!\,E(SK_{Server},\ (server\_name, starttime, lifetime)) \ \rightarrow\ Skip \end{pmatrix};
\end{aligned}
$$

5.5. Updated Model Based on Kerberos

Updated Producer based on Kerberos In Kafka messaging mechanism based on Kerberos, process $Producer$, as a client, first authenticates with AS and obtain a ticket to access TGS. Then it needs to authenticate with TGS to obtain tickets to access the server $Broker$ and authenticate with $Broker$ to send messages to $Broker$. The updated $Producer^{'}$ model is as follows:

$$
\begin{aligned}
Producer^{'} =_{df}\ & USER\_AS(producer);\ USER\_TGS(producer, broker); \\
& \begin{pmatrix} ComPL\,!\,Data \ \rightarrow\ ComPL\,?\,ack \ \rightarrow Producer^{'} \\ \lhd\ Connect\_Success == true \ \rhd \quad Stop \end{pmatrix};
\end{aligned}
$$

Updated Broker based on Kerberos AS a server, process $Broker$ needs to verify the identity of the client who wants to access it. After the verification is passed, $Broker$ also needs to provide its own identity to the client. Only after the two-way authentication is successful, the following communication with the client will continue. The updated model is as follows:

$$
\begin{aligned}
&Broker^{'} =_{df}\ \ Server(broker); \\
&\begin{pmatrix} ComPL\,?\,data \ \rightarrow\ ComPL\,!\,Ack \ \rightarrow Broker^{'} \\ \Box ComCL\,?\,request \ \rightarrow\ ComCL\,!\,Data \ \rightarrow Broker^{'} \end{pmatrix};
\end{aligned}
$$

Updated Consumer Based on Kerberos Similarly, process $Consumer$ is a client that need to be authenticated by AS and TGS to obtain a ticket to access $Broker$. It also needs to authenticate with $Broker$ to send messages requesting data. The updated $Consumer^{'}$ model is as follows:

$$
\begin{aligned}
Consumer^{'} =_{df}\ & USER\_AS(consumer);\ USER\_TGS(consumer, broker); \\
& \begin{pmatrix} ComCL\,!\,Request \ \rightarrow\ ComCL\,?\,data \ \rightarrow Consumer^{'} \\ \lhd\ Connect\_Success == true \ \rhd \quad Stop \end{pmatrix};
\end{aligned}
$$

Updated Intruder First, we update the facts the intruder has learned:

$$FACT^{'} =_{df} Fact \cup Time \cup Key \cup MSG \\ \cup \{ E(k, content) \mid k \in Key,\ content \in \{Data, Key, Time\}\}$$

Next, we add the following rules :

$$\{MK^{-1}, E(MK, content)\} \mapsto content, \\ \{SK^{-1}, E(SK, content)\} \mapsto content, \\ \{MK, content)\} \mapsto E(MK, content), \\ \{SK, content)\} \mapsto E(SK, content)$$

The first two rules describe that the intruder can use the corresponding key to decrypt the encrypted messages and get some contents. In the same way, the next two rules represent encryption. The final rule is a structural rule, explaining that the intruder can deduce fact f from a lager set $F^{'}$, if f can be deduced from set F.

Finally, we present an updated model of the intruder process disguised as a producer:

$$FakePro^{'}(F) =_{df} \\ \Box_{m \in MSG} InterceptPL!m \rightarrow FakePro^{'}(F \cup Info(m)) \\ \Box\Box_{m \in MSG \cap Info(m) \subset F} FakePL!m \rightarrow FakePro^{'}(F) \\ \Box\Box_{f \in Fact', f \notin F, F \mapsto f} Init\{datac\_leakage = flase\} \rightarrow Deduce^{'}.f.F \rightarrow \\ \begin{pmatrix} (DataP\_Leaking\_Success\{datap\_leakage = true\} \rightarrow FakePro^{'}(F \cup \{f\})) \\ \lhd\ f == Data\ \rhd \\ (DataP\_Leaking\_Success\{datap\_leakage = false\} \rightarrow FakePro^{'}(F \cup \{f\})) \end{pmatrix}$$

Similarly, we update the intruder model that disguises the consumer:

