Formal Verification of COCO Database Framework Using CSP

Peimu Li, Jiaqi Yin, Huibiao Zhu*
Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University, Shanghai, China

***Abstract*—Nowadays, many applications are built on distributed databases for scalability and high availability. Therefore, the architecture design of distributed databases needs to satisfy some functional properties to ensure that the database can perform transactions reliably and efficiently. COCO is a distributed OLTP database that supports epoch-based commit and replication and two variants of optimistic concurrency control which use physical time or logical time. In this paper, we first use process algebra CSP to model COCO's architecture. Then we use model checker PAT to verify seven properties, including deadlockfree, consistency, availability, partition tolerance (CAP), and basically availability, soft state, eventual consistency (BASE). The results show COCO's commit and replication protocol satisfy the CAP theorem, and two optimistic concurrency control variants satisfy the BASE theorem.**

***Index Terms*—Distributed OLTP Database, Process Algebra, COCO, Modeling, Verification**

I. Introduction

Many distributed OLTP databases use a shared-nothing architecture for scale out and data partitioning to achieve the scalability of data storage [1], [2]. When the data required by a transaction belongs to multiple data partitions, the system needs a protocol that can coordinate the collaborative work of multiple partitions to ensure the normal execution of the transaction, so a two-phase commit protocol (2PC) is proposed [3]. It is well known that 2PC causes significant performance degradation in distributed databases [4]. There have been some improvements aimed at the defects of 2PC, but most of them require some assumptions which are difficult to achieve, e.g., read/write sets of each transaction has to be known before execution [5].

Lu et al. [6] proposed epoch-based commit and replication, which is an improved protocol based on 2PC, and implemented it in distributed database COCO. The COCO database also supports two variants of optimistic concurrency control: physical time and logical time OCC, which can serialize transactions in physical or logical time [7].

The design of a distributed database architecture often needs to satisfy many functional properties. Fox et al. [8] put forward the CAP theory based on the characteristics of distributed systems. This theory explains three properties that restrict each other in the design of distributed system architecture: consistency, availability, and partition tolerance. Any framework can only satisfy two of them, but not all of them. Pritchett [9] proposed BASE theory, which is a compromise solution for the design of distributed systems against limitations of CAP theory. It allows data inconsistency for a period of time and satisfies three properties: basic availability, soft state and eventually consistency.

At present, the mainstream method of detecting the performance of the database are benchmark testing and load testing. Benchmark testing refers to a test method that quantitatively compares certain specific performance indicators in the system. Currently, the popular benchmarking tools include Facebook's LinkBench [10], Yahoo's YCSB [11] and BigDataBench [12]. Load testing is to test performance of the tested object by continuously increasing the task volume to the tested object until a certain index reaches or exceeds the expectation or a certain resource is exhausted.

Actually, for the reason that the test workload and benchmarks are artificially set, and the test results are directly affected by the hardware performance, the test results still can be improved. In order to solve these challenges, this paper applies a formal method called CSP to verify properties of the database architecture. CSP [13] is an algebra theory proposed by C. A. R. Hoare. It is an abstract language designed to describe process communication in concurrent systems. We use CSP to abstract the architecture and use the model checker PAT [14] to check the properties of the model. In this way, we can reduce the impact of hardware on the verification process and ensure completeness.

The remainder of this paper is organized as follows. In Section II, we briefly introduce the epoch-based commit and replication, two optimistic concurrency controls in COCO and process algebra CSP. In Section III, we use CSP to model the commit protocol and two types of concurrency control in COCO. In Section IV, we implement the achieved formed model of Section III in PAT, and give the definition of the properties that need to be verified and the verification results. Section V concludes the paper and provides some future work.

II. Background

In this section, we introduce the overall architecture of COCO database and process algebra CSP.

A. Epoch-based Commit and Replication

Epoch-based commit and replication contains two protocols: (1) a commit protocol and (2) a replication protocol.

The commit protocol is used to commit completed transactions at the end of the current epoch, shown in Fig.

*Corresponding author: hbzhu@sei.ecnu.edu.cn (H. Zhu).

