Vol.2020-OS-148 No.8
2020/2/28

# 集約ログ転送法を用いた外部整合的トランザクション機構

堀江 悠樹[1,a)]　梶原 顕伍[1]　川島 英之[2]　建部 修見[1]

**概要**：分散トランザクション処理を実行する際，外部整合性（external consistency， strong 1SR）が求められる．これを実現するために single master 方式では master node でトランザクション処理を行い，その結果を backup node に複製する方式が主である．この方式ではトランザクションを 1 つずつ実行するために性能が犠牲になる．そこで本研究では集約ログ転送法を提案する．集約ログ転送法により master node において，複数の Client から到着するトランザクションを backup node へ一括転送可能にする. さらに Client において複数のトランザクションをまとめて 1 つのトランザクションとして master node に送信し合意形成を行うことを可能にする．提案手法により，分散合意手法における合意形成処理の通信コストを削減し効率的な分散トランザクション処理が可能になる．本研究では提案手法を分散合意手法 Raft と並行性制御法 S2PL を用いた Key-Value Store として設計，実装，評価を行った．

Yuki Horie[1,a)]　Kengo Kajiwara[1]　Hideyuki Kawashima[2]　Osamu Tatebe[1]

***Abstract:*** When performing distributed transaction processing, external consistency (strong 1SR) is required; in order to achieve this, the single master method involves performing transaction processing on the master node and copying the result to the backup node. In this method, performance is sacrificed because transactions are executed one by one, so in this study we propose an aggregate log transfer method. Multiple logs can be transferred to the backup node at a time. In addition, multiple transactions can be transferred together to the master node as a single transaction in the client to achieve a consensus. This reduces communication costs and enables efficient distributed transaction processing. We design the proposed method using the concurrency control method S2PL and distributed concensus management method Raft, and implementated it in C++. The result of evaluation showed that our proposed method had some positive effect.

## 1. はじめに

### 1.1 分散トランザクション

システムを高信頼化するために分散データベースが広く使われている．分散データベースは，同じデータの複製を持つ複数のノードから構成され，全体としては 1 つのデータベースであるかのように振る舞うデータベースシステムである．分散データベースでは，データベースの負荷分散により性能および耐障害性の向上が図れる．このような分散データベースにおいてデータベースの一貫性を保ちながらデータアイテムの書き換えを行うには，分散トランザクション処理が必要になる．分散トランザクションにおいて要求される性質は，Consistency と Isolation である．Consistency とは，あるデータアイテムを複数のプロセスが観測した際の見え方に関する基準であり，トランザクションが正しく状態を遷移させ，データベースの整合性が保たれることを保証する．最も堅牢な Consistency は Linearizability と言われる．Isolation とは複数のトランザクションがデータベースにアクセスした際の，その値の見え方に関する基準であり，トランザクションは他のトランザクションから分離されており複数のトランザクションを並列に実行しても直列に実行したときと同じ結果になることを保証する．最も堅牢な Isolation level は Serializability と言われる．Linearizable かつ Serializable が保証されれば，外部整合性 (external consistency，strong 1SR(Strong one-copy serializability)) が保証されると言われる．外部整合性が保証されれば分散データベースにおいてトランザクションの実行順序と Commit 順序は必ず整合される．本研究では外部整合的である分散トランザクション処理を効率的に実現する方式に関して述べる．

[1] 筑波大学
[2] 慶應義塾大学
[a)] horie@hpcs.cs.tsukuba.ac.jp

### 1.2 研究課題

外部整合性を実現する方式には，大別して single-master 方式と multi-master 方式の 2 つがある．図 1 に例を示す．single-master 方式はデータの更新処理を行うサーバー (master) は 1 つだけ，それ以外のサーバーは master のコピーを持つという方式である．この方式では全てのサーバーにデータの更新が反映されるまで時間が発生する．この例としては Google Spanner[1] が挙げられる．multi-master 方式では master が複数存在し，データの更新処理を行う．そのためデータ更新の時間は single-master 方式に比べて少なくなるが，multi-master 方式の実現にはユーザが発行するトランザクションに deterministic 性，データの同時更新への対処などが要求されるために前者に比べて制約が厳しい．この例としては Ocean Vista[2]，SLOG[3]，STAR[4] が挙げられる．