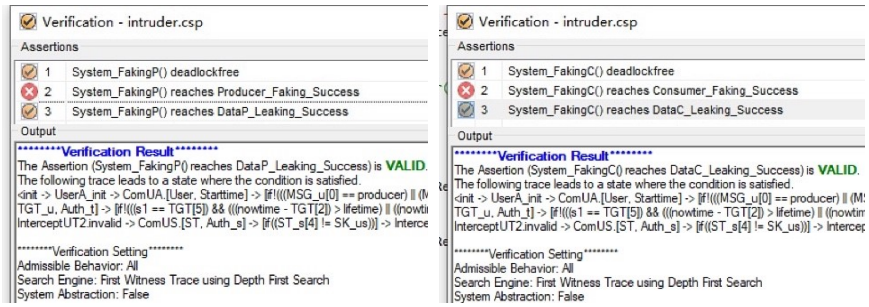
$$FakeCon(F)^{'} =_{df} \\ \Box_{m \in MSG} InterceptCL!m \rightarrow FakeCon(F \cup Info(m)) \\ \Box\Box_{m \in MSG \cap Info(m) \subset F} FakeCL!m \rightarrow FakeCon^{'}(F) \\ \Box\Box_{f \in Fact', f \notin F, F \mapsto f} Init\{datac\_leakage = flase\} \rightarrow Deduce^{'}.f.F \rightarrow \\ \begin{pmatrix} (DataC\_Leaking\_Success\{datac\_leakage = true\} \rightarrow FakeCon^{'}(F \cup \{f\})) \\ \lhd\ f == Data\ \rhd \\ (DataC\_Leaking\_Success\{datac\_leakage = false\} \rightarrow FakeCon^{'}(F \cup \{f\})) \end{pmatrix}$$

Overall Model based on Kerberos In Kafka messaging mechanism based on Kerberos protocol, the overall model is described as follows:

$$
\begin{aligned}
&System\_K =_{df} \ System\_FakingP \ || \ System\_FakingC \\
&System\_FakingP =_{df} \ (\ Producer^{'} \ [|COM\_PATH|] \ Broker^{'} \ [|COM\_PATH|] \\
&\quad Consumer^{'} \ [|COM\_PATH|] \ Kerberos \ [|INTR\_PATH|] \ FakePro^{'}(F)) \\
&System\_FakingC =_{df} \ (\ Producer^{'} \ [|COM\_PATH|] \ Broker^{'} \ [|COM\_PATH|] \\
&\quad Consumer^{'} \ [|COM\_PATH|] \ Kerberos \ [|INTR\_PATH|] \ FakeCon^{'}(F)) \\
&Kerberos =_{df} \ (\ AS \ [|COM\_PATH|] \ TGS \ [|COM\_PATH|] \ Clock)
\end{aligned}
$$

5.6. Verification Results of Model based on Kerberos

We implement the updated model based on Kerberos protocol and the updated intruder model in PAT and get the results of verification shown in Fig.9.



(a) dealock freedom (b) Acknowledgement Mechanism

Fig. 9. Verification Results in $System\_K()$

From Fig.9 (a), we can see *deadlock freedom* is vaild, which means the communication can be completed successfully in the system. $Producer\_Faking\_Success$ is invalid, which means that an intruder cannot disguise a producer and send bogus messages. The property $DataP\_Leaking\_Success$ is valid, which means that an intruder can still intercept the data message transmitted by a real producer successfully, thus causing the data leakage problem.

Similarly, in Fig.9 (b), the property *deadlock freedom* is vaild, which means each process in the system will not run into a deadlock state . The property $Consumer\_Faking\_Success$ is invalid, which means that an intruder cannot send a bogus messages for requesting data by disguising a consumer. The property $DataC\_Leaking\_Success$ is valid, which means that an intruder can still intercept the data transmitted by a broker to a consumer, thus leading to the problem of data leakage.

6. Related Work

In recent years, there have been some researches on the performance of Kafka messaging system in the field of distributed messaging system [14,15,16,29,30]. For instance, in order to set the configuration of the Kafka system correctly under certain hardware conditions to ensure its performance, Han et al. [29] analyzed the structure and workflow of Kafka and proposed a queue-based package flow model to predict the performance of Kafka cloud services. In the paper, they observed the effect of these parameters on the performance by substituting the correlation and fitting results into the fundamental constants of the model and inputting various configuration parameters.

Also, Han et al. [30] introduced a testing tool TRAK to compared the reliability of different messaging transmission semantics in Kafka under the environment of poor network performance by using two indicators namely message loss rate and repetition rate. And in the reliability evaluation of Kafka application scenarios, such as tracking website user information, monitoring server logs, online bank transfer and online booking, etc. Han et al. [28] also tested the effect of various configuration parameters on the reliability of the Kafka system in order to help users weigh the performance and reliability of the application in practical application.