DOI reference number: 10.18293/SEKE2022-072

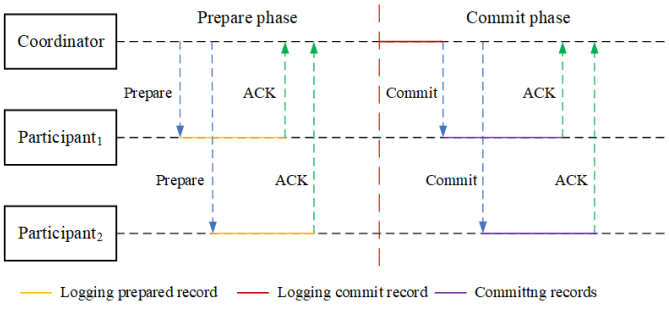


Fig. 1. Epoch-based commit

1. Epoch-based commit contains a $prepare$ phase and a $commit$ phase. In the $prepare$ phase, the coordinator node first sends a preparation message to all other participant nodes. When a participant node receives the preparation message, it prepares to commit all transactions in the current epoch by logging a durable prepared write record with all the transaction IDs (TIDs) of ready-to-commit transactions as well as the current epoch number. When a participant node logs all the necessary write records, it then replies an acknowledgement to the coordinator.

If any participant node fails to send an acknowledgement message due to failures, all transactions in the current epoch must be aborted. Otherwise the coordinator writes a durable commit record with the global current epoch number, and increases the global current epoch number. Then the coordinator sends a commit message to all participant nodes. When a participant node receives a commit message, it commits all the ready-to-commit transactions in the last epoch. Note that even if some transactions in the epoch are aborted due to conflicts, they do not affect the commits of other transactions in the epoch. At this point, the writes of all ready-to-commit transactions in the last epoch are visible to all other users. In the end, all participants send acknowledgement messages to the coordinator, and then start to execute the transactions of the next epoch.

The replication protocol is needed to guarantee consistency and availability of the database. The most common approach is the primary-backup replication. In COCO, atomicity and durability are satisfied at the end of the epoch, so that COCO can perform the replication on backup databases asynchronously. Therefore, after the data update on the primary is completed, the primary can release all locks.

B. Physical Time and Logical Time OCC

Physical Time OCC (PT-OCC) and Logical Time OCC (LT-OCC) are two distributed variants of OCC. In PT-OCC, the transaction needs to lock the data items in the write set. A lock request is only sent to the primary replica of each record. If the data item has been locked by other transactions, the transaction is directly aborted. If the record is in the write and read sets of the transaction at the same time, the transaction will verify whether the TID of the corresponding record in the read set is consistent with that in the primary. If the TID is inconsistent, it means that the record was updated by other transactions during the execution phase, so the transaction simply aborts.

When the transaction has locked the write set, it begins to verify its read set. A read verification request is only sent to the primary replica of each record. If a record in the primary is locked by other transactions or the TID is inconsistent with the record in the read set, then the transaction simply aborts. At the same time, COCO generates a new TID for the transaction. TID marks the order between transactions.

After successfully verifying the read set, the transaction writes the result back to the database. A write request is sent to the primary replica of each record. After the result is written, the lock of the record in the primary replica is released immediately, and then the result is written to other replicas asynchronously.

In LT-OCC, a serialized transaction can read, write and commit in the space of logical time. Each record in the database is associated with two logical timestamps $[wts, rts]$. Here, wts is the timestamp of the last modification time of this record. rts represents the effective time of this record, which means that it is valid to read this record at any logical time ts that satisfies $wts \leq ts \leq rts$. The control flow of the LT-OCC algorithm is basically the same as that of PT-OCC. The difference is TID is used to identify transactions in PT-OCC and $[wts, rts]$ is used in LT-OCC.