本研究では single master 方式を取り上げる．システム構成としては，Linearizability を保証するために Raft[5] を用い，Serializability を保証するために S2PL[6] を用いて Key-Value Store を実装する．この構成において性能ボトルネックとなるのは Raft における合意形成処理である，通常の合意形成処理においては，Client からの要求を逐次的に実行する必要がある．この場合，データベースに対して並行アクセスができないため，トランザクション処理のスループットは比較的低くなる．この逐次実行処理を並列化することができれば，性能を向上させることができる．

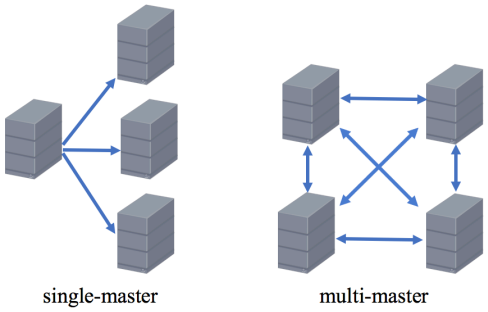


図 1 single-master 方式と multi-master 方式

### 1.3 貢献

本研究では分散合意手法 Raft と並行性制御法 S2PL を用いて外部一貫性のあるトランザクション処理を行える分散 Key-Value Store(KVS) を実装した．実装した KVS において Raft の合意形成に伴う通信コストを改善するために，複数のトランザクションをまとめて送信することで合意形成を同時に行うログ転送グループ化ならびに Client が複数のトランザクションをまとめて 1 つのトランザクションとして一括送信するバルク転送を提案する．Raft においてログ転送グループ化を適用するために，Raft のプロトコルを拡張し Entry に Client の ID を保持させることで複数の Entry をまとめて合意形成処理を行うことを可能にした．

提案手法を実装したシステムで実験を行い，評価を行った結果，ログ転送グループ化を行った場合，Client 数の増加に伴ってトランザクション処理のスループットはログ転送グループ化を行わなかった場合と比べて向上し Client 数 64 台の時に最大で 31.7 倍となった．また，バルク転送を行った場合，Client 数の増加に伴ってトランザクション処理のスループットはバルク転送を行わなかった場合と比べて向上し Client 数 2 台の時に最大で 283 倍となった．したがって提案手法によりトランザクション処理スループットが向上することを示した．

## 2. 背景

### 2.1 分散合意手法

分散合意手法は分散システムにおいて複製されたログの一貫性を保ち，耐障害性を高めるために用いられる手法である．分散合意手法には対処できる障害モデルが定義されている．分散システムにおける代表的な障害モデルには Crash-Recovery 障害モデルと Byzantine 障害モデルがある．Crash-Recovery 障害モデルはあるノードが壊れて任意の時間に復帰するという障害である．この障害に耐性のある分散合意手法は Raft や Paxos[7]，Zab[8] などがある．これらの合意手法は主に Key-Value Store として実装され研究が重ねられている [9][10]．Byzantine 障害モデルはあるノードがアルゴリズムに従わず予期しない振る舞いを示す障害であり最も深刻な障害とされる．この障害に耐性のある分散合意手法は PBFT[11][12]，PoW[13]，PoS[14] などがある．一般的に Byzantine 障害に対処できる分散合意手法は Crash-Recovery 障害に対処できる分散合意手法よりも通信が多く必要となるので非効率的である．その他に広く用いられている単純な分散合意手法として 2-Phase Commit(2PC)[15] が挙げられる．しかし 2PC はネットワークの分断やノードの故障といった障害に対応できず，対処できる障害モデルがない．

Google の提供するデータベースである Spanner[1] や Hadoop[16] といった分散合意手法が用いられる実システムでは Byzantine 障害を想定せず，Crash-Recovery 障害に対処できる分散合意手法を用いていることが多い．なぜなら管理者の統治が効いたシステムであれば，Byzantine 障害の原因となるバグやマルウェアは，デバッグやセキュリティ対策を行うことで防ぐことができるからである．

### 2.2 トランザクション処理

トランザクションとはデータベースを操作する上での不可分な処理のまとまりのことである．トランザクションは ACID 特性を満たさなければならない．トランザクション処理を行うデータベースシステムでは一般的に複数のト

