This is an extension to OSDI’14 paper *Extracting more concurrency from distributed transactions*.

A Proof of Correctness for **ROCOCO**

Shuai Mu$^{1,2}$, Wyatt Lloyd$^{3,4}$, Jinyang Li$^2$

$^1$Tsinghua University, $^2$New York University, $^3$University of Southern California, $^4$Facebook

NYU-TR-2014-970
Last revision: January, 2018

**Abstract:** This paper presents a proof of correctness for ROCOCO, a new concurrency control protocol for distributed transactions. ROCOCO can achieve strict serializability without the cost of blocking or aborting. It executes a transaction as a collection of atomic pieces, each of which commonly involves only a single server. Servers first track dependencies between concurrent transactions without actually executing them. At commit time, a transaction’s dependency information is sent to all servers so they can re-order conflicting pieces and execute them in a serializable order.
1 Introduction

Large distributed online transaction processing (OLTP) systems require concurrency control to guarantee strict serializability [3, 1], so that websites running on top of them can function correctly.

While concurrency control is a well-studied field, traditional protocols such as two-phase locking (2PL) and optimistic concurrency control (OCC) perform poorly when workloads exhibit a non-trivial amount of contention. The performance drop is particularly pronounced when running these protocols in a distributed setting.

In [2], we have presented Rococo (ReOrdering COnlicts for COncurrency), a distributed concurrency control protocol that extracts more concurrency under contended workload than previous approaches. Rococo achieves safe interleavings without aborting or blocking transactions using two key techniques: 1) deferred and reordered execution using dependency tracking; and 2) offline safety checking based on the theory of transaction chopping.

In this accompanying technical report, we formally specify the protocol of Rococo and prove its correctness.

Disclaimer: The proof in this technical report includes Rococo with optimizations but does not cover other useful extensions such as read-only transactions, merged pieces, failure recovery and garbage collection.

2 Overview

To use ROCOCO, programmers must structure their transactions as a collection of atomic pieces, each typically involving data access on a single server. Before any runtime execution, ROCOCO runs an offline checker to determine if a particular mix of transactions are safe (i.e. reorderable) for ROCOCO at runtime or not.

At the runtime, the ROCOCO protocol uses a set of coordinators to coordinate the execution and commit of transactions on behalf of clients. It runs in two phases. In the first phase, called start phase, a coordinator sends the pieces to their involved servers and establishes a provisional order of execution on each server. Servers typically defer execution of the pieces until the second round so they can be reordered if necessary. Servers complete the first phase by replying to the coordinator with dependency information that indicates the order of arrival for conflicting pieces of different transactions. The coordinator aggregates this dependency information collected from servers. In the second phase, called commit phase, a coordinator distributes the aggregated dependency information to all involved servers of a transaction. Servers aggregate the received information into its local dependency graph. They may also optionally ask other servers for missing dependency information. Using the complete dependency information, servers can recognize if the pieces of concurrent transactions arrived in a strictly serializable order in the first phase. If so, they execute pieces in that order. If not, servers reorder the pieces deterministically and then execute them. In both cases, ROCOCO is able to avoid aborts and commits all transactions.

3 Preliminaries

Before describing the protocol in details, we explain a few preliminary concepts.

Piece

A transaction is made up of many pieces, each of which represents logic that can be executed atomically on a single server. In our implementation, piece atomicity is achieved through single threaded local database execution. ROCOCO requires each piece to have a known read/write set prior to its execution so that a server can accurately infer conflict information while still deferring piece execution.

Dependency graph dep

Each server $S$ maintains a dependency graph, $S$.dep. In this graph, each vertex represents a transaction, and each edge represents the arriving order of two conflicting pieces during the start phase for a pair of transactions. We use $T' \rightarrow T$ to denote that $T'$ immediately precedes $T$ (and $T'$ and $T$ have conflicting accesses to a data item). We use $T' \leftarrow T$ to denote that $T'$ is an ancestor of $T$, i.e. there exists a path
$T' \rightarrow ... \rightarrow T$ in $S$.dep. In particular, we use $T' \xrightarrow{i} T$ to refer that the edge between $T'$ and $T$ are annotated as "i(mmediate)". $T' \xleftarrow{i} T$ means that the edges on the path between $T'$ and $T$ are all annotated as "i". Similarly, $T' \xrightarrow{d} T$ and $T' \xleftarrow{d} T$ refers to that the edges are annotated as "d(eferrable)".

**Transaction Status**

The dependency graph contains the status information of a transaction, which can be one of the following:

- **UNKNOWN.** This is a placeholder status for a transaction that does not have enough information in the dependency graph. It happens in graph propagation when a server shares only a subset of graph.
- **STARTED.** When a server receives the start request of a piece, which belongs to a transaction $T$, it sets $T$’s status to be STARTED.
- **COMMITTING.** When a server receives the commit request of a transaction $T$, it changes $T$’s status to COMMITTING.
- **DECIDED.** Once a server has received complete dependency information that allows it to guarantee a strictly serializable order of execution for transaction $T$, it sets $T$’s status be DECIDED.