C. CSP

CSP is a formal language which has an important impact on the development of Golang. The following is some widely used CSP syntax:

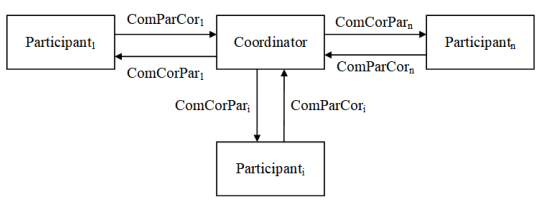
- $SKIP$ denotes that a process terminates successfully.
- $a \rightarrow P$ indicates the process executes action a first, and then behaves like P.
- $P \square Q$ represents the general choice. It behaves like P or Q, and the environment decides the selection.
- $P||Q$ stands for concurrent execution of P and Q. The shared actions must be executed synchronously.
- $P|||Q$ denotes that P interleaves Q.
- $c?x \rightarrow P$ indicates that a value is recieved through channel c and process assigns it to variable x, and then behaves like P.
- $c!e \rightarrow P$ denotes the process sends the value of expression e through channel c first, and then behaves like P.
- $P;Q$ represents sequential execution, process executes P, then executes Q after P which first terminates.
- $P \triangleleft b \triangleright Q$ denotes the condition. If b is true, then process behaves like P. Otherwise process behaves like Q.

III. Modeling COCO Database Framework

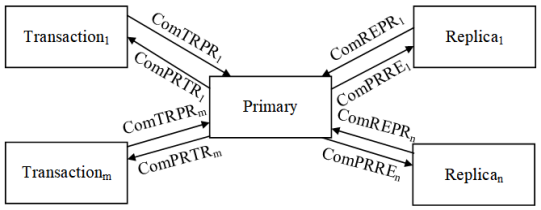
In this section, we use process algebra CSP to model COCO architecture. First, we introduce the messages and channels used in our model, and then we respectively introduce the CSP models of COCO architecture. COCO architecture includes epoch-based commit protocol, replication protocol and two variants of optimistic concurrency control.

A. Messages and Channels

Before modeling the commit protocol and concurrency control algorithm in COCO, we need to define the messages, and channels used for modeling. We define multiple channels for data interaction between various modules in the COCO database system. Fig. 2 gives the channels of communication in COCO:



(a) Channels of Epoch-based Commit



(b) Channels of Optimistic Concurrency Control and Replication

Fig. 2. Channels of COCO database

In order to define the message conveniently, we define three message content sets: REQ, ACK and $DATA$, which represent content of the request, confirmation and data messages respectively. Based on the above definitions, we design multiple types of messages for information exchange between entities. These messages are defined as follows:

$MSG =_{df} MSG_{req} \cup MSG_{ack} \cup MSG_{data}$

$MSG_{req} =_{df} \{msg_{req}.C.P.Content \mid msg_{req} \in TYPE , C \in Coordinator, P \in Participant, Content \in REQ, \}$

$MSG_{ack} =_{df} \{msg_{ack}.P.C.Content \mid c \in Coordinator , P \in Participant, Content \in ACK, \}$

$MSG_{data} =_{df} \{msg_{data}.P.R.Content \mid P \in Primary, R \in Replica, Content \in DATA, \}$.

In COCO framework, we define MSG_{req} to represent the request messages, including six types of requests: prepare, commit, abort, read, write, and lock. Six types of requests are in $TYPE$ set. MSG_{ack} represents confirmation of the request message, and MSG_{data} represents data information passed by the transaction in the read and write process.

B. Epoch-based Commit and Replication Modeling

The epoch-based commit protocol includes two types of processes: coordinator and participants. The coordinator process is modeled as follows:

$$
\begin{aligned}
&Coordinator() =_{df} (|||i : \{1..N\}@ComCorPar[i]!msg_{prep}.\\
&\quad C.P.epoch\_num \rightarrow Skip); (|||i : \{1..N\}@ComParCor\\
&\quad [i]?msg_{ack}.P.C.R\_C\_TID \rightarrow Check\{if(msg_{ack} ==\\
&\quad No)\{hasNo = true\}\} \rightarrow Skip); decide\{if(hasNo ==\\
&\quad true)\{choice = ABORT\}else\{choice = COMMIT\}\}\\
&\quad \rightarrow CoordCommitPhase(choice)
\end{aligned}
$$

