Vol.2011-OS-118 No.16
2011/7/28

分散キーバリューストアを対象とした オブストラクションフリートランザクションの実装

熊崎宏樹　津邑公暁
齋藤彰一　松尾啓志

Web サービスを支えるミドルウェアとして広く用いられている分散キーバリューストアでは一般に単一のキーバリューペアのみに対しての操作しかサポートされておらず，トランザクションがサポートされていない事が一般的である．本研究ではそれらの分散キーバリューストアを対象として，ダイナミックソフトウェアトランザクショナルメモリを応用したトランザクションを実装し，評価を行った．

Implementation of Obstruction-free Transaction on Distributed Key Value Store

Hiroki Kumazaki , Tomoaki Tsumura ,
Shoichi Saito and Hiroshi Matsuo

General key-value stores support queries treating only one key-value pair, and don't support transacion. In this paper, we implement and evaluate transaction with serializable consistency on such key-value store.

1. はじめに

web サービスの裏で日夜膨大になるアクセス数に対処するために分散キーバリューストアが広く使われている．本論文ではそれらのキーバリューストアでは不可能だったトランザクション操作を，既存のキーバリューストアの機能のみを用いて実現する方法を提案する．

†1 名古屋工業大学
Nagoya Institute of Technology

キーバリューストアとはバイナリ列であるキーとバリューを対応付けて保存するデータストアの総称である．その操作の単純さにより高い性能を得られるが，同時に 1 つのキーバリューペアでしかデータを扱うことが出来ないため，複数のキーバリューペア間での論理的な依存関係を維持する事は難しい．

キーバリューストアはその単純さの為，分散ハッシュなど[1] を用いて複数の計算機へ負荷を分散することで性能を高める事ができる（スケールアウト）．web サービスの負荷は大きく変動する上，事前の見積もりが難しいため，負荷の量に応じて計算機の台数を変動させることで性能とハードウェアコストのトレードオフを調節可能な分散キーバリューストアが広く使われている．

分散キーバリューストアには大きく分けて，揮発性のデータストアを用いた実装と不揮発性のそれを用いた実装があり，前者ではメモリのみを使ったキーバリューストアである memcached[2] が広く使われている．後者ではハードディスクなどの二次記憶装置をストレージとして利用する Cassandra[3] が注目を集めている．

memcached プロトコル[4] には 10 種類以上のコマンドが定義されているが，本論文の中で利用するプロトコルのみを以下に示す．なお本論文で提案する手法は不揮発性のキーバリューストアを前提としている．

set　指定したキーバリューペアを保存する．

add　指定したキーバリューペアが存在していない場合に限り保存する．既に存在した場合には NOT_STORED が返る．

get　指定したキーバリューペアを獲得する．そのキーが保存されていない場合には END が返る．

gets　get の戻り値と同時に 64bit 長のユニークな値が返る．この数値は各キーごとに固有で，対応するバリューが更新される毎に異なる値に設定される．

cas　保存したいキーバリューペアに加えて gets で得たユニークな値を同時に指定し，それが保存されているキーの現行のユニークな値と一致した場合に限りキーバリューペアに保存される．cas という名前は CPU の命令の一つである compare and swap に由来している．前回獲得した値から書き変わっていない事を確認した上での保存を行えるため，キーバリューペアのアトミックな更新が実現できる．

delete　指定したキーバリューペアを削除する．そのキーが存在しない場合は NOT_FOUND が返る．

2. キーバリューストアの問題点

キーバリューストアを利用する上でいくつか問題点がある．

2.1 一 貫 性

キーバリューストアは 1 つのクエリーで 1 つのキーバリューペアにしか操作できない．そのため複数箇所を書き換える際には書き換え途中のキーバリューペアが他のクライアントから参照・変更された場合の一貫性を保証できない．

一般に分散環境で一貫性を維持するにはトランザクションを用いるが，一般的な web サービスの構成ではデータを保存するサーバの台数よりもユーザーのアクセスを受け付けるアプリケーションサーバの台数が遥かに多いため，アプリケーションサーバーの追加・離脱を考慮するべきである．しかしその場合にいくつか問題点がある．

2.2 耐 障 害 性

ロックベースなトランザクションではクライアントが離脱した際にロックを抱え込んだまま処理が進まない状況に陥る事を回避するため，タイムアウトによるアンロックを行う 3 相コミットを行う必要がある．しかしその複雑さはスケーラビリティを獲得する際の問題となる可能性が高いと考える．

