Vol.2017-OS-140 No.15
2017/5/17

Raftに基づく分散データベースの性能解析

梶原 顕伍[1,a)]　川島 英之[2]　建部 修見[2]

概要：分散システムにおいて合意を取る手法の 1 つに Raft がある．Raft は分散システムにおける様々な障害に対応できるよう設計された分散合意手法である．Raft ではクライアントの命令一つ一つをストレージに記録し，それを他ノードに配布することでデータの共有を行っている．命令が他ノードのストレージに書き込まれたことを確認してからクライアントにコミットを返すという手順で処理を行っているため，通信やストレージアクセスのレイテンシがボトルネックになってしまう．そこで本研究ではストレージアクセスの回数を減らし，スループットの向上を図るためにバルクログ転送法を提案する．実際に Raft を用いて Key-Value Store を実装し，バルク転送法による性能の変化を測定した結果，従来手法と比較して 2.48 倍の性能向上を観測した．

キーワード：分散合意, Raft, Key Value Store, データシステム

1. 序論

1.1 背景

データの高信頼化を達成する代表的な手法として，複数のノードを利用して分散データシステムを構成し，その上でデータを複製させるアプローチがある．複製を用いることで，分散データシステムにおいてマスターデータを保持しているノードが故障した場合でも，複製データを保持しているノードがサービスを提供することが可能になる．これにより，システム内部の障害をクライアントに気付かれることなく，データ管理サービスは提供され続けることが可能である．複製を管理するには分散して存在する複数のデータオブジェクトの状態について，複数のノードの間で合意を得る必要がある．

1.2 研究課題

分散システムで合意を得る手段の一つに，2014 年に Usenix ATC 会議で報告された Raft [1] がある．Raft は Paxos [2] と同等の性能を有しながらも Paxos よりも格段の理解しやすさを提供する高可用な分散合意プロトコルである．Paxos の理解が極めて困難であるため，Raft 以前には分散合意を行うサービスの構築は極めて難しいとされ，そのようなサービスは Chubby [3] や ZooKpper [4] などしか知られていなかった．一方，Raft に基づくサービスの実装は既に 50 を超えている [5]．Raft に基づく Key-Value store としては etcd [6] が著名であり，そのようなサービスの高性能化は社会的な要請が高い．

そこでまず我々は分散 Key-Value Store を Raft を用いて実装した．これを以後 RKVS と記す．そして RKVS の挙動を仔細に観察した結果，RKVS にボトルネックが存在することを発見した．それはクライアントの命令をログとして逐一ストレージに書いている点である．このストレージへのログ転送は，一つのノードのみならず，複数のノードで実行される．RKVS を含め，KVS のインタフェースには，データを挿入する PUT と，データを検索する GET がある．RKVS に対して PUT が頻繁に実行された場合，このログ転送が致命的に RKVS の性能を劣化させる．従って RKVS の性能を改善するには，ログ転送を高速化する必要がある．

1.3 貢献

RKVS におけるログ転送コストを改善するために，我々はログ転送を一括して行うバルクログ転送法を提案する．バルクログ転送法は複数の PUT 命令により生成される複数のログを一括してストレージへ転送する手法である．RKVS では leader ノードと follower ノードのいずれにおいてもストレージへのログ転送を行う．提案手法は follower において適用される．提案手法を実現するためには，複数のログを 1 つのバッファにまとめあげる機能，バッファの

[1] 筑波大学大学院システム情報工学研究科
Graduate School of Systems and Information Engineering, University of Tsukuba
[2] 筑波大学計算科学研究センター
Center for Computational Sciences, University of Tsukuba
[a)] kajiwara@hpcs.cs.tsukuba.ac.jp

ログを一括して転送する機能，そしてログ転送完了までクライアントへのコミット通知を遅延させる機能が必要である．これらの機能を有する要素を実現し，RKVS へ有機的に結合させた．提案手法を導入した RKVS の性能が，ログ転送を全て省略した理想的な RKVS と同等になることを実験的に示す．

1.4 論文構成

本論文の構成は以下の通りである．第 2 章では Raft について解説する．第 3 章で提案手法について述べる．第 4 章では実装した RKVS の設計について述べる．第 5 章ではバルクログ転送法を組み込んだ RKVS の評価を示す．第 6 章では関連研究を述べる．第 7 章では本研究の結論を述べる．

2. Raft

2.1 Replicated State Machines

耐障害性のある分散環境を構築する際，しばしば Replicated State Machines [7]（以後 RSM と記す）という手法が用いられる．各ノードがそれぞれステートマシンとして機能し，合意によってステートマシンを複製する．RSM による複数ノードの同期の流れを図 1 に示す．以下の順で複数ノードが同期される．

(1) クライアントがクラスタ内の 1 ノードに命令を送る．
(2) 命令を受け取ったノードはローカルのストレージにログとして書き込み，他のノードに命令を送る．他のノードは受け取った命令をローカルのストレージにログとして書き込む．
(3) ログが他ノードに書き込まれたことを確認し，ログから命令を読み，ステートマシンに適用する．他ノードにも適用命令を送り，他ノードにも命令を適用させる．
(4) 命令を受け取ったノードがクライアントに結果を返す．