We define an ordering among these four status flags: UNKNOWN < STARTED < COMMITTING < DECIDED. This order is useful for aggregating the status information, e.g. when a server receives a commit request. In particular, the server always keeps the highest status flag seen for a particular transactions. The simplified protocol only uses STARTED and COMMITTING flags. The optimized protocol uses all.

### 4 Simplified Rococo

#### 4.1 Pseudocode

**Coordinator $C::$PROCESS_TXN($T$)**

```plaintext
// the start phase
for each piece $p_i$ in $T$ do
    wait until all input for $p_i$ is ready
    (dep_i, output_i) := S_i.START_TXN($p_i$)
    dep := DEP_UNION(dep, dep_i)
    output := output \cup output_i
    dep[T], status := COMMITTING
// the commit phase
for each server $S_j$ involved in $T$ do
    output := output \cup S_j.COMMIT_TXN($T$, dep)
reply output
```

**Algorithm 2: Server $S::$START_TXN($p$)**

```plaintext
$S$.dep[p.owner], status := STARTED
for each $p'$ in $S$.dep that conflicts with $p$ do
    if $p$.immediate then
        add $p'$.owner $\xrightarrow{i}$ p.owner into $S$.dep
        output := EXECUTE($p$)
    else
        add $p'$.owner $\xrightarrow{d}$ p.owner into $S$.dep
        output := nil
reply (S.dep, output)
```
Algorithm 3: Server \( S::\text{commit}_\text{txn}(T, \dep_info) \)

1. \( S.dep := \text{dep}_\text{union}(S.dep, \dep_info) \)
2. \( \text{for each } T' \leadsto T \text{ in } S.dep \text{ and } S.dep[T'].status = \text{STARTED} \) do
3.   \( \text{if } T' \text{ does not involve } S \text{ then} \)
4.   \( \text{dep}' := S'.\text{ask}_\text{txn}(T') \) where \( S' \) is a server involved in \( T' \)
5.   \( S.dep := \text{dep}_\text{union}(S.dep, \dep') \)
6.   \( \text{wait until } S.dep[T'].status \geq \text{COMMITTING} \)
7. \( T^{\text{SCC}} := \text{strongly}_{\text{connected}}_{\text{component}}(T, S.dep) \)
8. \( \text{for each } T' \notin T^{\text{SCC}} \text{ and } T' \leadsto T \) do
9.   \( \text{if } T' \text{ involves } S \text{ then} \)
10. \( \text{wait until } T'.\text{finished} = \text{true} \)
11. \( \text{DETERMINISTIC}_{\text{topological}}_{\text{sort}}(T^{\text{SCC}}) \)
12. \( \text{for each } T' \text{ in } T^{\text{SCC}} \) do
13.   \( \text{if } T' \text{ involves } S \text{ and } T'.\text{finished} = \text{false} \) then
14.     \( \text{for each deferred } p' \text{ of } T' \) do
15.     \( T'.\text{output} := T'.\text{output} \cup \text{execute}(p') \)
16.     \( T'.\text{finished} := \text{true} \)
17. \( \text{reply } T'.\text{output} \)

\( ^* \text{Once the } S.dep \text{ has changed during the loop, the loop condition should be recalculated.} \)

Algorithm 4: Server \( S::\text{ask}_\text{txn}(T) \)

18. \( \text{wait until } S.dep[T].status \geq \text{COMMITTING} \)
19. \( \text{reply } S.dep \)

Algorithm 5: \( \text{OPERATOR dep}_\text{union} \ (\dep, \dep') \)

20. \( \text{for each } T' \rightarrow T \text{ in } \dep' \) do
21.   \( \text{append } T' \rightarrow T \text{ into } \dep \)
22. \( \text{for each } T \text{ in } \dep' \) do
23.   \( \text{if } \dep[T].status < \dep'[T].status \text{ then} \)
24.     \( \dep[T].status := \dep'[T].status \)

4.2 Proof

In order to prove that ROCOCO is serializable, we need to argue that the execution of ROCOCO does not cause any cyclic serialization graphs. And then we prove ROCOCO is strictly serializable by proving that the serialization graph it generates preserves natural time order.

**Serialization graph:** The serialization graph (a.k.a. conflict graph) is a directed graph corresponding to some global execution schedule. Each vertex represents a transaction. Each directed edge represents an ordered conflict: if transactions \( T_i \) and \( T_j \) have conflicting accesses to the same item (i.e. one access is write) and the corresponding data access of \( T_i \) executes before that of \( T_j \), then the serialization graph contains \( T_i \rightarrow T_j \).

To begin with, all conflicting accesses reflected in the serialization graph are observed by some server. And we use this basic property throughout the proofs.