$$
\begin{aligned}
&CoordCommitPhase(choice) =_{df}\\
&\quad if(choice == COMMIT)\{\\
&\qquad writeCommitRecord \rightarrow increaseEpoch\{epoch\_n\\
&\qquad um = epoch\_num + 1\} \rightarrow Skip;\\
&\qquad (|||i : \{1..N\}@ComCorPar[i]!msg_{commit}.C.P.epo\\
&\qquad ch\_num - 1 \rightarrow Skip); (|||i : \{1..N\}@ComParCor\\
&\qquad [i]?msg_{ack}.P.C.com\_TID \rightarrow Skip);\\
&\qquad releaseResults \rightarrow Coordinator()\\
&\quad \}\\
&\quad else\{\\
&\qquad (|||i : \{1..N\}@ComCorPar[i]!msg_{abort}.C.P.epo\\
&\qquad ch\_num \rightarrow Skip);\\
&\qquad (|||i : \{1..N\}@ComParCor[i]?msg_{ack}.P.C.abor\\
&\qquad t\_TID \rightarrow Skip);\\
&\qquad abort \rightarrow Coordinator()\\
&\quad \}
\end{aligned}
$$

In the *prepare* phase, the *coordinator* first sends a MSG_{prep} to N numbers of *participants*, and then receives MSG_{ack} sent by *participant* nodes. If there are *participant* nodes that cannot commit the transaction, the *choice* is assigned to $ABORT$, otherwise it is assigned to $COMMIT$. In the *commit* phase, if $choice == ABORT$, then the *coordinator* needs to send MSG_{abort} to all *participant* nodes to abort all transactions in this epoch. After receiving MSG_{ack} from all *participant* nodes, it re-executes all transactions in the current epoch. If $choice == COMMIT$, *coordinator* writes a durable commit record with *epoch_num*, and then increases *epoch_num*. It also sends MSG_{commit} to all *participant* nodes, and then receives MSG_{ack} sent by *participant* nodes. After completing above tasks, the *coordinator* releases results to users and then executes the commit of the next epoch.

The definition of the participant process is as follows:

$$
\begin{aligned}
&Participant(i) =_{df} ComParCor[i]?msg_{prep}.C.P.epoch\_num\\
&\rightarrow \begin{pmatrix} logWriteRecord \rightarrow ComParCor[i]!Yes.\\ P.C.R\_C\_TID \rightarrow ParComPhase(i)\\ \Box\\ ComParCor[i]!No.P.C.R\_C\_TID\\ \rightarrow ParComPhase(i) \end{pmatrix}\\
&ParComPhase(i) =_{df} ComCorPar[i]?msg_{req}.C.P.epoch\_num\\
&\rightarrow \begin{pmatrix} ComParCor[i]!msg_{ack}.P.C.com\_TID\\ \rightarrow Participant(i)\\ \triangleleft(msg_{req} == msg_{commit})\triangleright\\ ComParCor[i]!msg_{ack}.P.C.abort\_TID\\ \rightarrow Participant(i) \end{pmatrix}
\end{aligned}
$$

In *prepare* phase, the *participant* first receives the MSG_{prep} sent by the *coordinator*, then if the *participant* successfully writes the commit record, a Yes MSG_{ack} is sent directly to the *coordinator*. Otherwise, the *participant* sends No MSG_{ack}. In *commit* phase, the *participant* first waits to receive MSG_{req} from the *coordinator*. If $MSG_{req} == MSG_{commit}$, then the *participant* commits the result and sends a commit message to the *coordinator*. If $MSG_{req} == MSG_{abort}$, the *participant* sends an abort

message. After that, $participants$ and the $coordinator$ synchronize to enter the next epoch of transaction commit.

Based on the above modeling, an overall epoch-based commit protocol is defined as follows:

$$Epoch\_commit() =_{df} Coordinator()|||(|||i : \{1...N\}@Participant(i))$$

The epoch-based commit protocol consists of a $coordinator$ and several $participants$. The $coordinator$ and each $participant$ execute concurrently, and freely interleave execution between $participant$ processes.