RSM ではまず，クライアントの命令を受け取った順にストレージに保存する．このログを他ノードに複製し，ログの順番通りに実行させることで，全ノードの状態を一意に定めることができる．一般に利活用されている RSM の例として，Chubby [3] や ZooKeeper [4] などが挙げられる．同様に，Raft においても RSM に従ってノードの状態管理を行う．

2.2 Raft のアーキテクチャ

2.2.1 状態

Raft クラスタのノードには，LEADER, CANDIDATE, FOLLOWER という 3 種類の状態がある．Raft の状態遷移図を図 2 に示す．状態は以下の条件で遷移する．

- プロセス起動時は FOLLOWER である．
- 現在の状態が FOLLOWER もしくは CANDIDATE で，予め決められたタイムアウト時間が経過した場合は CANDIDATE に遷移する，
- 現在の状態が CANDIDATE で，クラスタのノード数の過半数の票を獲得した場合，LEADER に遷移する．
- 現在の状態が CANDIDATE もしくは LEADER で，自分のターム（後述）より大きなタームを持つノードからメッセージを受け取った場合，FOLLOWER に遷移する．

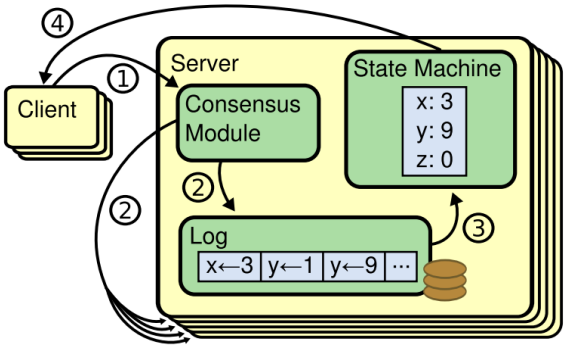

図 1 ログの複製による複数ノードの同期の流れ ([1] より引用)

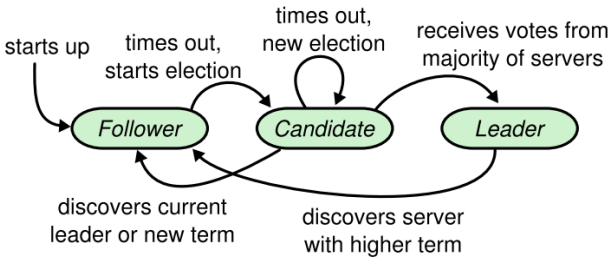

図 2 ノードの状態遷移図（[1] より引用）

LEADER ノードはクラスタ内に 1 台しか存在しない．クライアントは LEADER ノードとのみインタラクションする．クライアントが FOLLOWER ノードにアクセスしてきた場合は LEADER ノードにリダイレクトされる．

FOLLOWER ノードは LEADER ノードに従う．FOLLOWER ノードと LEADER ノードはクライアントから命令を受け取り，コミットするまでを以下のように振舞う．

(1) LEADER ノードはクライアントから更新要求を受信すると，ログエントリを生成する．
(2) LEADER ノードがログエントリをストレージに保存する．
(3) LEADER ノードがログエントリを FOLLOWER ノードへ転送する．
(4) FOLLOWER ノードは受信したログエントリをストレージに保存する．
(5) FOLLOWER ノードが ACK を LEADER ノードへ送信する．
(6) LEADER ノードを含め，Raft クラスタの過半数が命令を受け取り，ログに書き込んだことを LEADER ノードが確認し，LEADER ノードが命令を自分自身のステートマシンに適用する．
(7) クライアントに COMMIT を通知する．

クライアントから命令を受けてコミットするまでの一連の流れを図 3 に示す．図中の各ステップは上で解説した 7

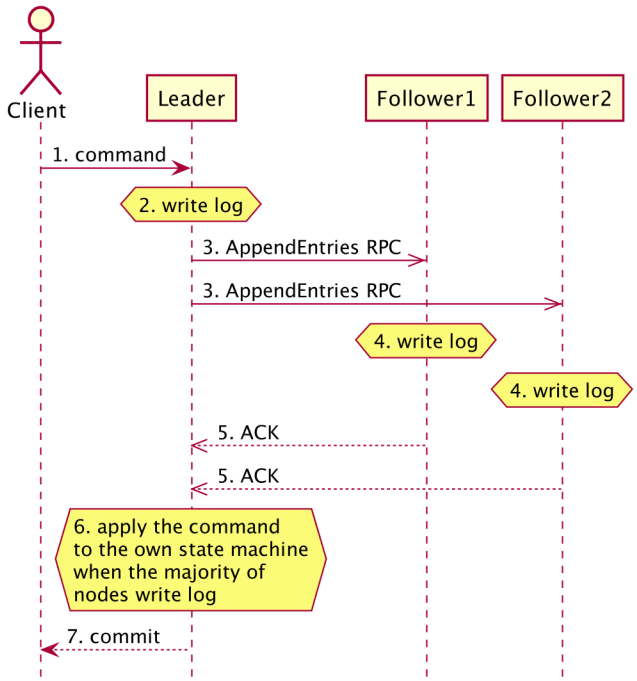

図 3 クライアントの命令をコミットするまでの流れ

ステップに対応している．

その後もしくはクライアントへの COMMIT 通知と同時に，LEADER ノードはログエントリがコミットされたことを FOLLOWER ノードに伝え，FOLLOWER ノードはステートマシンに命令を適用する．最終的には全てのノードが同じ命令を同じ順番で実行するため，全てのノードでデータが複製される．