**Lemma 1** For any pair of transactions \( T_i \rightarrow T_j \) in the serialization graph, \( T_i \) contains a piece \( p_i \) and \( T_j \) contains a piece \( p_j \), such that \( p_i \) and \( p_j \) has conflicting access to certain items and there exists a server \( S_{i,j} \) which executes \( p_i \) before \( p_j \).
Proof: The lemma is obvious by the definition of serialization graph. If we think of pieces as subtransactions, we can draw serialization graph of pieces. Because we assume atomic execution on pieces, the serialization graph of pieces must be acyclic, i.e. one piece must be executed before another.

A similar property holds true for servers’ dependency graphs, except that the directed edges there reflect arrival orders instead of execution orders.

Lemma 2 If any \( T_i \rightarrow T_j \) appears in any server’ dependency graph, then \( T_i \) contains a piece \( p_i \) and \( T_j \) contains a piece \( p_j \) such that \( p_i \) and \( p_j \) have conflicting access on certain items and there exists a server \( S_{i,j} \) which has processed the start request of \( p_i \) before \( p_j \).

Proof: This lemma is also obvious according ROCOCO’s protocol specification. The only tricky part is that the server observes \( T_i \rightarrow T_j \) does not has to be \( S_{i,j} \) because the dependency information may be propagated to other servers. Therefore, a dependency information on a server may either comes from tracking or propagating. However, there must be initially some server that establishes this dependency by tracking, and that server is \( S_{i,j} \).

About the relationship between the dependency graph and conflicting pieces, the following property is obvious according to the specification.

Lemma 3 If the start request of two conflicting pieces \( p_i \) (of transaction \( T_i \)) and \( p_j \) (of transaction \( T_j \)) arrive at the same server, and \( p_i \) arrives earlier than \( p_j \), the server should observe \( T_i \rightarrow T_j \) after it processes the start request of \( p_j \). In particular, if \( p_i \) and \( p_j \) are immediate pieces. The server should observe \( T_i \rightarrow T_j \) after it processes the start request of \( p_j \).

Proof: Obvious.

The immediacy propagation in the offline checking ensures the following critical property, which allows us to use a simple way to labelize the edges in both serialization graph and dependency graph.

Lemma 4 For any two conflicting pieces \( p_i \) and \( p_j \) from different transactions, they must both be immediate pieces or both be deferrable pieces.

Proof: This lemma is guaranteed by offline checking. In offline checking, if two pieces may conflict with each other, and one of them is immediate, we should change the other to immediate too.

Lemma 4 allows us to use \( i \) or \( d \) to labelize the edges in the serialization graph as well. According to Lemma 1, we can use \( T_i \rightarrow_i T_j \) to represent the case that the conflicting pieces \( p_i \) and \( p_j \) are immediate pieces, and \( T_i \rightarrow_d T_j \) to represent the case that \( p_i \) and \( p_j \) are both deferrable pieces. We will refer \( \rightarrow_i \) as immediate edge, and \( \rightarrow_d \) as deferrable edge.

Similarly, the dependency tracking labelization is also supported by Lemma 4. We can also use \( T_i \rightarrow_i T_j \) and \( T_i \rightarrow_d T_j \) to imply that the conflicting pieces \( p_i \) and \( p_j \) are immediate pieces or deferrable pieces. Notice that the edges in serialization graph and dependency graph refer to different things, the serialization graph reflects the actual execution orders, and the dependency graph reflects the piece arrival (processing) orders.

Combined with the above ways to represent immediate/deferrable relations, our offline checking also guarantees another important property about immediate relations.

Proposition 1 The serialization graph cannot contain a cycle of transactions all connected by immediate edges.

Proof: We will prove by contradiction. Assume there exists a cycle \( \delta \) in the serialization graph, such that all edges in \( \delta \) are immediate edges. Suppose \( \delta = T_1 \rightarrow_i T_2 \rightarrow_i \cdots \rightarrow_i T_n \). We will prove that the existence of \( \delta \) leads to some contradiction in the offline checking.

1. According to Lemma 1 and Lemma 4, for any \( T_i \rightarrow_i T_j \) in \( \delta \), \( T_i \) contains an immediate piece, which can be denoted by \( p_i' \), \( T_j \) contains an immediate piece, which can be denoted by \( p_j \), such that \( p_i' \) and \( p_j \) conflicts on a server \( S_{i,j} \), and \( p_i' \) is executed before \( p_j \) on \( S_{i,j} \).
2. For \( \delta = T_1 \xrightarrow{i} T_2 \xrightarrow{i} \ldots \xrightarrow{i} T_n \), we can expand it by identifying the conflicting pieces between each pair of transactions. Let \( T_1 \) be represented by \( p_1 \cdot p_1' \), \( T_2 \) represented by \( p_2 \cdot p_2' \), etc. Then \( \delta = p_1 \cdot p_1' \xrightarrow{i} p_2 \cdot p_2' \xrightarrow{i} \ldots \xrightarrow{i} p_n \cdot p_n' \xrightarrow{i} p_1 \cdot p_1' \). To clarify, each “..” in \( p_i \ldots p_i' \) refers nothing else but that \( p_i \) and \( p_i' \) belongs to the same transaction (not that \( p_i \) should execute before \( p_i' \)).