The replication protocol mainly includes two types of processes, $Primary\_replication$ and $Replica$. The $Primary\_replication$ is responsible for controlling and coordinating entire replication execution on the primary database. $Replica$ is a process that runs on the replica server and cooperates with the primary. The definitions are as follows:

$$\begin{aligned}
&Primary_replication(records) =_{df} (|||i : \{1..N\}@ComP \\
&\quad RRE[i]!msg_{write}.P.R.records \rightarrow ComREPR[i]?ms \\
&\quad g_{ack}.R.P.content \rightarrow Primary_replication(records)) \\
&Replica(i) =_{df} ComPRRE[i]?msg_{write}.P.R.records \rightarrow \\
&\quad WriteBack(i, records) \rightarrow ComREPR[i]! \\
&\quad msg_{ack}.R.P.Yes \rightarrow Replica(i) \\
&WriteBack(i, records) =_{df} \\
&\left(\begin{array}{c} Skip \\ \triangleleft(records == null)\triangleright \\ \left(\begin{array}{c} write\{replica_i.record = record\} \rightarrow \\ WriteBack(i, records') \\ \triangleleft(replica_i.record.tid < record.tid)\triangleright \\ WriteBack(i, records') \end{array} \right) \end{array} \right)
\end{aligned}$$

The $WriteBack$ first determines whether the $records$ is empty. If it is empty, the process terminates directly. Otherwise, if the database record's tid is greater than the tid of $records$, it indicates that the record has been updated by other transactions executed later, so the replica directly skips this record. Otherwise, it updates the record, and then calls $WriteBack(i, records')$. $records'$ contains all $records$ elements except the first record.

$$\begin{aligned}
&Replication(records) =_{df} \\
&Primary_replication(records)|||(|||i : \{1...N\}@Replica(i))
\end{aligned}$$

The overall protocol consists of a $primary\_replication$ and N $replica$ processes. The modeling is shown above.

C. PT-OCC and LT-OCC Modeling

OCC can be roughly divided into three phases: (1) locking the write set, (2) validating the read set, (3) writing back to the database. However, there is a difference between PT-OCC and LT-OCC in phase (2), and the other phases are basically the same. Based on the above facts, we can perform unified modeling on phases (1) and (3) of the two algorithms, and model the phase (2) independently.

In the first phase, the transaction sends a MSG_{lock} to the primary of the record, and then receives the MSG_{ack} sent by the primary. If $content == Yes$, then the transaction continues to send the request for the next record, otherwise the transaction releases locked records and aborts. The definition of $Trans\_lock$ and $Releaselock$ are as follows:

$$\begin{aligned}
&Trans_lock(i, write_set) =_{df} Skip \triangleleft (write_set == null)\triangleright \\
&\left(\begin{array}{c} ComTSPR[i]!msg_{lock}.T.P.record \rightarrow ComPRTS[i]? \\ msg_{ack}.P.T.content \rightarrow \\ \left(\begin{array}{c} Trans_lock(i, write_set') \\ \triangleleft(content == Yes)\triangleright \\ Releaselock(i, lockedRecords) \rightarrow Skip \end{array} \right) \end{array} \right) \\
&Releaselock(i, records) =_{df} Skip \triangleleft (records == null)\triangleright \\
&\quad release\{database.record.locked = false\} \rightarrow \\
&\quad Releaselock(i, records')
\end{aligned}$$

The definition of $primary$ in the first phase is as follows:

$$\begin{aligned}
&Primary_lock() =_{df} |||i : \{1...N\}@ComTSPR[i]?msg_{lock}. \\
&\quad T.P.record \rightarrow find\{if(record \notin read_set)\{record.tid \\
&\quad = database.record.tid\}\} \rightarrow \\
&\left(\begin{array}{c} Locking\{database.record.locked = True\} \rightarrow \\ ComPRTS[i]!msg_{ack}.P.T.Yes \rightarrow Primary_lock() \\ \triangleleft(database.record.locked == false \wedge \\ record.tid == database.record.tid)\triangleright \\ ComPRTS[i]!msg_{ack}.P.T.No \rightarrow Primary_lock() \end{array} \right)
\end{aligned}$$

If $record \notin read\_set$, $primary$ reads $database.record.tid$ as the tid of $record$. If the record is not occupied by others and $record.tid == database.record.tid$, the $primary$ can lock and send Yes to the transaction, otherwise send No.