2.3 優先度の反転

アプリケーションサーバに用いる計算機は増減するため，必ずしも均一な性能の計算機ばかりを使うわけではない．しかし高性能な計算機と低性能な計算機で同時にロックベースなトランザクションを使った場合，フェアな条件下では低性能な計算機が相対的により長くの時間ロックを獲得する可能性がある．これは低性能な計算機によって全体のスループットが低下してしまうという事である．

3. 提 案 手 法

不揮発性の分散キーバリューストアを用いてトランザクションを実現する方法を示す．

3.1 トランザクションの特性

提案手法によって実現されるトランザクションの特性について説明する．

3.1.1 トランザクションの分離

複数のキーバリューペアに対する一連の操作の単位をトランザクションと呼ぶ．複数のトランザクションは並列して実行されるが，操作はトランザクション間で完全に分離され，最終的にはすべてが直列に実行されたかのような結果だけが残る．これは serializable 分離レベルと呼ばれる．

3.1.2 オブストラクションフリー

トランザクションはオブストラクションフリー[5] に実行される．この特性ではどのトランザクションも単体に孤立させた上で充分な時間実行した場合に必ず完了する．

また任意のクライアントがいかなるタイミングで遅延・離脱した場合でも全体をデッドロックさせることなく進行する．しかし 2 つ以上のトランザクションが衝突した場合には全体での進行が保証されない場合がある．

3.2 手法の概略

ここでは提案手法の概略を示す．提案手法は DSTM[6] の作法をキーバリューストアに応用したものである．ここではトランザクションを用いて保護されるキーバリューペアをトランザクショナルキーバリューペアと呼ぶ．トランザクショナルキーバリューペアに対して実行可能な操作はオープン, 保存, 取得のみとする．任意のキーバリューペアをカプセル化し、トランザクショナルキーバリューペアとして扱う事によってトランザクションを実現する．

トランザクショナルキーバリューペアは，それぞれがカプセル化対象のデータとは別にトランザクションの ID を一つ保持する．保持されている ID はトランザクショナルキーバリューペアの持ち主を表す物で，後からそのキーバリューペアをオープンするトランザクションは現行の持ち主の ID を参照することで必ず適切なバージョンのデータにアクセスすることが可能となる．

3.2.1 トランザクショナルキーバリューペアの実体

どのようにしてキーバリューペアをカプセル化するかについて説明する． カプセル化する前のデータ構造を図 1 に示す．これをカプセル化することで図 2 という形でキーバリューストア内に保存される．矢印は常にキーに対するバリューの対応を表している点に注意が

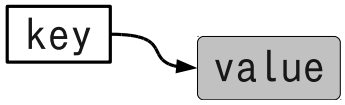

図 1 カプセル化前のキーバリューペア

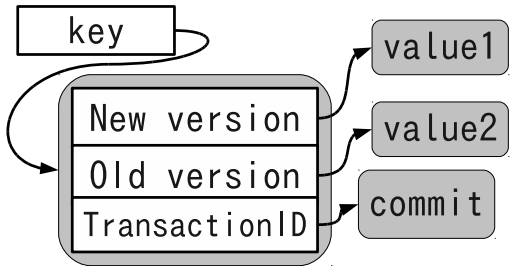

図 2 カプセル化後のキーバリューペア

必要である．つまり図 2 での New value と Old value と Transaction などはキーに対するバリューとして新たなキーが 3 つ保存されているという事を意味する．以後，キーが 3 つ入ったこのバリューをロケータと定義する．

3.2.2 ロケータの実体

ロケータの持つ New value・Old value・Transaction という識別子は具体的なキー名としての文字列ではなく，実際にはランダムな文字列である．以後の図では 3 段になっている場合，図 2 での表記と対応する．

ロケータの持つ 3 つ目の識別子である TransactionID はそのキーバリューペアへのアクセス権を持つトランザクションの ID を示す．キーの持つ値にアクセスする際にはこの ID を参照して適切なバージョンを選ぶ必要があり，書き換えはアクセス権を所有したトランザクションからでないと不可能とする．アクセス権を獲得する操作を以後オープン操作と呼ぶ．