ランザクションを並列に処理する必要がある．安全に並列処理するためには，各トランザクションが他のトランザクションから分離する (Isolation を満たす) 必要がある．並行性制御を実現する一般的な方式の 1 つに，2 Phase Lock のようにロックを用いる方式がある．2 Phase Lock(2PL) とはトランザクション処理でアクセスしなければならないデータにその都度ロックをかけていき，全てのロックを獲得したらロックを解放するという方式である．ロックを獲得していく状態を成長層，ロックを開放していく状態を縮退層という．成長層と縮退層はそれぞれ 1 つずつしか存在できず，成長層と縮退層の順序は変わってはならないという制約がある．この方式により Serializable が保証される．

Strict 2 Phase Lock(S2PL) は 2PL の亜種であり，トランザクションの完了後に write ロックを解放するという方式である．これにより Durability を保証することができる．トランザクションの完了後に read/write ロックを解放するという方式は storong Strict 2 Phase Lock と呼ばれる．

### 2.3 Raft

Raft は Paxos と同等の性能を示しながらも理解しやすさを改善し，実装しやすいよう設計された分散合意手法である．Raft は冗長化のために SMR を用いて複製されたログを管理する．リレーショナルデータベースである CockroachDB[17] や Key-Value Store(KVS) である etcd[18] など実際のシステムに多く用いられている．本章では Raft について述べる．

Raft クラスタは 1 台の Leader と残りの Follower から構成される．Raft は過半数に基づき合意を取るアルゴリズムであるため，クラスタのノード数は奇数とするのが一般的であり原論文では 5 台が最適とされている．例えば，ノード数 5 台と 6 台の Raft クラスタを比べた場合，過半数はノード数 5 台のクラスタの場合場合 3 台，ノード数 6 台のクラスタの場合 4 台であるが，障害許容ノード数はどちらの場合も 2 台である．つまりノード数を偶数にしても故障箇所と過半数が増え，障害許容ノード数はノード数が奇数の場合と変わらないため耐障害性が下がってしまう．

## 3. 設計と実装

本章では実装した外部整合的なトランザクション処理を行う Raft を用いた KVS のアーキテクチャについて述べる．プログラミング言語 C++を用いて実装し，コンパイラには g++8.2.0 を用いた．

### 3.1 提案

#### 3.1.1 外部一貫性のあるトランザクション処理

分散トランザクション処理においてデータベースの一貫性を保ちながらデータの書き換えを行うために外部一貫性が求められる．これを実現するために本研究では single master 方式を用いる．Linearizability を保証するために Raft を用いて，Serializability を保証するために S2PL を用いるシステムを提案する．提案システムにおいて Raft に S2PL を適用するために以下のようにプロトコルを改変した．

- Leader は Client からトランザクションを受け取ったらデータベースのロックを獲得しトランザクションを行い，その後でトランザクションを Log に書き込むようにした
- S2PL のロッキング範囲を拡大し合意形成に成功したらデータベースのロックを解放するようにした
- Log への並行アクセスにより Log 書き込みの際に衝突が起きてトランザクションが失われてしまうことを防ぐために，Log にトランザクションを書き込む際には Log のロックを獲得した後に Log への書き込みを行い，保持しているデータベースのロック情報，Log の情報，Client の情報を紐付けて保存したら Log のロックを解放するようにした

#### 3.1.2 ログ転送グループ化

Raft のナイーブなプロトコルでは複数の Client から命令を同時に受け取った場合合意形成処理各命令毎に逐次的に合意形成処理を行う．この場合，データベースに対して並行アクセスができないため性能の劣化につながる．Paxos ベースの分散合意手法 APUS[19] のプロトコルでは，各 Client に ID を割り当てて Entry に命令情報と一緒に保持している．これにより，各 Client の最後の命令を特定することができるため，同じ命令が複数回実行されてしまうことを防ぎ．複数の Entry をまとめて合意形成処理を行うことを可能にしている.Raft のプロトコルにこの手法を適用することは可能であり，Raft に適用した例は存在しない．本研究ではこれをログ転送グループ化として提案する，ログ転送グループ化により逐次処理による性能劣化問題を解決できると考える．この方式では Client から到着する複数のトランザクションを Leader でバッチ処理することにより，データベースへの並行アクセスを実現する．また Leader ノードから Follower ノードへログを転送する際に他の Client から受け取ったログがあればまとめて転送し，合意処理を行い効率化を図る．ログ転送グループ化を適用するために以下のようにプロトコルを改変した．