In addition, Sean et al. [15] wanted to find the practical problems that arise when companies use Kafka as a single data store, and to be able to propose solutions to solve these problems. To this end, they proposed some preliminary approaches to ensure the consistency of data from multiple database tables when distributed over Kafka, and how to solve compliance problems by encrypting/decrypting data from Kafka producers and consumers. We can see that these studies mostly focused on the performance analysis of Kafka and how to improve the performance of Kafka applications, but in this paper we focus on the reliability and security of data in the interaction and messaging transmission of various components in Kafka.

At the same time, there are many successful studies on the property analysis and verification of systems and network protocols by formal methods [7,11,12,23,27]. For instance, Lowe et al. [12] analyzed and verified the communication protocol TMN using CSP and FDR, and they have detected the security loopholes of the protocol and put forward the optimization scheme from the theoretical aspect. Thampibal et al. [23] proposed an alternative of formalizing the high-level railway network by using hierarchical timed coloured Petri nets and verified the constructed model with CPN tool to ensure its correctness and security. Wang et al. [27] analyzed the security of the OpenFlow scheduled bundle mechanism and found that it suffered from some kinds of possible attacks by modeling and verifying the mechanism using CSP and PAT. In this paper, we choosed the process algebra CSP and model checking tool PAT to analyze and verify the reliability and security of Kafka messaging mechanism.

7. Conclusion and Future Work

In this paper, we adopted the process algebra CSP to model Kafka messaging mechanism, and utilized the model checker PAT to verify five properties, including *Deadlock Freedom, Acknowledgement Mechanism, Parallelism, Sequentiality* and *Fault Tolerance*. The results of verification show that all properties are valid, which means the pattern of

the distributed messaging system can guarantee the correctness and reliability of communications. In order to further analyze the security of Kafka messaging mechanism, we added the intruder model and the Kerberos model. By comparing the results of Kafka messaging mechanism with or without the secure protocol Kerberos, we can conclude that the protocol Kerberos can effectively prevent the camouflage attack of the intruder, but it can not resist attacks to intercept data, so there are still some security problems.

In the future work, we will put forward a preliminary improvement method from the theoretical aspect to solve these security problems, including digital signature and key encryption, so as to further improve the security of Kafka in the process of messaging transmission.

Acknowledgments. This work was partially supported by the National Natural Science Foundation of China (Grant Nos. 62032024, 61872145), the "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software (Grant No. 22510750100), and the Dean's Fund of Shanghai Key Laboratory of Trustworthy Computing (East China Normal University).

References

1. Rabbitmq., https://www.rabbitmq.com/tutorials/amqp-concepts.html
2. Adams, C.: Kerberos authentication protocol. In: van Tilborg, H.C.A., Jajodia, S. (eds.) Encyclopedia of Cryptography and Security, 2nd Ed, pp. 674–675. Springer (2011), https://doi.org/10.1007/978-1-4419-5906-5_81
3. Brookes, S.D., Hoare, C.A.R., Roscoe, A.W.: A theory of communicating sequential processes. J. ACM 31(3), 560–599 (1984), https://doi.org/10.1145/828.833
4. Clarke, E.M., Henzinger, T.A., Veith, H.: Introduction to model checking. In: Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.) Handbook of Model Checking, pp. 1–26. Springer (2018), https://doi.org/10.1007/978-3-319-10575-8_1
5. Dobbelaere, P., Esmaili, K.S.: Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper. In: Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems, DEBS 2017, Barcelona, Spain, June 19-23, 2017. pp. 227–238. ACM (2017), https://doi.org/10.1145/3093742.3093908
6. Eugster, P.T., Felber, P., Guerraoui, R., Kermarrec, A.: The many faces of publish/subscribe. ACM Comput. Surv. 35(2), 114–131 (2003), https://doi.org/10.1145/857076.857078
7. Fei, Y., Zhu, H.: Modeling and verifying NDN access control using CSP. In: Sun, J., Sun, M. (eds.) Formal Methods and Software Engineering - 20th International Conference on Formal Engineering Methods, ICFEM 2018, Gold Coast, QLD, Australia, November 12-16, 2018, Proceedings. Lecture Notes in Computer Science, vol. 11232, pp. 143–159. Springer (2018), https://doi.org/10.1007/978-3-030-02450-5_9
8. Hesse, G., Matthies, C., Uflacker, M.: How fast can we insert? an empirical performance evaluation of apache kafka. In: 26th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2020, Hong Kong, December 2-4, 2020. pp. 641–648. IEEE (2020), https://doi.org/10.1109/ICPADS51040.2020.00089
9. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM 21(8), 666–677 (1978), https://doi.org/10.1145/359576.359585
10. Lee, V.Y., Liu, Y., Zhang, X., Phua, C., Sim, K., Zhu, J., Biswas, J., Dong, J.S., Mokhtari, M.: ACARP: auto correct activity recognition rules using process analysis toolkit (PAT). In: Donnelly, M.P., Paggetti, C., Nugent, C.D., Mokhtari, M. (eds.) Impact Analysis of Solutions