LEADER ノードが FOLLOWER ノードにログエントリを送信する際，AppendEntriesRPC という RPC（Remote Procedure Call）を用いる．これが LERDER の生存を通知するハートビートを兼ねており，送るべきログエントリが無い場合でも空の AppendEntriesRPC を FOLLOWER ノードに送信する．

LEADER ノードが故障したり，LEADER ノードへのネットワークコネクションが切断されると，FOLLOWER はハートビートを受信できない．このような場合，FOLLOWER ノードはタイムアウトすることによって状態を CANDIDATE へと変更する．CANDIDATE ノードは RequestVote RPC を他ノードに送る．各ノードは RPC の情報を元に CANDIDATE への投票を行い，新しい LEADER を選出する．この選出にはクラスタを構成するノード数の過半数（以後過半数と記す）の投票が必要である．過半数の投票を得たノードが存在しなければ，LEADER 選出作業を再実行する．タイムアウト時間はランダマイズされているので最終的には LEADER は 1 ノードに定まる．

2.2.2 ターム

ネットワークの遅延や，ネットワークの分断と再接続により，新しい LEADER と古い LEADER が同時に存在する場合が考えられる．このような状況においても，FOLLOWER はどちらの LEADER に従うべきなのかを判別しなければならない．

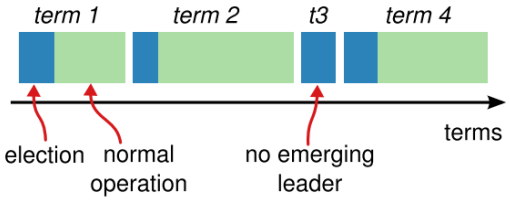

図 4 ターム増加の例（[1] より引用）

そこで，Raft ではタームという概念を導入し，各ノードに自分自身のタームを保持させている．タームは単調増加する整数であり，タームが大きいほど時刻が進んでいるということにすると，FOLLOWER よりタームが小さい LEADER からの AppendEntriesRPC を無視することが可能になる．

具体的にはタームは FOLLOWER のタイムアウト時にインクリメントされる．また，自分よりもタームが大きいノードの存在を検知したときにはそのノードと同じタームになるまでタームがインクリメントされ，自分自身は FOLLOWER となる．ターム増加の例を図 4 に示す．青い部分が LEADER 選出を行っている時間を，緑の部分が LEADER が決まり通常の動作を行っている時間を表す．ターム 1，2，4 はタームの最初に LEADER が決まっているが，ターム 3 は LEADER が決まらずに次のタームに進んでいる．

タームという概念を導入することによって，古い LEADER を通じたクライアントの命令の破棄が可能になる．このように，有効な LEADER はクラスタ内に 1 台だけであるということは保証されている．

2.2.3 ログ

ログエントリの構成は次の通りである．

(1) LEADER がクライアントから命令を受けたときの LEADER のターム
(2) ログ内でのエントリのインデックス
(3) 命令の内容

各ログエントリのタームは LEADER 選出やログの一貫性のチェックに使用される．

ネットワークの遅延等によって図 5 のような状況になることがある．図中の 1 つの箱はログエントリであり，箱内上側の数字はログエントリが生成されたときのタームを，下側の文字列は命令を，ログエントリの横の並びはログを表す．上から，1 つ目は LEADER のログ，2 つ目から 5 つ目は FOLLOWER のログを表している．この図の場合全てのノードが全てのログを保持しているわけではないが，インデックス 7 までのログエントリは過半数のノードが保持しているのでコミットされる．