- 各 Entry に ClientID を保存するようにし，各 Client の最後の命令を特定できるようにした．
- Client から到着する複数のトランザクションを Leader ノードにおいてバッチ的に処理し，一度にログに書き込みを行うようにした．
- Leader ノードから Follower ノードへログを転送する際に他の Client から受け取ったログがあればまとめて転送し合意処理を行うようにした．

#### 3.1.3 バルク転送

合意形成処理による通信コストを改善するために Client において，複数のトランザクションをまとめて 1 つのトランザクションとして一括送信するバルク転送方式を提案する．バルク転送を行うことで合意形成処理の回数を減らし，通信コストを削減することでトランザクション処理スループットの向上を図る．バルク転送は複数のトランザクションをまとめて送りその中でトランザクションに 1 つでも障害が起きると他の全てのトランザクションに影響を与えてしまうため，耐障害性を犠牲にしている．バルク転送を適用するために以下のようにプロトコルを改変した．

- Client において，複数のトランザクションを指定した個数読み込み，それを 1 つのトランザクションとしてまとめて Leader に送付するようにした．
- Client からバルクによりまとめられたトランザクションを受け取った Leader は，まとめてロックを取るようにした．

### 3.2 トランザクション処理の流れ

図 2 に実装した Raft がログ転送グループ化法を用いて外部一貫性のあるトランザクション処理を行う流れを示す．ナイーブな方式では命令の実行は合意形成後だが，提案方式では S2PL を用いて，Leader は合意形成前にトランザクションを実行し，データベースオブジェクトのロックを獲得し合意形成後にそのロックを開放するという方式とした．S2PL を用いることで，トランザクションの合意形成が終了しロックが開放されるまでするまで競合する他のトランザクションはデータベースの同じ箇所にアクセスできないため Serializability が保証される．また，Raft のプロトコルにより，Leader はトランザクションを Client から受け取った順に実行し，それをログに書き込み複製し，合意が形成できたらログの順序通りに Follower もトランザクションを実行するため全ノードで同じ状態が定まり，Linearizability が保証される．Raft クラスタは次のように命令を処理する．

(1) 複数の Client が Leader にトランザクションを送る
(2) Leader は受け取った複数のトランザクションをバッチ的に処理し実行する．トランザクションはデータベースオブジェクトをロックした後に実行されデータベースオブジェクトの更新が行われる．
(3) Leader は複数のトランザクションを同時に Log に書き込み保存する．Log の書き込みの際には Log にもロックをかけて，データベースオブジェクトのロック情報を保存した後に Log のロックを解放する．
(4) Leader は Log を Follower に送信する. 複数の Log がある場合はまとめて送信する．
(5) Follower は受け取った Log を書き込み保存する．
(6) Follower は Log を書き込んだことを Leader に伝える．
(7) Leader はクラスタの過半数のノードが送信した Log を書き込んだことを確認したら，Log に紐付けられたデータベースオブジェクトのロック情報を用いて，データベースオブジェクトのロックを解放する．
(8) Client に各トランザクションの Commit を伝え，Follower にも Log に書き込んだ複数のトランザクションを適用するよう伝える．
(9) Follower はトランザクションを実行する．

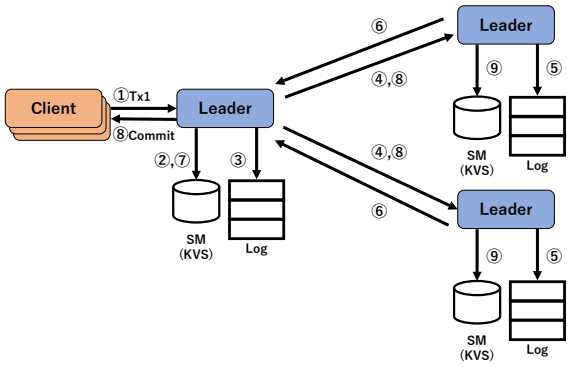


図 2　トランザクション処理の流れ

### 3.3 Raft

本モジュールでは，Raft の全ての挙動を管理する．Raft ノードは起動すると Timer と Receive の 2 つのスレッドを立ち上げる．Timer スレッドではタイムアウトのための時間の計測，Leader の sendAppendEntriesRPC の送信の管理を行う．Receive スレッドでは，Client と他の Raft ノードとの接続を管理し，Raft が他のノードとの接続が確立できた場合には Worker スレッドを立ち上げる．Worker スレッドでは Client や他の Raft ノードとの通信の管理を行い，通信内容を判別して Client からのトランザクションの受信，Log と KVS への書き込み，RPC への応答，選挙の票の管理等を行う．本モジュールはその他にも他のノードの情報，State，Term などを管理している．