Next, the algorithm verifies whether the $records$ in the read set are still valid. We use $Primary\_valid$ to describe the behavior of the primary in the validation phase, and its definition is as follows:

$$\begin{aligned}
&Primary_valid() =_{df} |||i : \{1...N\}@(ComTSPR[i]? \\
&\quad msg_{valid}.T.P.record \rightarrow ComPRTS[i]!msg_{data}. \\
&\quad P.T.database_record \rightarrow Primary_valid())
\end{aligned}$$

Transactions in the verification phase have different behaviors under different concurrency control. The verification phase of the transaction in PT-OCC and LT-OCC are defined as follows:

$$\begin{aligned}
&Trans_valid_PT(i, read_set) =_{df} Skip \triangleleft (read_set == null) \\
&\triangleright \left(\begin{array}{c} ComTSPR[i]!msg_{valid}.T.P.record \rightarrow ComPRT \\ S[i]?msg_{data}.P.T.content \rightarrow \\ \left(\begin{array}{c} validate \rightarrow Trans_valid_PT(i, read_set') \\ \triangleleft(content.locked == false \wedge \\ record.tid == content.record.tid)\triangleright \\ Releaselock(i, lockedRecords) \rightarrow Stop \end{array} \right) \end{array} \right) \\
&Trans_valid_LT(i, read_set) =_{df} Skip \triangleleft (read_set == null) \\
&\triangleright \left(\begin{array}{c} ComTSPR[i]!msg_{valid}.T.P.record \rightarrow ComPRT \\ S[i]?msg_{data}.P.T.content \rightarrow \\ \left(\begin{array}{c} validate \rightarrow Trans_valid_LT(i, read_set') \\ \triangleleft(record.wts == content.wts \wedge \\ (content.rts \geq record.tid \vee \\ content.locked == false))\triangleright \\ Releaselock(i, lockedRecords) \rightarrow Stop \end{array} \right) \end{array} \right)
\end{aligned}$$

The main difference between the transaction in LT-OCC and PT-OCC is the condition for judging whether the record is successfully verified. In LT-OCC, the wts of the $record$ in the $read\_set$ of the transaction must be the same as the wts of the corresponding record in the database. For the case where the rts of the $record$ is less than the tid of the transaction, as long as the record has not been locked by

other transactions, the transaction also considers the $record$ to be valid and the verification is successful.

After verifying all records in $read\_set$, the transaction writes the records in $write\_set$ into the $primary$ database. The transaction definition in the write phase is as follows:

$$Trans\_write(i, write\_set) =_{df} Skip \triangleleft (write\_set == null) \triangleright \begin{pmatrix} ComTSPR[i]!msg_{write}.T.P.record \rightarrow ComPRT \\ S[i]?msg_{ack}.P.T.content \rightarrow ComTSPR[i]! \\ msg_{release}.T.P.write\_set \rightarrow Trans\_write(\\ i, write\_set') \end{pmatrix}$$

The transaction first checks whether there are any records in $write\_set$. If not, the write is complete and the transaction can be terminated at this point. Otherwise, the transaction sends a write request MSG_{write} to the $primary$, and then receives the MSG_{ack} from the $primary$. After that, the transaction sends a request $MSG_{release}$ to release the record lock to the $primary$, and then tries to write the next record. The primary definition in write phase is as follows:

$$Primary\_write() =_{df} |||i : \{1...N\}@(ComTSPR[i]? msg_{write}.T.P.record \rightarrow write\{database.record = record\} \rightarrow ComPRTS[i]!msg_{ack}.P.T.Yes \rightarrow ComTSPR[i]?msg_{release}.T.P.write\_set \rightarrow Release(i, record) \rightarrow Primary\_replication(record))$$

The $primary$ receives the write request MSG_{write} from the transaction, and then writes the $record$. After successfully writing the $record$, the $primary$ sends a confirmation message MSG_{ack} to the transaction. After receiving the release request from the transaction, the $primary$ calls the $Release$ function to release the lock of the record, and then enters the replication phase.