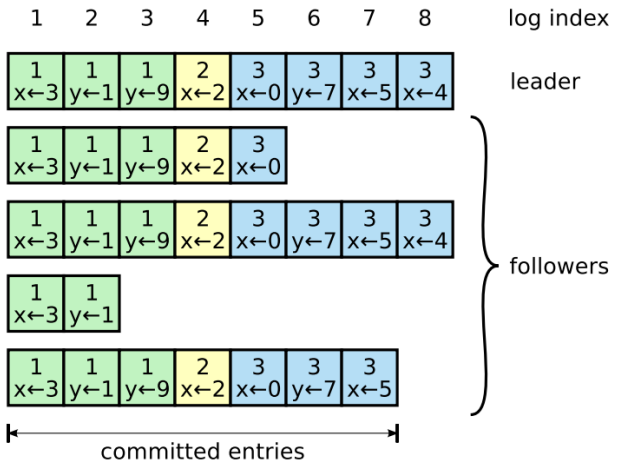

図 5 ログの例（[1] より引用）

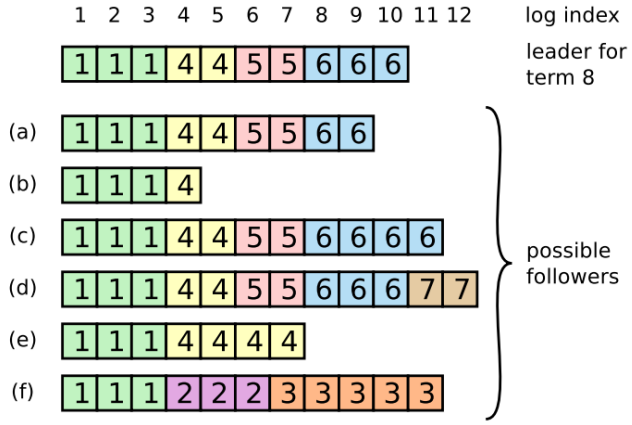

図 6 LEADER クラッシュ後のログの例（[1] より引用）

LEADER ノードは AppendEntries RPC の FOLLOWER ノードからの返答によって，各 FOLLOWER ノードに次に送るべきログエントリの index (nextIndex) を適宜更新している．ログが足りていない FOLLOWER にも LEADER からログエントリを再送するので，最終的には全てのノードが同じログを保持する．nextIndex についての具体的な処理は後述する．

Log Matching Property

異なるノードの 2 つのログについて，Raft では以下の 2 つの性質が成り立つ．

(1) あるインデックスのログエントリのタームが等しいなら，そのログエントリは同じ命令を保持している．
(2) あるインデックスのログエントリのタームが等しいなら，それ以前のログエントリは同一である．

1 つ目の性質は，1 つのインデックスに 1 つのログエントリしか生成されないという事実と，ログエントリはインデックスを移動しないという事実から保証されている．

2 つ目の性質は，AppendEntries RPC による consistency check によって保証されている．具体的には，LEADER ノードは AppendEntries RPC を送信する際に送信するログエントリの 1 つ前のログエントリのインデックスとターム情報を添付して送信する．それを受け取った FOLLOWER ノードは添付された 1 つ前のログエントリのインデックスとタームが自分のログのものと一致するか確認する．もし一致していたら AppendEntries RPC は成功し，ログにエントリが append される．一致しなかったら失敗であり，その旨を LEADER ノードに伝える．

これは帰納的に証明できる．まず，ログが空の状態なら consistency check は満たされている．ログエントリが追加された場合，次回の consistency check は必ず成功する．よって，2 つ目の性質は保証されていると言える．

LEADER クラッシュ時の振舞い

ノードが正常な時の動作では図 5 のようにログの不足という状況しか起こりえないので，異なるログが複数のノードで保持されるということはなく，ログの一貫性は保たれている．しかし，LEADER ノードがクラッシュした場合にはログの一貫性は崩壊してしまう．その例を図 6 に示す．図中の各箱の番号はタームである．(a)，(b) はログが不足している．(c)，(d) はコミットされていない余剰のログが存在する．(e)，(f) はその両方である．

この状況に対応するために，LEADER ノードは nextIndex という値を各 FOLLOWER ノード毎に持っている．LEADER ノードは選出で選ばれて LEADER になった時に，各 FOLLOWER ノードに対する nextIndex の値を（LEADER ノード自身のログの最後のインデックス+1）に設定する．その後，AppendEntries RPC を FOLLOWER に送る．consistency check により，成功か失敗が返ってくる．成功の場合は nextIndex の値をインクリメントして終了する．失敗の場合は nextIndex の値をデクリメントして再送する．FOLLOWER 側では，失敗した場合は失敗したインデックス以降のログエントリを削除する．こうしていると，いつかは AppendEntries RPC が成功する．成功した後は順々にログエントリを追加していくと，LEADER ノードと FOLLOWER ノードのログの一貫性が再び保たれる．

3. Raft に基づく分散 Key-Value Store の高性能化

前章では Raft の合意形成プロトコルについて述べた．本章では Raft を用いて実装した分散 Key-Value Store である RKVS について述べる．

3.1 ログ転送の一括化による高性能化

Raft を用いて構成された分散 Key-Value Store にはボトルネックが存在する．それはクライアントの命令をログとして逐一ストレージに書いている点である．このストレージへのログ転送は，2.2.1 節で述べたように，LEADER

Algorithm 1 バルクログ転送法

Require: $N \geq 1, count \geq 0$, log, entry, file, leaderNode
  log.append(entry)
  $count \Leftarrow count + 1$
  if $count \geq N$ **then**
    file.sync()
    $count \Leftarrow 0$
    send(leaderNode, log.size)
  else
    send(leaderNode, "accept")
  end if

ノードのみならず，FOLLOWER ノードでも実行される．KVS のインタフェースには，データを挿入する PUT と，データを検索する GET ，データを削除する DELETE がある．KVS に対して PUT が頻繁に実行された場合，このログ転送が KVS の性能を致命的に劣化させる可能性がある．従って RKVS の性能を改善するには，ログ転送を高速化する必要があると考えられる．

KVS におけるログ転送コストを改善するために，我々はログ転送を一括して行うバルクログ転送法を提案する．バルクログ転送法は複数の PUT 命令により生成される複数のログを一括してストレージへ転送する手法である．提案手法は FOLLOWER に適用される．提案手法を実現するためには，複数のログを 1 つのバッファにまとめあげる機能，バッファのログを一括して転送する機能，そしてログ転送完了まで，クライアントへのコミット通知を遅延させる機能が必要である．