### 3.4 Log

本モジュールでは，Log を管理する．Log を構成する LogEntry には Leader がトランザクションを受け取ったときの Term，トランザクションを発行した Client の ID，トランザクションが格納されている．LogEntry には Index は格納されておらず，Log モジュールが各 LogEntry の Index を管理する．Raft モジュールは，Log モジュールを通して各 LogEntry の Index，Term，Client の ID を取得する．Log の実装には，C++言語の std::vector を用いた．Log に新たな LogEnrty を追加する際にはロックを取り，複数スレッドでの競合を防ぐようにした．

### 3.5 KVS

本モジュールでは，KVS を管理する．この KVS は Raft の StateMachine に当たり，Raft は KVS を安全に複製し SMR を実現することが Raft の目的である．KVS の実装には std:map を用いてデータの保存を行うようにした．また，KVS にトランザクションを適用した際はそのトランザクションの合意が取れるまで適用範囲をロックすることで排他制御を行った．KVS の操作として GET，PUT を実装した．

### 3.6 Client

本モジュールでは，Client の全ての挙動を管理する．Client は起動すると，起動の際に与えたファイルからトランザクション群を読み取り，メモリに保存する．その後 Leader にトランザクションを送信し，合意が取れたら次のトランザクションを送信していく．Client は最初に Raft クラスタに接続する際，Leader がわからないのでクラスタを構成する Raft ノードのいずれかにアクセスする．アクセスを受けたノードは Leader を知っていたら Leader のアドレスを返す．このようにして Client は Leader にアクセスする．

## 4. 評価

本章では実装したシステムを用いて外部整合性のあるトランザクション処理の性能を評価する．

### 4.1 実験環境

実験環境を表 1 に示す．実験ではまず，ログ転送グループ化の有無と，トランザクションを Raft に送る Client 数を 1，2，4，8，16，32，64 と増やしてスループットを測定した．スループットは Raft クラスタが単位時間に処理するトランザクションの数とし，1 つのトランザクションに 2 つの read-modify-write を含めた．Raft クラスタのサーバー数は分散合意手法で一般的な台数とされている 5 台とした．グループ転送でトランザクションをまとめて送る最大数は 64 とした．次に，Raft クラスタのサーバー数を 3，5，7，9，11 と増やして同条件で実験を行った．最後に，バルク転送を適用して Client 数を 1，2，4，8，16 と増やしてスループットを測定した．

表 1 実験環境

| | |
|---|---|
| #Nodes | 5 |
| OS | CentOS release 6.10 |
| CPU | Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz |
| #Cores | 24 |
| RAM | 24GB |
| Network | GigabitEthernet |

### 4.2 実験結果と考察

ログ転送グループ化を行わなかった場合と行った場合のスループットの測定結果を図 3，図 4 にそれぞれ示す．Raft クラスタのサーバー数を 3，5，7，9，11 とした場合のスループットの測定結果を図 5 に示す．バルク転送を適用し 100 個のトランザクションを 1 つにまとめて送信する実験を行った場合のスループットの測定結果を図 6 に示す．それぞれの場合のスループットの差異について，考察を行う．

#### 4.2.1 Client 数の変化

ログ転送グループ化を行わなかった場合 (図 3)，Client 数が 1 から 4 の間でスループットが向上した．これは通信の遅延の隠蔽に起因すると考えられる．Client が 1 台の場合は，Leader は Client にコミットメッセージを送り，次の命令を待っている間は新たな処理を進めることができない．しかし Client が複数の場合は，Leader はある Client にコミットメッセージを送り，次の命令を待っている間に他の Client からの命令を受け取って合意処理をすすめることができる．つまり，Client と Leader による通信の遅延が隠蔽される．Client 数が 4 以上になるとスループットは頭打ちとなっているが，これは遅延が隠蔽され尽くしたためだと考えられる．