3. There exists at least a pair of transactions \( \{p_i, p_i'\} \) that \( p_i \) and \( p_i' \) are not the same piece, otherwise we will have a cycle in the serialization graph of pieces, which is not possible because each piece is executed atomically at a server (explained in Lemma 1). The pair \( \{p_i, p_i'\} \) suggests an S-edge in SC-graph between \( p_i \) and \( p_i' \).

4. In the cycle \( p_1 \cdot p_1' \xrightarrow{i} p_2 \cdot p_2' \xrightarrow{i} \ldots \xrightarrow{i} p_n \cdot p_n' \xrightarrow{i} p_1 \cdot p_1' \). Each \( p_i \xrightarrow{i} p_j \) suggests an i-typed C-edge in SC-graph.

5. The above C-edges and S-edge(s) form an SC-cycle of which all C-edges are i-typed in SC-graph, which is a contradiction to our offline checking.

6. Q.E.D.

As each server may have only different dependency graph, if we consider the collective dependency graph from all servers, it has similar properties. Because we assume the dependency tracking on each server is also atomic, the collective dependency graph also does not contain any immediate cycle.

**Proposition 2** The collective dependency graph does not contain a cycle of transactions all connected by immediate edges.

**Proof:** Similar to Proposition 1.

To start the proof for serialization, we first show the relationship between the serialization graph and servers’ dependency graphs.

**Proposition 3** For two transactions \( T_i \) and \( T_j \), if \( T_i \xrightarrow{\pi} T_j \) appears in the serialization graph, then there exists a server that is involved in both \( T_i \) and \( T_j \), and has \( T_i \xrightarrow{\pi} T_j \) in its dep before it sets the finish flag of \( T_j \) to true (line 68).

**Proof:** According to Lemma 1, there exists a server \( S_{i,j} \) who executes a conflicting piece \( p_i \) (which belongs to \( T_i \)) before \( p_j \) (which belongs to \( T_j \)). By Lemma 4, \( p_i \) and \( p_j \) both are immediate pieces or both are deferrable pieces. There are two cases w.r.t. the arriving order of the start requests of \( p_i \) and \( p_j \) on server \( S_{i,j} \).

1. **Case:** \( p_i \) arrives earlier than \( p_j \) on \( S_{i,j} \).
   1.1. In this case, according to Lemma 3, server \( S_{i,j} \) will observe \( T_i \xrightarrow{\pi} T_j \) in its dependency graph after processing the start request of \( p_j \).
   1.2. Q.E.D.

2. **Case:** \( p_i \) arrives later than \( p_j \) on \( S_{i,j} \).
   2.1. In this case \( p_i \) and \( p_j \) must both be deferrable pieces, otherwise \( p_i \) and \( p_j \) would be executed according to their arrival order (\( p_j \) before \( p_i \)), a contradiction to the fact that \( p_j \) is executed after \( p_i \).
   2.2. After \( S_{i,j} \) has processed the start request of \( p_i \), according to Lemma 3, \( S_{i,j} \) will observe a \( T_j \xrightarrow{\pi} T_i \) in \( S_{i,j}.dep \), as the status \( T_i \) is STARTED.
   2.3. \( T_i \) and \( T_j \) must be in the same SCC when \( T_i \)’s finish flag is set to true. Otherwise according to specification, the server should wait until it executes \( p_j \) before it can execute \( p_i \), which is against the fact that \( p_i \) is executed earlier than \( p_j \).
   2.4. \( T_i \) and \( T_j \) are in the same SCC suggests that \( S_{i,j} \) must have observe a cycle \( T_i \xrightarrow{\pi} T_j \xrightarrow{\pi} T_i \) in its dep.
   2.5. Q.E.D.

Proposition 3 reveals a fact that a cycle in the serialization graph corresponds to a cycle in the collective dependency graph of all servers. Next, we will show that all servers involved in a cycle of dependency edges are
guaranteed to observe the complete cycle. This is achieved by ROCOCO’s dependency propagation mechanism and by requiring a server to wait for all ancestors of a transaction to become COMMITTING. We recall that dependency information is propagated through the system via three types of messages: coordinators’ commit requests, servers’ replies to start requests, servers’ replies to other servers’ ask requests. We refer to any of these dependency-carrying messages as a dependency message.

The following lemma reveals the relationship between dependency graph and a dependency message that contains some transaction as COMMITTING.

**Lemma 5** For any two transactions $T_i$ and $T_j$, if $T_i \rightarrow T_j$ appears in the dependency graph on some server, $T_i \rightarrow T_j$ should be included in any dependency message that includes $T_j$ as COMMITTING.