バルクログ転送法をアルゴリズム 1 に示す．FOLLOWER はエントリを受け取ると log というエントリの配列の最後にエントリを追加し，カウンタの値を 1 増やす．1 度にストレージに書き込むエントリの数 N にカウンタが達した場合，LEADER に log のサイズを伝え，カウンタの値を 0 にリセットする．そうでなければ，エントリを受け取ったことを LEADER に伝える．LEADER は各 FOLLOWER がログをどこまでストレージに書き込んだかが分かる．過半数のノードがクライアントから送られてきた命令をストレージに書き込んだら，LEADER 自身の KVS に命令を適用し，クライアントにコミットを返す．

4. システム構成

プロトタイプとして RKVS Server と Client を作成した．表 1 の環境で実装を行い，ソースコードは全部で 1911 行となった．図 7 に RKVS のシステム構成を示す．RKVS ノードは相互に接続を開始可能である．クライアントノードは RKVS ノードに対してのみ接続開始可能である．クライアントノードは命令のリストを持つ．RKVS ノードでは Raft モジュールが全ての動作を制御している．Log や KVS も Raft モジュールが管理するモジュールの一つである．以後の節で RKVS のモジュールについて解説する．

表 1 開発環境

言語	Java
コンパイラ	javac 1.6.0_45

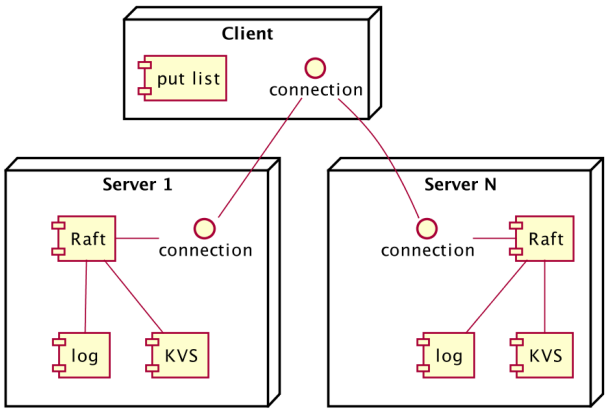

図 7 RKVS の構成

4.1 RKVS Server

4.1.1 Raft

本モジュールは第 2 章で述べた分散合意プロトコル Raft としての処理を管理する．Communicator を呼び出しての Send/Recv，受け取ったメッセージに応じてログの書き込みや，コミット済みのログエントリの KVS への適用，得票数による LEADER への昇格など Raft の全ての機能を行う．また，AppendEntries RPC の送信，RequestVote RPC の送信，時間経過によるタイムアウト処理も本モジュールで管理している．

本モジュールには currentTerm，votedFor，commitIndex，lastApplied というフィールドを持たせている．currentTerm は自分の現在のタームを表す．0 に初期化され，他ノードのタームが currentTerm よりも大きい場合にそのノードと同じタームを currentTerm に代入する．また，タイムアウトして CANDIDATE になる時にはインクリメントされる．votedFor は自分がどのノードに投票したかを保持する．どのノードにも投票していない場合は null を持つ．commitIndex は 0 に初期化され，コミット可能な最後のログエントリの index を保持する．AppendEntries RPC には LEADER ノードの commitIndex (leaderCommit) が付与されている．leaderCommit が自分の commitIndex より大きかった場合，commitIndex に leaderCommit を代入する．lastApplied は 0 に初期化され，最後に KVS に適用したログエントリの index を保持する．commitIndex $\geq$ lastIndex ならば (lastIndex+1) 番目のログエントリを KVS に適用し，lastIndex をインクリメントする．

RaftNode

本モジュールは Raft クラスタの他ノードの情報を管理するモジュールである．ノードのアドレス，ポート番号，前章で解説した nextIndex，自分のログと同じログをどこ

まで持っているかという情報，どこまでログをストレージに書き込んだかという情報，自分に投票したかどうかの情報を持たせている．

ClientNode

本モジュールはクライアントの情報を管理するモジュールである．クライアントノードのアドレス，ポート番号，commit 待ちのログの index (waitIndex) を持たせている．commitIndex ≥ waitIndex ならばクライアントにコミットを返す．

4.1.2 Communicator

本モジュールでは RKVS の他ノード及びクライアントとの通信を管理する．通信が発生する度にコネクションを張り直すのは時間がかかるので，1 度張ったコネクションは使い回すようにした．実際の通信には高い RAS（Reliability Availability Serviceability）を持つ高速インターコネクトである InfiniBand を用いた．本モジュールは複数のスレッドからアクセスされる場合があるので，ロックを取ってから使用するよう実装した．

4.1.3 Log

本モジュールではログの管理を行う．各ログエントリには LEADER がクライアントから命令を受け取った時のタームと命令が格納されている．本モジュールは複数のスレッドからアクセスされる場合があるので Java の標準ライブラリで提供される同期リストを用いた．