for Chronic Disease Prevention and Management - 10th International Conference on Smart Homes and Health Telematics, ICOST 2012, Artiminio, Italy, June 12-15, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7251, pp. 182–189. Springer (2012), https://doi.org/10.1007/978-3-642-30779-9_23
11. Liu, A., Zhu, H., Popovic, M., Xiang, S., Zhang, L.: Formal analysis and verification of the PSTM architecture using CSP. J. Syst. Softw. 165, 110559 (2020), https://doi.org/10.1016/j.jss.2020.110559
12. Lowe, G., Roscoe, A.W.: Using CSP to detect errors in the TMN protocol. IEEE Trans. Software Eng. 23(10), 659–669 (1997), https://doi.org/10.1109/32.637148
13. PAT: Process analysis toolkit., http://pat.comp.nus.edu.sg/
14. Prabhu, C., Gandhi, R.V., Jain, A.K., Lalka, V.S., Thottempudi, S.G., Rao, P.P.: A novel approach to extend KM models with object knowledge model (OKM) and kafka for big data and semantic web with greater semantics. In: Barolli, L., Hussain, F.K., Ikeda, M. (eds.) Complex, Intelligent, and Software Intensive Systems - Proceedings of the 13th International Conference on Complex, Intelligent, and Software Intensive Systems, CISIS 2019, Sydney, NSW, Australia, 3-5 July 2019. Advances in Intelligent Systems and Computing, vol. 993, pp. 544–554. Springer (2019), https://doi.org/10.1007/978-3-030-22354-0_48
15. Rooney, S., Urbanetz, P., Giblin, C., Bauer, D., Froese, F., Garcés-Erice, L., Tomic, S.: Kafka: the database inverted, but not garbled or compromised. In: Baru, C., Huan, J., Khan, L., Hu, X., Ak, R., Tian, Y., Barga, R.S., Zaniolo, C., Lee, K., Ye, Y.F. (eds.) 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, December 9-12, 2019. pp. 3874–3880. IEEE (2019), https://doi.org/10.1109/BigData47090.2019.9005583
16. Sharvari T, S.N.K.: A study on modern messaging systems- kafka, rabbitmq and NATS streaming. CoRR abs/1912.03715 (2019), http://arxiv.org/abs/1912.03715
17. Si, Y., Sun, J., Liu, Y., Dong, J.S., Pang, J., Zhang, S.J., Yang, X.: Model checking with fairness assumptions using PAT. Frontiers Comput. Sci. 8(1), 1–16 (2014), https://doi.org/10.1007/s11704-013-3091-5
18. Skeirik, S., Bobba, R.B., Meseguer, J.: Formal analysis of fault-tolerant group key management using zookeeper. In: 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2013, Delft, Netherlands, May 13-16, 2013. pp. 636–641. IEEE Computer Society (2013), https://doi.org/10.1109/CCGrid.2013.98
19. Sun, D., Zhu, H., Fei, Y., Xiao, L., Lu, G., Yin, J.: Formalization and verification of TESAC using CSP. Int. J. Softw. Eng. Knowl. Eng. 29(11&12), 1741–1760 (2019), https://doi.org/10.1142/S0218194019400199
20. Sun, J., Liu, Y., Dong, J.S.: Model checking CSP revisited: Introducing a process analysis toolkit. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation, Third International Symposium, ISoLA 2008, Porto Sani, Greece, October 13-15, 2008. Proceedings. Communications in Computer and Information Science, vol. 17, pp. 307–322. Springer (2008), https://doi.org/10.1007/978-3-540-88479-8_22
21. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: towards flexible verification under fairness. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5643, pp. 709–714. Springer (2009), https://doi.org/10.1007/978-3-642-02658-4_59
22. Tbatou, Z., Asimi, A., Asimi, Y., Sadqi, Y., Guezzaz, A.: A new mutuel kerberos authentication protocol for distributed systems. Int. J. Netw. Secur. 19(6), 889–898 (2017), http://ijns.jalaxy.com.tw/contents/ijns-v19-n6/ijns-2017-v19-n6-p889-898.pdf
23. Thampibal, L., Vatanawood, W.: Formalizing railway network using hierarchical timed coloured petri nets. In: ICIT 2019 - The 7th International Conference on Information Tech-