**Proof**: We will prove by contradiction. Assume there is a dependency message that contains $T_j$ as COMMITTING but does not contain $T_i \rightarrow T_j$. First, we will demonstrate that the condition of $T_i \rightarrow T_j$ appearing in the dependency graph will lead to a result that $T_i \rightarrow T_j$ must be included in the commit request of $T_j$. Second, we will show the assumption on existence of such a message will lead to a contradiction that such message cannot exists.

1. According to Lemma 2, as $T_i \rightarrow T_j$ appears in the dependency graph, there are two conflicting pieces $p_i$ and $p_j$ that belongs to $T_i$ and $T_j$ respectively. $p_i$ and $p_j$ belong to the same server, and $p_j$ arrives later than $p_i$. In the reply to the start request of $p_i$, $T_i \rightarrow T_j$ will be included. Therefore, the coordinator will also include $T_i \rightarrow T_j$ into the commit request of $T_j$.

2. Without loss of generality, assume the first dependency message that contains $T_j$ as COMMITTING but does not contain $T_i \rightarrow T_j$. By Step 1, $c_0$ cannot be the commit request of $T_j$. So $c_0$ can only be sent out by some server. Let $S$ represent the server. Consider how $S$ has $T_j$ as COMMITTING. $S$ must receive some other dependency message that contains $T_j$ as COMMITTING. Let this message be represented as $c'$. Because $c_0$ is the first message that contains $T_j$ as COMMITTING but not contains $T_i \rightarrow T_j$. So $c'$ has to contain $T_i \rightarrow T_j$. According to the specification $c_0$ should contain $T_i \rightarrow T_j$ too. This is a contradiction.

3. Q.E.D.

The above proposition obviously leads to the following one:

**Proposition 4** If there exists a cycle of transactions, $\theta$: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$, among the collective dependency graphs of all servers, then for any pair transactions $T_i \rightarrow T_j$ in $\theta$, $T_i \rightarrow T_j$ should be present in any dependency message which contains transaction $T_j$ as COMMITTING.

**Proof**: Obvious by Lemma 5.

With these properties at our hand, we can move on to prove ROCOCO is serializable.  

**Theorem 1** ROCOCO is conflict serializable in that it results only in acyclic serialization graph.

**Proof**: We will prove by contradiction. Assume ROCOCO results in an execution schedule whose the serialization graph contains a cycle $\delta = T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$. This cycle suggests there is no serial order to which the execution is equivalent to. The key to proving this is false is to show that the presence of $\delta$ implies a similar cycle, $\theta$, in the collective dependency graph. Every server involved in $\theta$ will reorder $\theta$ following the same deterministic order before executing any deferrable pieces of $\theta$. Given this claim, we can argue that there exists an equivalent serial execution order, which contradicts with the assumption. Such reordering is possible because $\theta$ contains at least one deferrable edge, which is already explained in Proposition 1.

1. Consider any pair of transaction $T_i \rightarrow T_j$ in $\delta$. According to Proposition 3, $T_i \rightarrow T_j$ in $\delta$ corresponds to a path $T_i \rightarrow T_j$ in some server’s dependency graph. Therefore $\delta$ necessarily implies a cycle $\theta$ in the collective dependency graph of all servers. And an immediate edge in $\delta$ suggests an immediate path in $\theta$. That said, if $T_i \rightarrow T_j$ in $\delta$, $\theta$ should contain $T_i \rightarrow T_j$.

---

1This theorem corresponds to Proposition 1 in the original paper.
2. According to Proposition 1 and Proposition 2, both $\delta$ and $\theta$ must contain deferrable edge(s). At least one pair of transactions $T_i$ and $T_j$ exists, such that $T_i \overset{d}{\rightarrow} T_j$ in $\delta$ and $T_j \overset{d}{\rightarrow} T_i$ in $\theta$. The existence of the deferrable edge(s) suggests it is possible to find a serial execution order which is compliant to the execution order of the immediate pieces.

3. For any server that is involved in any transaction of $\theta$, according to the specification, the server must wait until the status of the transaction becoming COMMITTING before executing its deferred pieces. Because the waiting is transitive, plus Proposition 4, the server must observe the entire $\theta$ before it starts to execute any deferred piece of any transaction of $\theta$. Moreover, consider the strongly connected component that $\theta$ belongs to, represented by $\theta^{SCC}$, all servers involved in any transaction of $\theta^{SCC}$ shall observe the same $\theta^{SCC}$.

4. By specification, the server should sort $\theta^{SCC}$ before it executes any deferrable piece of $\theta^{SCC}$. Because each cycle in $\theta^{SCC}$ contains at least a deferrable edge, the sorting is possible and will be compliant to the execution of immediate pieces. Because the input ($\theta^{SCC}$) is the same on each server, the output (sorted order) should also be the same. This order is the equivalent serial execution order, which contradicts the assumption earlier.