また，ボトルネックであるストレージアクセスによる遅延を緩和するために，バルクログ転送を実装した．具体的には，前回の書き込み後から数えてログエントリが一定数溜まったところでストレージへの書き込みを行うことでストレージアクセスの回数を減らした．

4.1.4 KVS

本モジュールでは Key-Value の情報を木構造を用いてメモリ上で管理する．実装には Java の標準ライブラリである TreeMap を使用した．TreeMap は赤黒木 [8] を元にして実装されている．赤黒木への探索，削除，挿入は O($log(n)$) で計算されるので，データ量が大きくなり木が大きくなったとしてもスループットへの影響はほとんど無いと言える．また，KVS のインターフェースとして GET，PUT，DELETE を実装した．

4.2 Client

クライアントノードは RKVS への命令を行う．RKVS のスループットを測定するためにあらかじめ命令のリストを作成し，メモリに読み込んでおく．

すべての PUT 命令を読み込んだ後，RKVS ノードのいずれかのノードをランダムに選んでアクセスする．LEADER ノードではないノードにアクセスした場合，アクセスしたノードが LEADER の場所を知っていた場合には LEADER ノードの場所が返って来るので，その情報を元に LEADER にアクセスする．アクセスしたノードが LEADER の場所を知らない場合，少し待ってから再びランダムにノードを選びアクセスする．時間の経過により LEADER は決まるので上記の処理を繰り返せばいずれは LEADER にアクセスできる．

LEADER にアクセスできたら命令を送信する．Raft の内部処理によって LEADER の KVS に命令が適用され，commit が返ってきたら次の命令を送信する．これを繰り返し，すべての命令がコミットされるまで続ける．

表 2 実験環境

# of Nodes	5
OS	CentOS release 6.8 (Final)
CPU	Intel(R) Xeon(R) CPU E5-2665 0 @ 2.40GHz × 2
# of CPU cores	8 × 2
Memory	64GB

表 3 実験パラメータ 1

# of Clients	1
# of Operations in one group	1
Commands	Only PUT
Key	8Bytes All different
Data Size	Variable

5. 評価

5.1 実験環境

実験を行った環境を表 2 に示す．また，クライアントと RKVS の LEADER との通信の遅延を除去するために LEADER ノードが動いているマシンでクライアントプログラムを走らせ，localhost へアクセスさせた．

5.2 RKVS の評価

5.2.1 データ数による性能の変化

実験内容

RKVS に送るデータ数を変えてスループットを測定した．実験パラメータを表 3 に示す．

実験結果

図 8 に実験結果を示す．横軸が RKVS に送るデータの数，縦軸がスループットを表している．実験結果から，データの数によるスループットへの影響は無いことが観察される．このことから，ログと KVS へのアクセスの遅延はデータ数による影響を受けないと考えられる．

5.2.2 ストレージアクセスを省いた場合との比較

実験内容

本章の冒頭でストレージへのログ転送が性能劣化要因である可能性を指摘した．この仮説を検証するため，LEADER および FOLLOWER におけるストレージへのログ転送（図 3 の第 2,4 ステップ）を省略して RKVS の性能を測定した．この省略により得られる測定結果は理想的な条件下におけ

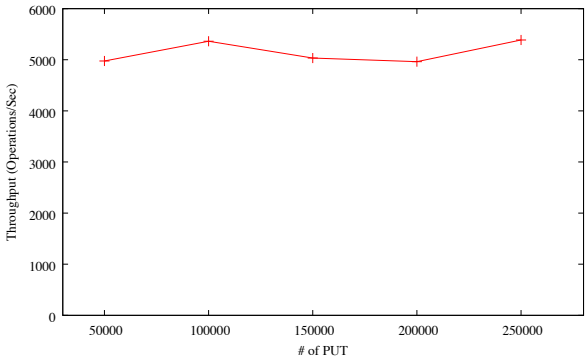

図 8 データ数とスループットの関係

表 4 実験パラメータ 2

# of Clients	1
# of Operations in one group	1
Commands	Only PUT
Key	8Bytes All different
Data Size	100,000

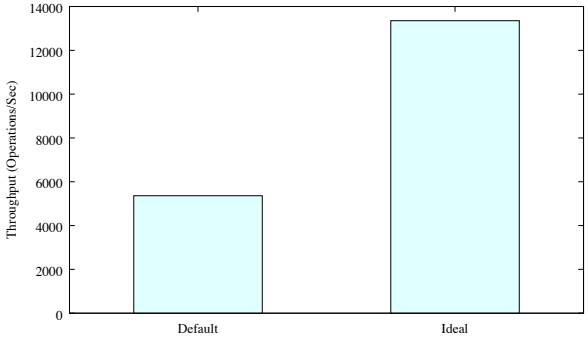

図 9 ストレージ書き込みによる性能の変化

る実験結果を表す．実験パラメータを表 4 に示す．

実験結果

図 9 に実験結果を示す．左のデータが RKVS を測定したものであり，右のデータが RKVS のログのストレージ書き込みを省いて測定したものである．縦軸はスループットを表し，高い程性能が良い．実験結果から，ストレージ書き込みが大きなボトルネックになっていることが読み取れる．これは，ログエントリをクライアントや LEADER から受け取る度にストレージ書き込みを行っているからだと推測される．このボトルネックを解決するにはストレージ書き込みの回数を減らす必要がある．