nology: IoT and Smart City, Shanghai, China, December 20-23, 2019. pp. 338–343. ACM (2019), https://doi.org/10.1145/3377170.3377221
24. Treat, T.: Benchmarking nats streaming and apache kafka., https://dzone.com/articles/benchmarking-nats-streaming-and-apachekafka
25. Vucnik, M., Svigelj, A., Kandus, G., Mohorcic, M.: Secure hybrid publish-subscribe messaging architecture. In: Begusic, D., Rozic, N., Radic, J., Saric, M. (eds.) 2019 International Conference on Software, Telecommunications and Computer Networks, SoftCOM 2019, Split, Croatia, September 19-21, 2019. pp. 1–5. IEEE (2019), https://doi.org/10.23919/SOFTCOM.2019.8903868
26. Wang, G., Koshy, J., Subramanian, S., Paramasivam, K., Zadeh, M., Narkhede, N., Rao, J., Kreps, J., Stein, J.: Building a replicated logging system with apache kafka. Proc. VLDB Endow. 8(12), 1654–1655 (2015), http://www.vldb.org/pvldb/vol8/p1654-wang.pdf
27. Wang, H., Zhu, H., Xiao, L., Fei, Y.: Formalization and verification of the openflow bundle mechanism using CSP. Int. J. Softw. Eng. Knowl. Eng. 28(11-12), 1657–1677 (2018), https://doi.org/10.1142/S0218194018400223
28. Wu, H.: Research proposal: Reliability evaluation of the apache kafka streaming system. In: Wolter, K., Schieferdecker, I., Gallina, B., Cukier, M., Natella, R., Ivaki, N.R., Laranjeiro, N. (eds.) IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops 2019, Berlin, Germany, October 27-30, 2019. pp. 112–113. IEEE (2019), https://doi.org/10.1109/ISSREW.2019.00055
29. Wu, H., Shang, Z., Wolter, K.: Performance prediction for the apache kafka messaging system. In: Xiao, Z., Yang, L.T., Balaji, P., Li, T., Li, K., Zomaya, A.Y. (eds.) 21st IEEE International Conference on High Performance Computing and Communications; 17th IEEE International Conference on Smart City; 5th IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2019, Zhangjiajie, China, August 10-12, 2019. pp. 154–161. IEEE (2019), https://doi.org/10.1109/HPCC/SmartCity/DSS.2019.00036
30. Wu, H., Shang, Z., Wolter, K.: TRAK: A testing tool for studying the reliability of data delivery in apache kafka. In: Wolter, K., Schieferdecker, I., Gallina, B., Cukier, M., Natella, R., Ivaki, N.R., Laranjeiro, N. (eds.) IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops 2019, Berlin, Germany, October 27-30, 2019. pp. 394–397. IEEE (2019), https://doi.org/10.1109/ISSREW.2019.00101

Junya Xu obtained her master degree in formal methods from East China Normal University, Shanghai, in 2021. Her research interests include process algebra and its applications, program analysis and verification.

Jiaqi Yin is currently an assistant professor in Northwestern Polytechnical University, Xi'an, China. He earned his Ph.D. degree in software engineering from East China Normal University, Shanghai, in 2022. His research interests contain formal methods. edge computing, and process algebra.

Huibiao Zhu is currently a professor in East China Normal University, Shanghai. He earned his Ph.D. degree in formal methods from London South Bank University, London, in 2005. During these years, he has studied various semantics and their linking theories for Verilog, SystemC, web services and probability system. He was the Chinese PI of the Sino-Danish Basic Research Center IDEA4CPS.

Lili Xiao is currently a lecturer in Donghua University, Shanghai. She earned her Ph.D. degree in software engineering from East China Normal University, Shanghai, in 2022. Her research interests include process algebra and its applications, program analysis and verification, and weak memory.

Received: July 07, 2021; Accepted: May 11, 2022.