5. Q.E.D.

Next we are going to prove ROCCO is strictly serializable by proving it preserves the commit to start ordering in the serialization graph. ²

Theorem 2 ROCCO is strictly conflict serializable in that not only it results in acyclic serialization graph but also it obeys the commit to start ordering. In particular, for two transactions $T_1$ and $T_n$, if $T_1$ starts after $T_n$ commits, then the serialization graph the serialization graph must not contain a path $T_1 \sim T_n$.

Proof: We will prove by contradiction. Assume that for two transactions $T_1$ and $T_n$, $T_1$ starts after $T_n$ commits and the serialization graph contains a path $T_1 \sim T_n$. We can show that the path $T_1 \sim T_n$ in the serialization graph necessarily implies that $T_1$ has already been issued to some server before $T_n$ commits. This contradicts with the assumption that $T_1$ only starts after $T_n$ has committed.

1. Consider any pair of transaction $T_i \sim T_j$ on the path $T_i \sim T_j$ in the serialization graph. By Proposition 3, some server will observe $T_i \sim T_j$ in its dep. So the path $T_1 \sim T_n$ in the serialization graph necessarily implies the existence of another path $T_1 \sim T_n$ in the collective dependency graph of all servers. Let the second path be denoted by $T_1 \sim T_2 \sim \ldots \sim T_n$.

2. By specification, $T_n$ can commit only after its finish flag is set on all involved servers. Consider the procedure how this is done on a server denoted as $S_n$. $S_n$ firstly needs to receive a dependency message including $T_n$ as COMMITTING, which contains a $T_{n-1} \sim T_n$. According to specification (line 52-53), the server then needs to receive a dependency message including $T_{n-1}$ as COMMITTING. Transitivity, $S_n$ needs to receive a dependency message including $T_1$ as COMMITTING before it can set the finish flag of $T_n$. This means the commit request of $T_1$ has been sent out, which is a contradiction because $T_1$ only starts after $T_n$ commits.

3. Q.E.D.

5 Optimized Rococo

The simplified ROCCO is easier to understand and prove, but it is merely practical, mainly because of the rapidly growing dependency graph size when dependency is tracked and the way the graph is exchanged (the whole graph is sent every time). We have two main optimization techniques to reduce the graph size in tracking and propagating to make the protocol more practical. The two techniques are 1) keep a smaller set of dependencies when tracking, 2) include a smaller set of ancestors in the dependency graph when

²This theorem corresponds to Proposition 2 in the original paper.
propagating. The description of these optimizations are shown in our original paper. In this section we will give their specification as pseudocode and then a formal proof about why the optimizations are correct based on the proof on the simplified protocol given in the earlier section.

5.1 Pseudocode

**Algorithm 6:** Server $S::$START_TXN($p$)

45 \hspace{1em} $S$.dep[$p$.owner].status := STARTED
46 \hspace{1em} $S$.TRACK_DEP($p$)
47 // execute an immediate piece in the start phase
48 if $p$.immediate then
49 \hspace{1em} output := EXECUTE($p$)
50 \hspace{1em} dep_info := $S$.SUBGRAPH($T$)
51 reply (dep_info, output)

**Algorithm 7:** Server $S::$COMMIT_TXN($T$, dep_info)

52 \hspace{1em} $S$.dep := DEP_UNION($S$.dep, dep_info)
53 \hspace{1em} \textcolor{red}{\textbf{a}} for each $T'$→$T$ ∈ $S$.dep: $S$.dep[$T'$].status < COMMITTING do
54 \hspace{2em} if $T'$ does not involve $S$ then
55 \hspace{3em} dep' := $S'$.ASK_TXN($T'$) where $S'$ is a server involved in $T'$
56 \hspace{3em} $S$.dep := DEP_UNION($S$.dep, dep')
57 \hspace{2em} wait until $S$.dep[$T'$].status ≥ COMMITTING
58 \hspace{1em} $T^{\text{SCC}}$ := STRONGLY_CONNECTED_COMPONENT($T$, $S$.dep)
59 \hspace{1em} for each $T' \notin T^{\text{SCC}}$ and $T'$→$T$ do
60 \hspace{2em} if $T'$ involves $S$ then
61 \hspace{3em} wait until $T'.finished = true$
62 \hspace{1em} \text{DETERMINISTIC_TOPOLOGICAL_SORT($T^{\text{SCC}}$)}
63 \hspace{1em} for each $T'$ in $T^{\text{SCC}}$ do
64 \hspace{2em} $S$.dep[$T'$].status := DECIDED
65 \hspace{2em} if $T'$ involves $S$ and $T'.finished = false$ then
66 \hspace{3em} for each deferred $p'$ of $T'$ do
67 \hspace{4em} $T'.output := T'.output \cup \text{EXECUTE}(p')$
68 \hspace{3em} $T'.finished := true$
69 reply $T$.output

\textcolor{red}{\textbf{a}}Once the $S$.dep has changed during the loop, the loop condition should be recalculated.