5.2.3 バルクログ転送法

実験内容

ストレージ書き込みの回数を減らすために，バルクログ転送法を実装し，1 グループ内のログエントリ数を変えてスループットを測定した．表 5 に実験パラメータを示す．

実験結果

図 10，図 11 に実験結果を示す．横軸が 1 グループ内のログエントリ数，縦軸がスループットを表している．1 グループ内のログエントリ数は図 10 は 1〜10，図 11 は 10〜100 である．実験結果から，1 グループ内のログエントリ数が大きくなるにつれてスループットが向上していることが観察される．1 グループ内のログエントリ数が 1 と比較して，10 の場合は 2.1 倍，100 の場合は 2.48 倍の性能向上を観測した．

また，前節で述べた従来手法と，従来手法のストレージアクセスを省いたものとの比較を図 12 に示す．このときの 1 グループ内のログエントリ数は 100 とした．実験結果から，従来手法と比べてバルクログ転送法によって 2.48 倍の性能向上を得られることを観測した．また，ストレージアクセスを省いた場合の 0.994 倍とほぼ同等の性能を観測した．

表 5 実験パラメータ 3

# of Clients	1
# of Operations in one group	Variable
Commands	Only PUT
Key	8Bytes All different
Data Size	100,000

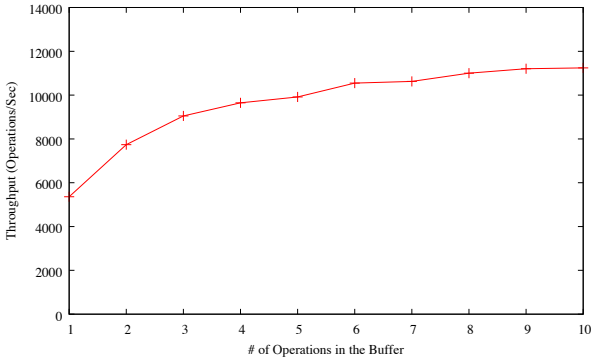

図 10 1 グループ内のログエントリ数による性能向上（パラメータ 1〜10）

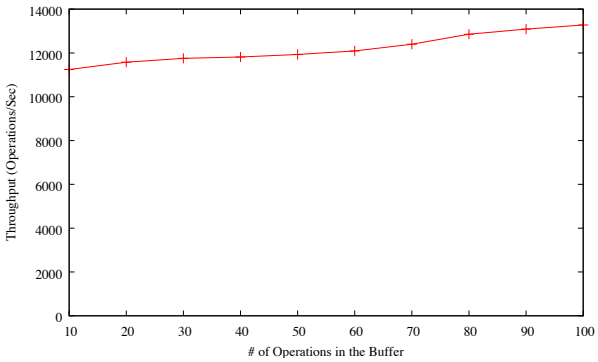

図 11 1 グループ内のログエントリ数による性能向上（パラメータ 10〜100）

6. 関連研究

本章では，Raft 及び他の分散合意プロトコルを高性能化することを課題とした研究を紹介する．

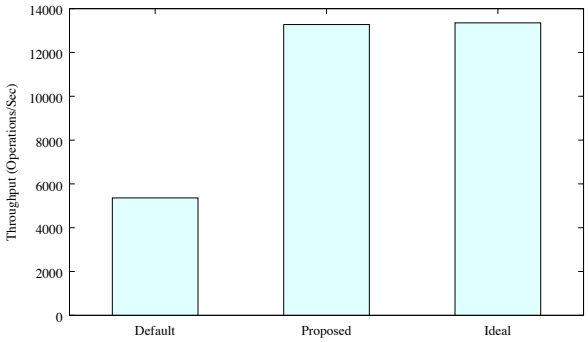

図 12 バルクログ転送法による性能向上

6.1 LEADER 決定の最適化

Raft において，FOLLOWER はランダマイズされたタイムアウト時間が経過すると CANDIDATE になり LEADER 選出を始めると述べた．CANDIDATE もまた，過半数の投票を得られない場合にはタイムアウトする．そのタイムアウト時間を決める際には上限と下限が予め設定されており，限られた範囲の中でランダマイズされる．

元論文 [1] では FOLLOWER と CANDIDATE のタイムアウト時間を同じものとして測定していた．この論文 [9] では CANDIDATE のタイムアウト時間について，あるルールを定めることによって LEADER 決定を高速化する手法を提案している．一つの例として，LEADER になることが許されなかった CANDIDATE は次回のタイムアウト時間を数倍にするといったルールがある．Raft では新しいログを持っていないと投票してもらえないので，このルールが適用された CANDIDATE は一度新しいログを貰わなければリーダーになることができない．従ってこのルールは合理的であると言える．

6.2 DARE

DARE(Direct Access REplication) [10] は，従来の TCP プロトコルや UDP プロトコルを用いた通信ではなく，RDMA(Remote Direct Memory Access) を用いてステートマシンを複製することで高速化を図っている．RDMA とは，あるノードからリモートノードの CPU を介せずに直接メモリを読むことで高速通信を可能にする技術である．