#### 4.2.2 ログ転送グループ化の有無

ログ転送グループ化を行った場合 (図 4)，Client 数が 1 から 64 の間でスループットが向上した．ログ転送グループ化を行わなかった場合 (図 3) と比べると，スループットはどの Client 数の場合でも向上し，Client 数 64 の時に最大で 31.7 倍となった．これはログ転送グループ化により複数の Client から同時に受け取った命令をまとめて転送し合意処理を行うことで通信のコストが抑えられているからだと考えられる．この結果から Raft の合意処理における通信のコストがいかに大きいかということがわかる．

#### 4.2.3 サーバー数の変化

ログ転送グループ化を行い，Client 数を 64 として Raft クラスタのサーバー数を変化させて実験を行った場合 (図 5)，サーバー数が 3 から 11 と増加させるとスループットは低下した．これはサーバー数の増加により Raft クラスタの Leader が合意をとるのに必要な Follower 数が増えて，通信時間と待ち時間が増加するためだと考えられる．サーバー数を増やすことで性能は劣化するが，Raft クラスタの耐障害性は増加すると考えられる．

#### 4.2.4 バルク転送による効率化

ログ転送グループ化を行い，Client によるバルク転送を適用し Client 数を変化させて実験を行った場合 (図 6)，Client 数が 1 から 16 の間でスループットが向上した．ログ転送グループ化，バルク転送を行わなかった場合 (図 3) と比べるとスループットはどの Client 数の場合でも向上し，Client 数 2 の時に最大で 283 倍, ログ転送グループ化を行いバルク転送を行わなかった場合 (図 4) と比べると，

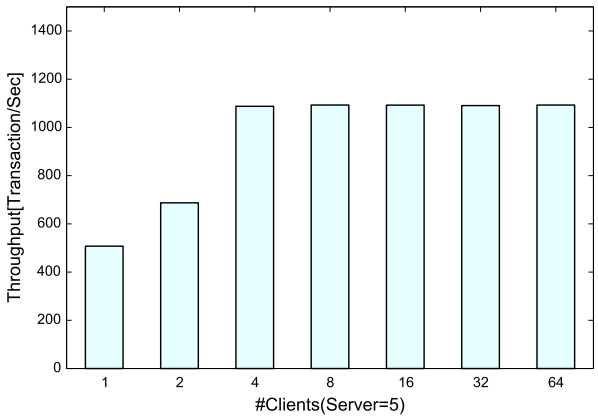


図 3　スループット (ログ転送グループ化無し)

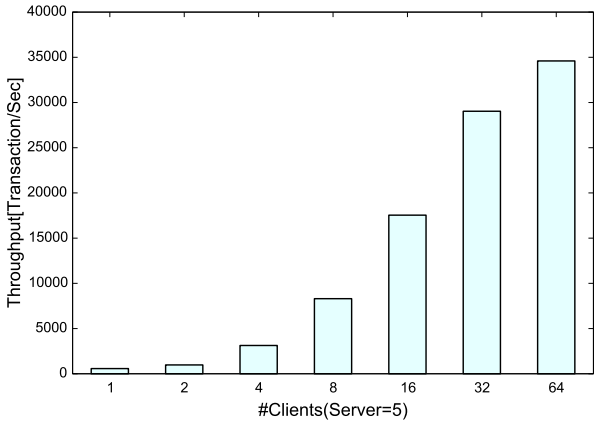


図 4　スループット (ログ転送グループ化有り)

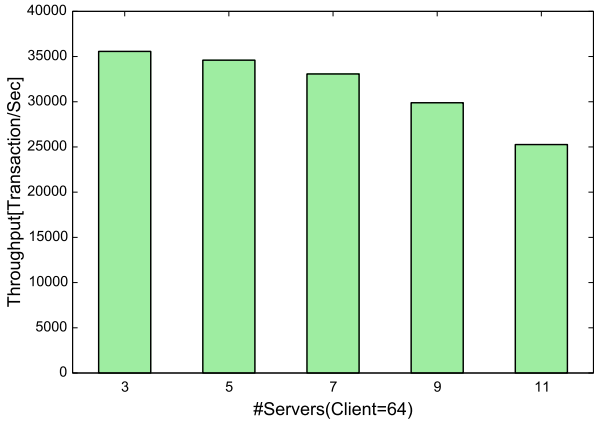


図 5　スループット (サーバー数の変化)