The overall definition of concurrency control algorithm includes three parts: transaction, primary and replica. The definition of $Primary$ is as follows:

$$Primary() =_{df} Primary\_lock()|||Primary\_valid()||| Primary\_write()$$

The definition of $Transaction\_PT$ in PT-OCC is as follows. $Transaction\_LT$ in LT-OCC differs only in the second phase.

$$Transaction\_PT(i, read\_set_i, write\_set_i) =_{df} (Transaction\_lock(i, write\_set_i); Transaction\_valid\_PT(i, read\_set_i); Transaction\_write(i, write\_set_i))$$

The definitions of PT-OCC and LT-OCC are as follows:

$$PT\_OCC() =_{df} (|||i : \{1...N\}@Transaction\_PT(i, read\_set_i, write\_set_i))||Primary()||(|||i : \{1...N\}@Replica(i))$$

$$LT\_OCC() =_{df} (|||i : \{1...N\}@Transaction\_LT(i, read\_set_i, write\_set_i))||Primary()||(|||i : \{1...N\}@Replica(i))$$

Both $PT - OCC$ and $LT - OCC$ are composed of concurrent execution of multiple $transaction$, a $Primary$ and multiple $Replica$ processes.

IV. Implementation and Verification

In this section, based on the achieved formed model in Section III, now we conduct verification of the properties abstracted from the specification.

A. Properties

a) Deadlockfree: This property refers to the situation in which processes are never deadlocked. PAT provides atomic statements to verify deadlockfree.

b) Consistency: This property asserts during the execution of a transaction, data can only be converted from one consistency state to another consistency state.

$$\#define\ Consistency\ (\wedge i : \{1...N\}record_i == last\_record) \vee (\wedge i : \{1...N\}record_i == cur\_record)$$
$$\#assert\ Epoch\_commitl() \mid=\ Consistency$$

c) Availability: It means every request in a distributed system can be responded to.

$$\#define\ Availability\ (hasNo == True \wedge finished == True)$$
$$\#assert\ Epoch\_commit() \mid=\ Availability$$

d) Partition Tolerance: It means that when a node or network partition in a distributed system fails, the entire system can still provide external services that satisfy consistency and availability.

$$\#define\ PartitionTolerance\ finished == True$$
$$\#assert\ Epoch\_commit() \mid=\ PartitionTolerance$$

e) Basically Availability: This property means that when some requests failure or unpredictable failures occur in the system, the system can still guarantee the normal execution of most transactions.

$$\#define\ BasicallyAvailability\ (existCrash == True) \wedge (available == True)$$
$$\#assert\ PT\_OCC() \mid=\ BasicallyAvailability$$
$$\#assert\ LT\_OCC() \mid=\ BasicallyAvailability$$

f) Soft State: This property refers to allowing the data in the system to have an intermediate state, and this state does not affect the overall availability of the system.

$$\#define\ SoftState\ (\vee i : \{1...N\}record_i! = last\_record) \wedge (\vee i : \{1...N\}record_i! = cur\_record)$$
$$\#assert\ PT\_OCC() \mid=\ SoftState$$
$$\#assert\ LT\_OCC() \mid=\ SoftState$$

g) Eventually Consistency: It refers to the fact that all data copies in the system can finally reach a consistent state after a period of synchronization without the guarantee of strong consistency of system data.

$$\#define\ EventuallyConsistency\ EG((\wedge i : \{1...N\}record_i == last\_record) \vee (\wedge i : \{1...N\}record_i == cur\_record))$$

$$\#assert\ PT\_OCC()\ |=\ EventuallyConsistency$$
$$\#assert\ LT\_OCC()\ |=\ EventuallyConsistency$$

B. Results

We use the model checker PAT to verify the main frameworks of COCO distributed database such as epoch-based commit and replication, PT-OCC and LT-OCC. The verification results are shown in Fig. 3 and Fig. 4. The