一般に，ステートマシンの複製には通信に起因する遅延がボトルネックになっていることが大半であるので，この手法は有効なアプローチであると言える．実際に DARE[10] では RDMA を用いることで最大 35 倍のスループットの向上に成功している．

6.3 分散データベースシステム

Spanner [11] は Paxos に基づいてデータ複製を行う分散データベースシステムである．Paxos や Raft は Chubby [3] などで見られるようにサイズの小さな構成情報の管理に使われることが多いが，Spanner ではデータ本体の複製管理に Paxos が使われている点が興味深い．Spanner のオープンソース実装として CockroachDB [12] があり，これには Raft が用いられている．

分散データベースシステムにおいて複製オブジェクトの一貫性を取る手法には分散トランザクションがある．Farm [13]，DrTM [14] や RamCloud [15] においてはそのような手法が用いられている．

7. 結論

現代の情報社会においてデータの高信頼化は最重要事項の一つである．これを達成する有効な手法の一つに分散システムにおけるデータ複製がある．データ複製のプロトコルとして著名な手法に Raft がある．分散 Key-Value Store を Raft を用いて実装したシステムである RKVS で性能調査を行った結果，性能ボトルネックを発見した．それはクライアントの命令をログとして逐一ストレージに書いている点であった．このストレージへのログ転送は，LEADER ノードと FOLLOWER ノードの両方で実行される．KVS に対して PUT が頻繁に実行された場合，このログ転送が致命的に RKVS の性能を劣化させることが予見された．

RKVS におけるログ転送コストを改善するために，ログ転送を一括して行うバルクログ転送法を提案した．バルクログ転送法は複数の PUT 命令により生成される複数のログを一括してストレージへ転送する手法である．提案手法を実現するためには，複数のログをひとつのバッファにまとめあげる機能，バッファのログを一括して転送する機能，そしてログ転送完了まで，クライアントへのコミット通知を遅延させる機能が必要だった．そこでこれらの機能を有する要素を実現し，RKVS へ有機的に結合させた．提案手法を導入した RKVS の性能はログ転送を全て省略した理想的な RKVS と同等になる実験結果が観察された．

謝辞 本研究の一部は，JST CREST「ポストペタスケールデータインテンシブサイエンスのためのシステムソフトウェア」，JST CREST「EBD：次世代の年ヨッタバイト処理に向けたエクストリームビッグデータの基盤技術」，JST CREST「広域撮像探査観測のビッグデータ分析による統計計算宇宙物理学」，科研費「#16K00150」による．

参考文献

[1] Ongaro, D. and Ousterhout, J.: In Search of an Understandable Consensus Algorithm, *USENIX Annual Technical Conference*, pp. 305–320 (2014).
[2] Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM*, Vol. 21, No. 7, pp. 558–565 (1978).
[3] Burrows, M.: The Chubby lock service for loosely-coupled distributed systems, OSDI'06, Symposium on Operating Systems Design and Implementation(2006), USENIX, pp. 335–350 (2006).

[4] Hunt, P., Konar, M., Junqueira, F. P. and Reed, B.: ZooKeeper: Wait-free Coordination for Internet-scale Systems, *Proceedings of the USENIX Annual Technical Conference*, pp. 145–158 (2010).
[5] The Raft Consensus Algorithm: Raft Implementations (2017). [Accessed 2017-01-28].
[6] Philips, B.: etcd 2.3.7 Documentation (2016). [Accessed 2016-10-25].
[7] Schneider, F. B.: Implementing fault-tolerant services using the state machine approach: a tutorial, *ACM Computing Surveys 22, 4 (Dec. 1990)*, pp. 299–319 (1990).
[8] 鴨 浩 靖：http://taurus.ics.nara-wu.ac.jp/algo/rbtree.pdf (2013). [Accessed 2017-01-24].
[9] Howard, H., Schwarzkopf, M., Madhavapeddy, A. and Crowcroft, J.: Raft Refloated: Do We Have Consensus?, *ACM SIGOPS Operating Systems Review - Special Issue on Repeatability and Sharing of Experimental Artifacts*, Vol. 49, No. 1, pp. 12–21 (2015).
[10] Poke, M. and Hoefler, T.: DARE: High-Performance State Machine Replication on RDMA Networks, *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC'15)*, ACM, pp. 107–118 (2015).
[11] *Spanner: Google's Globally-Distributed Database* (2012).
[12] CockroachDB: CockroachDB (2017). [Accessed 2017-01-28].
[13] *No Compromises: Distributed Transactions with Consistency, Availability, and Performance*, ACM (2015).
[14] Wei, X., Shi, J., Chen, Y., Chen, R. and Chen, H.: Fast In-memory Transaction Processing Using RDMA and HTM, *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, New York, NY, USA, ACM, pp. 87–104 (online), DOI: 10.1145/2815400.2815419 (2015).
[15] Lee, C., Park, S. J., Kejriwal, A., Matsushita, S. and Ousterhout, J.: Implementing Linearizability at Large Scale and Low Latency, *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, New York, NY, USA, ACM, pp. 71–86 (online), DOI: 10.1145/2815400.2815416 (2015).