スループットはどの Client 数の場合でも向上し,Client 数 1 の時に最大で 201 倍となった．これはバルク転送により複数のトランザクションをまとめて送付し，合意形成処理を行うことで合意形成処理に伴う通信コストが削減されたからだと考えられる．

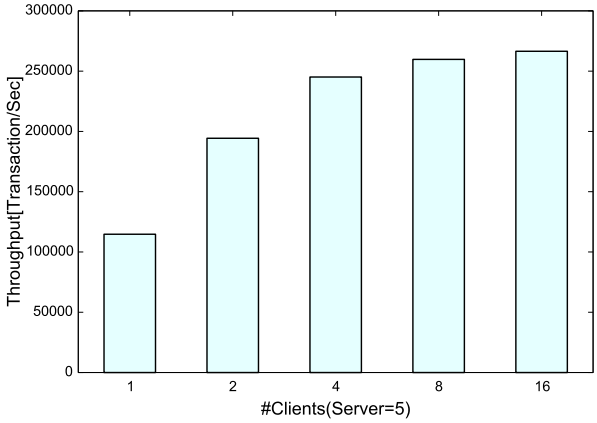


図 6　スループット (バルク転送の適用)

## 5. 関連研究

### 5.1 分散トランザクション

#### 5.1.1 Cloud Spanner

Cloud Spanner[1] は Google が提供しているグローバルに分散化されたリレーショナルデータベースである．Spanner は本研究と同じく single master 方式を採用しており，グローバル規模でデータを分散し，外部整合性のある分散トランザクションを提供している．データの複製には Paxos ベースのプロトコルが用いられており，高可用性と水平スケーラビリティが特徴である．

#### 5.1.2 Ocean Vista

Ocean Vista[2] は multi master 方式を採用したトランザクション処理プロトコルである．Ocean Vista ではデータベース間で water-mark を定期的に交換し合うことで非同期トランザクション処理を可能にしており，グローバルに分散化したデータベースでも通信の遅延を隠蔽し，同時実行制御とデータの複製のコストを抑えられるという特徴がある．

#### 5.1.3 SLOG

SLOG[3] は Spanner と同等の高可用性と水平スケーラビリティを持ちながら，低レイテンシな書き込みを実現する multi master 方式のグローバルに分散化されたデータベースである．これまでグローバルに分散化されたデータベースでは (1)Serializable，(2) 低レイテンシな書き込み，(3) 高スループットなトランザクションの 3 つを全て実現することはできなかったが，SLOG ではアプリケーションワークロードの物理的な局所性を活用することで全てを実現している．

#### 5.1.4 STAR

STAR[4] は multi master 方式を採用し，非対称にレプリケーションを行う分散データベースである．シングルパーティショントランザクションとクロスパーティショントランザクションの実行を行うレプリカが分離されているとい

う特徴がある．これにより通信コストを削減し分散合意手法を用いることなく分散トランザクション処理を行えるため，各トランザクションを効率的に実行することを可能にしている．

### 5.2 分散合意

#### 5.2.1 Paxos

Paxos[7] は全てのオペレーションに順序付けを行いログを配布することで実行順序を統一し，実行結果に不整合が起こらないように設計されたプロトコルである．Paxos は single master 方式を採用しており，過半数の backup から合意が取れれば値を承認する仕組みである．Paxos は合意プロトコルの原点とされ，Paxos をベースにしたプロトコルの研究は盛んに行われている．Paxos のプロトコルは難解とされており，本研究で用いた分散合意手法 Raft は Paxos の難解さを改善することを目標に設計・開発されたものである．

#### 5.2.2 APUS

APUS[19] は RDMA を用いた Paxos ベースの分散合意プロトコルである．RDMA はメッセージ通信に際しての CPU 消費コストを大幅に削減する技法である．APUS では複数 Client からの命令をまとめて処理し，RDMA を用いて通信を行いステートマシンを複製することで高速化を図っている．本研究ではこの手法を用いてトランザクション処理の効率化を図った．APUS は Redis[20] や MySQL[21] といった広く使用されているデータベースシステムに組み込んで性能を測定している．その結果複製しなかった場合と比較して APUS のスループットのオーバーヘッドは 4.3 % のみであった．また ZooKeeper[22][23] や S-Paxos[24] といった 5 つの既存の合意プロトコルとコンセンサスにかかる時間を比較すると，平均して 32.3 倍 APUS のほうが高速であった．