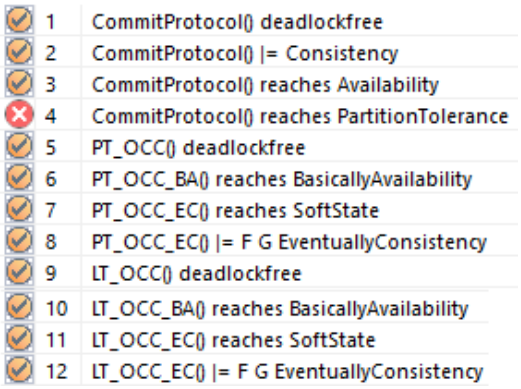


Fig. 3. The Verification Results of COCO Database Framework

Fig. 4. The Details of the Partial Verification Results

epoch-based commit protocol executed at the end of the epoch satisfies the consistency and availability but does not satisfy the partition tolerance(i.e. the 4th property in Fig. 3), which is in line with the CAP theory. During an epoch, the two types of concurrency control satisfy basically availability, soft state and eventually consistency, which meet the BASE theory. From the analysis of the above results, the COCO distributed database guarantees basic availability within an epoch and at the same time can satisfy strong consistency at the end of the epoch.

V. Conclusion and Future Work

COCO is a distributed database that regards the epoch as the unit of transaction commit and uses optimistic concurrency control. This paper used process algebra CSP to model COCO's epoch-based commit and replication, physical time OCC and logical time OCC, and implemented these models in the model checker PAT. The CAP and BASE theories put forward the properties that the distributed system architecture needs to satisfy, and we verified the properties of COCO in an epoch cycle. It has been verified that (1) epoch-based commit and replication satisfy consistency and availability but not partition tolerance, and (2) PT-OCC and LT-OCC satisfy basic availability, soft state, and eventually consistency. This shows that COCO can guarantee high availability during an epoch cycle, and can also guarantee consistency at the end of the epoch. In the future, we will verify the isolation of COCO and sequential consistency of concurrency control.

VI. Acknowledgements

This work was partly supported by the National Key Research and Development Program of China (Grant No. 2018YFB2101300), the National Natural Science Foundation of China (Grant Nos. 61872145, 62032024), Shanghai Trusted Industry Internet Software Collaborative Innovation Center, and the Dean's Fund of Shanghai Key Laboratory of Trustworthy Computing (East China Normal University).

References

[1] Shute J, Vingralek R, Samwel B, et al. F1: A Distributed SQL Database That Scales. PVLDB, 2013, 6(11): 1068–1079.
[2] Verbitski A, Gupta A, Saha D, et al. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. SIGMOD, 2017: 1041–1052.
[3] Mohan C, Lindsay B, and Obermarck R. Transaction Management in the R* Distributed Database Management System. TODS, 1986, 11(4): 378–396.
[4] Bailis P, Fekete A, Franklin M, Ghodsi A, Hellerstein J, and Stoica I. Coordination Avoidance in Database Systems. PVLDB, 2014, 8(3): 185–196.
[5] Lin Q, Chang P, Chen G, Ooi B C, Tan K, and Wang Z. Towards a Non-2PC Transaction Management in Distributed Database Systems. SIGMOD, 2016: 1659–1674.
[6] Lu, et al. Epoch-based Commit and Replication in Distributed OLTP Databases. PVLDB, 2021, 14(5): 743-756.
[7] Kung H T and Robinson J T. On Optimistic Methods for Concurrency Control. TODS, 1981, 6(2): 213–226.
[8] Fox A, Brewer E A. Harvest, Yield, and Scalable Tolerant Systems. IEEE, 1999: 174-178.
[9] Pritchett D. BASE: An Acid Alternative: In Partitioned Databases, Trading some Consistency for Availability can Lead to Dramatic Improvements in Scalability. Queue, 2008, 6(3): 48-55.
[10] Timothy A, et al. LinkBench: a Database Benchmark Based on the Facebook Social Graph. SIGMOD, 2013: 1185-1196.
[11] Cooper B F, Silberstein A, Tam E, et al. Benchmarking Cloud Serving Systems with YCSB. SOCC, 2010: 143-154.
[12] Liang F, Feng C, Lu X, et al. Performance Benefits of DataMPI: A Case Study with BigDataBench. BPOE, 2014: 111-123.
[13] Hoare C A R. Communicating Sequential Processes. Communications of the ACM, 1978, 21(8): 666-677.
[14] Si Y, Sun J, Liu Y, et al. Model Checking With Fairness Assumptions using PAT. Frontiers of Computer Science, 2014, 8(1): 1-16.