**Algorithm 8:** Server $S::$ASK_TXN($T$)

70 wait until $S$.dep[$T$].status ≥ COMMITTING
71 if $S[T].status ==$ DECIDED then
72 \hspace{1em} dep_info := $T^{\text{SCC}}$
73 else
74 \hspace{1em} dep_info := \{ $T'\rightarrow T$: $T'\rightarrow T$ ∈ $S$.dep and \#$T'' \in T'\rightarrow T$ and $T'' \notin T^{\text{SCC}}$: $S$.dep[$T''$].status = DECIDED or UNKNOWN \}
75 // An extra optimization is setting the status of an ancestor of $T$ in dep_info to UNKNOWN to reduce the size of graph exchanged.
76 reply dep_info
Algorithm 9: Server $S::\text{trackDep}(p)$

```plaintext
77 let $T$ be $p.owner$
78 if $p.immediate$ then
79   for each conflicting $T'$ in $S.dep$ and $T' \not\rightarrow T \in S.dep$ do
80     add $T' \rightarrow T$ into $S.dep$
81 else
82   for each conflicting $T'$ in $S.dep$ and $T' \not\leftarrow T \in S.dep$ do
83     add $T' \leftarrow T$ into $S.dep$
```

5.2 Proof

In proof of the optimizations, we take an incremental approach. Figure 1 shows the proof structure of the simplified protocol. With the given optimizations, all the lemmas and propositions still remain true, but they are not enough prove the theorems (strict-serializability), because new status are introduced in the algorithms. We are going to provide a new proposition about the optimized protocols, and revise the proof for the two theorems based on both previous properties and the new one.

Lemma 3 actually suggests why the optimization of tracking a smaller dependency graph is correct. In the simplified version of ROCOCO, Lemma 3 is very obvious, because the simplified protocol actually provides much stronger guarantee. Instead of a path, it actually adds a directed edge to every pair of conflicting transactions. So the optimization in the tracking actually cuts down the dependency graph to exactly what Lemma 3 needs.

In the proof of the simplified protocol, Proposition 4 is the key to link the cycle in the serialization graph and the propagation of the dependency graph. In the proof of the optimized protocol, Proposition 4 alone is not enough to prove serializability, because we have brought in a new status in the dependency propagation: DECIDED. Therefore we need the following new proposition to explain the property of DECIDED transactions.

**Proposition 5** If there exists a cycle of transactions, $\theta$: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$, among the collective dependency graphs of all servers, then for any $T_i$ in $\theta$, $\theta$ is included in all dependency messages including $T_i$ as DECIDED.

**Proof:** Assume there is a dependency message that contains a transaction of $\theta$ as DECIDED but it does not contain the cycle of $\theta$. We will show this assumption cannot be true because it will lead to a contradiction.

Without loss of generality, assume the first dependency message in the system that contains any transaction of $\theta$ as DECIDED but does not contain $\theta$ is $d_0$. Let the transaction be represented as $T_i$. 

---

Figure 1: Proof structure of the simplified protocol
1. Let $S$ represent the server that sends out $\tilde{d}_0$. Consider how $T_i$ became DECIDED on server $S$. $T_i$ becomes DECIDED either from 1) graph aggregation with another dependency message containing $T_i$ as DECIDED; or 2) the committing procedure that $S$ has all ancestors of $T_i$ as DECIDED.

2. Assume it is the first case: $T_i$ became DECIDED $S$ from graph aggregation. Otherwise, according to our assumption, any dependency message containing $T_i$ as DECIDED before $\tilde{d}_0$ (if any) must contain $\theta$. And then according to specification, $\tilde{d}_0$ should contain $\theta$ too. This contradicts with the assumption that $\tilde{d}_0$ does not contain $\theta$.

3. Assume it is the second case: $S$ turned the status of $T_i$ into DECIDED in the commit procedure. $S$ must have received all the dependency message(s) that contains all transactions in $\theta$ as COMMITTING. According to Proposition 4, $S$ will observe the whole cycle $\theta$ in a strongly connected component before it decides $T_i$. This means that $\tilde{d}_0$ will contain $\theta$, which contradicts with the assumption that $\tilde{d}_0$ does not contain $\theta$.

4. In both cases we have reached contradiction, which means the assumption that $\tilde{d}_0$ does not contain $\theta$ cannot be true.

5. Q.E.D.

With the help of Proposition 5, the revised proof for serializability is listed below.

**Theorem 1** ROCOCO is conflict serializable in that it results only in acyclic serialization graph.

**Proof:** We will prove by contradiction. Assume ROCOCO results in an execution schedule whose serialization graph contains a cycle $\delta = T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$. This cycle suggests there is no serial order to which the execution is equivalent to.