### 5.3 シングルトランザクション

#### 5.3.1 Silo

Silo[25] はロックを用いない楽観的並行実行制御 (OCC(Optimistic Concurrency Contorol)) に基づいてトランザクション処理を行う手法である．Silo のコミットプロトコルでは論理クロックを用いて非集中処理を行うことで共有メモリアクセスを回避し，高いトランザクション性能を示す．TicToc[26] や MOCC[27] 等の他の並行性制御法で Silo のコミットプロトコルが拡張されたものが用いられている．

## 6. 結論

本研究は外部整合性を保証する分散トランザクションシステムを実現するために，分散合意手法 Raft と並行性制御法 S2PL に基づく分散 Key-Value Store を実装し高性能化を目指したものである．効率的な処理を行うために複数のトランザクションを一括で処理するログ転送グループ化ならびに Client が複数のトランザクションをまとめて 1 つのトランザクションとして一括送信するバルク転送を提案し，これを Raft のプロトコルに適用することで性能の向上を図った．

実装したシステムを評価した結果，提案システムは 5 ノードからなる Raft クラスタにおいて，Client 数に対してスループットがスケールすることが観察された．ログ転送グループ化を行った場合とログ転送グループ化を行わなかった場合のスループットを比較すると，どの Client 数の場合でもスループットは向上し，Client 数 64 の時に最大で 31.7 倍となった．また，バルク転送を行った場合とバルク転送を行わなかった場合のスループットを比較すると，どの Client 数の場合でもスループットは向上し，Client 数 2 の時に最大で 283 倍となった．したがって，Raft のプロトコルにログ転送グループ化とバルク転送を適用することで，分散トランザクション処理を効率化することができ，性能の向上につながることを示した．

## acknowledgement

This work is partially supported by JST CREST JPMJCR1303, JPMJCR1414 and KAKENHI JP17H01748, JP19H04117 and project commissioned by NEDO

### 参考文献

[1] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, Vol. 31, No. 3, p. 8, 2013.

[2] Hua Fan and Wojciech Golab. Ocean vista: gossip-based visibility control for speedy geo-distributed transactions. *Proceedings of the VLDB Endowment*, Vol. 12, No. 11, pp. 1471–1484, 2019.

[3] Kun Ren, Dennis Li, and Daniel J Abadi. Slog: serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment*, Vol. 12, No. 11, pp. 1747–1761, 2019.

[4] Yi Lu, Xiangyao Yu, and Samuel Madden. Star: scaling transactions through asymmetric replication. *Proceedings of the VLDB Endowment*, Vol. 12, No. 11, pp. 1316–1329, 2019.

[5] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pp. 305–319, 2014.

[6] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

[7] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, Vol. 21, No. 7, pp. 558–565, July 1978.

[8] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-

backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pp. 245–256. IEEE, 2011.

[9] Amitabha Roy and Subramanya R Dulloor. Cyclone: High availability for persistent key value stores. *arXiv preprint arXiv:1711.06964*, 2017.

[10] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. P axos made transparent. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 105–120. ACM, 2015.

[11] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pp. 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[12] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems*, pp. 363–376. ACM, 2010.

[13] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[14] Andrew Poelstra, et al. Distributed consensus from proof of stake is impossible, 2014.

[15] Muhammad Atif. Analysis and verification of two-phase commit & three-phase commit protocols. In *2009 International Conference on Emerging Technologies*, pp. 326–331. IEEE, 2009.

[16] Apache Hadoop. https://hadoop.apache.org/.

[17] CockroachDB: Ultra-resilient SQL for global business. https://www.cockroachlabs.com.

[18] etcd. https://coreos.com/etcd.

[19] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. Apus: Fast and scalable paxos on rdma. In *Proceedings of the 2017 Symposium on Cloud Computing*, pp. 94–107. ACM, 2017.

[20] Redis. https://redis.io/.

[21] MySQL. https://www.mysql.com/.

[22] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, Vol. 8. Boston, MA, USA, 2010.

[23] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pp. 245–256. IEEE, 2011.

[24] Martin Biely, Zarko Milosevic, Nuno Santos, and Andre Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pp. 111–120. IEEE, 2012.

[25] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 18–32. ACM, 2013.

[26] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, pp. 1629–1642. ACM, 2016.

[27] Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proceedings of the VLDB Endowment*, Vol. 10, No. 2, pp. 49–60, 2016.