1. Consider any pair of transaction $T_i \rightarrow T_j$ in $\delta$. According to Proposition 3, $T_i \rightarrow T_j$ in $\delta$ corresponds to a path $T_i \rightarrow T_j$ in some server’s dependency graph. Therefore $\delta$ necessarily implies a cycle $\theta$ in the collective dependency graph of all servers. And an immediate edge in $\delta$ suggests an immediate path in $\theta$. That said, if $T_i \rightarrow T_j$ in $\delta$, $\theta$ should contain $T_i \rightarrow T_j$.

2. According to Proposition 1 and Proposition 2, both $\delta$ and $\theta$ must contain deferrable edge(s). At least one pair of transactions $T_i$ and $T_j$ exists, such that $T_i \rightarrow T_j$ in $\delta$ and $T_i \rightarrow T_j$ in $\theta$. The existence of the deferrable edge(s) suggests it is possible to find a serial execution order which is compliant to the execution order of the immediate pieces.

3. For any server that is involved in any transaction of $\theta$, according to the specification, the server must wait until the status of the transaction becoming DECIDED and all its ancestors involved in this server is finished before executing its deferred pieces. There are two ways the transaction becomes DECIDED: through graph aggregation and through the committing procedure.

4. We now discuss by case. If through graph aggregation, then the server will observe the entire $\theta$ (and $\theta^{SCC}$) after aggregation, according to Proposition 5. If through committing procedure, according to Proposition 4, the server must observe the entire $\theta^{SCC}$ before it starts to execute any deferred piece of any transaction of $\theta$. Therefore, in both cases, the server will observe $\theta^{SCC}$ before executing any transaction of $\theta$.

5. By specification, the server should sort $\theta^{SCC}$ before it executes any deferrable piece of $\theta^{SCC}$. Because each cycle in $\theta^{SCC}$ contains at least a deferrable edge, the sorting is possible and will be compliant to the execution of immediate pieces. Because the input ($\theta^{SCC}$) is the same on each server, the output (sorted order) should also be the same. This order is the equivalent serial execution order, which contradicts with the assumption that such order does not exist.

6. Q.E.D.

Below is the revised proof for strict-serializability.
**Theorem 2** Rococo is strictly conflict serializable in that not only it results in acyclic serialization graph but also it obeys the commit to start ordering. In particular, for two transactions $T_1$ and $T_n$, if $T_1$ starts after $T_n$ commits, then the serialization graph the serialization graph must not contain a path $T_1 \leadsto T_n$.

**Proof:** We will prove by contradiction. Assume that for two transactions $T_1$ and $T_n$, $T_1$ starts after $T_n$ commits and the serialization graph contains a path $T_1 \leadsto T_n$. We can show that the path $T_1 \leadsto T_n$ in the serialization graph necessarily implies that $T_1$ has already been issued to some server before $T_n$ commits. This contradicts with the assumption that $T_1$ only starts after $T_n$ has committed.

1. Consider any pair of transaction $T_i \rightarrow T_j$ on the path $T_1 \leadsto T_n$ in the serialization graph. By Proposition 3, some server will observe $T_i \leadsto T_j$ in its dep. So the path $T_1 \leadsto T_n$ in the serialization graph necessarily implies the existence of another path $T_1 \leadsto T_n$ in the collective dependency graph of all servers. Let the second path be denoted by $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$.

2. By specification, $T_n$ can commit only after its finish flag is set on all involved servers. Consider the procedure how this is done on a server denoted as $S_n$. Again there are two cases where $S_n$ turns $T_n$ to DECIDED: through graph aggregation and through committing procedure. If this is done through graph aggregation, it means recursively we can trace to a server that decides $T_n$ through committing procedure. Let this server be denoted as $S'$, where $S'$ could be the same server as $S_n$.

3. Consider how $S'$ commits $T_n$. It needs to receive the message that contains $T_n$ as committing, and we know this message will contain $T_n \rightarrow T_n$ according to Lemma 5. The server $S'$ then needs to transitively collect dependencies for $T_{n-1}$. There are two cases now, the server $S'$ can either receive a message that has $T_{n-1}$ as COMMITTING, or a message that has $T_{n-1}$ as DECIDED.

3.1. If it is the first case, i.e., the message has $T_{n-1}$ as COMMITTING, then the server $S_n$ needs to continue to track for $T_{n-2}$.

3.2. If it is the second case, i.e., the message has $T_{n-1}$ as DECIDED, then let us trace this message again to the first server that turns $T_{n-1}$ to DECIDED through the commit procedure. Let the server be denoted as $S''$. Consider how $S''$ commits $T_{n-1}$, it needs to receive either the message that has $T_{n-1}$ as COMMITTING, and this message must contain $T_{n-2} \rightarrow T_{n-1}$.

4. We can recursive apply the previous step, until we reach some server $S$ that sees $T_1 \rightarrow T_2$. This means $T_1$ must already have started before $T_n$ commits, which contradicts the assumption that $T_1$ starts after $T_n$ commits.

5. Q.E.D.

**References**