3.2.3 トランザクションステータス

トランザクションステータスはトランザクション ID をキーとして保持される value であり，commited,abort,active の 3 つの値のうちいずれかをとる．カプセル化されているキーバリューペアの値を読み取る際には必ずトランザクションステータスを参照してロケータから適切なバージョンのデータを読み出す必要がある．ステータス値に対応する読み出し方は以下の通りである．

commited: トランザクションが操作を完了している事を意味し，キーバリューペアを読み出す際には New version の値が最新である．

abort:トランザクションが操作を中断させられた事を意味し Old version の値が最新である．

active:キーバリューペアは更新を為されている最中であることを意味するため，アクセスする前にトランザクションの衝突を解決する必要がある．

トランザクションステータスは生成される時は必ず active 状態であり，遷移時は必ず cas コマンドを用いて不可分に書き換えが行われる．commited もしくは active 状態になった後には二度とステータスは遷移しない．起こりうる全てのステータスの遷移を図 3 に示す．

3.2.4 トランザクションの調停

複数のトランザクションが同時刻に同じトランザクショナルキーバリューペアにアクセスした場合，調停を行ってトランザクションの挙動を制御し，最終的にどちらのトランザクションがアクセスする権利を獲得するか決定する必要がある．調停では後からオープンしようとしたトランザクションが待機するか，先にオープンしていたトランザクションを abort

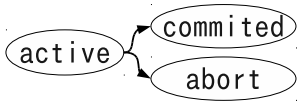

図 3 トランザクションステータスの遷移

状態にするかを選ぶ．究極的にはこの選択は乱数を用いてもトランザクションは可能だが性能を高めるためには効率のよい調停方法を用いる必要がある．本実装では上限値付きバックオフ待機を用いた．

3.3 トランザクションの実装

ここでは提案手法を用いてトランザクションを実行する方法を説明する．手順の擬似コードを以下に示す．

トランザクションの利用例

```
retry:
  try{
    id = beginTransaction(); // トランザクションを開始
    open(id, "key1"); // key1 をオープン
    open(id, "key2"); // key2 をオープン
    v1 = get(id, "key1"); // key1 が保持する最新の値を獲得
    set(id, "key2", v1 + 1); // その値に 1 を足した結果を key2 に保存
    commit(id); // トランザクションをコミット。
  }catch(abortException){
    goto retry; // アボートされた場合，やり直す
  }
```

トランザクショナルキーバリューペアにアクセスする際はまずトランザクション ID を引数に添えて open メソッドを呼ぶ事でアクセス権を自トランザクションへ移す．abort された場合には abortException を発生させ，トランザクションを初めからやり直す．以後では各操作の実装について示す．

トランザクションを開始する

```
1: key random_add(value){ // ランダムなキーを付加して保存する関数
2:   retry:
```

```
 3:     key_name = random_string();
 4:     if(kvs.add(key,value) == SUCCESS) return key_name;
 5:     goto retry;
 6: }
 7: void beginTransaction(){
 8:   myTransactionID = kvs.random_add(active);
 9:   return myTransactionID;
10: }
```

クライアントはまずトランザクションを開始する際に自身のステータスを active 状態で add する (8 行目)．これにより他のクライアントに対し，自分が作業中であることを示す．

トランザクショナルキーバリューペアのオープン操作

```
11: void open(myTransactionID, key){
12: retry: // 再開用ラベル
13:   locator, unique = kvs.gets(key); // キーとユニーク値を同時に獲得
14:   old, new, owner = locator.split(); // 3 つの文字列に分割
15:   if(owner == myTransactionID) // 既に自分がアクセス権を獲得していたら
16:     return;
17:   else{ // 他者が所有権を保持しているなら
18:     ownersStatus, ownersUnique = kvs.gets(owner);
19:     switch(ownersStatus){
20:       case(commited):
21:         nextOldkey = new;  // 新しいバージョンを最新値として採用
22:         break;
23:       case(abort):
24:         nextOldkey = old;  // 古いバージョンを最新値として採用
25:         break;
26:       case(active):
27:         // 衝突している時は競合を解決
28:         resolve_contention(myTransactionID, owner,ownersUnique);
29:         goto retry; // 初めからやりなおし
30:     }
31:     currentValue = kvs.get(nextOldkey);
32:     clonedKey = kvs.random_add(currentValue); // 最新の値を複製
33:
34:     // 最新の値を保持しつつロケータが自分を指すよう CAS コマンドを発行する
35:     result = kvs.cas(key, unique,
36:                      [nextOldkey,clonedKey,myTransactionID]);
37:     if(result == FAILED){
38:       kvs.delete(clonedKey);
39:       goto retry: // 失敗したらやりなおし
40:     }
41:     return; // ロケータの書き換えに成功
42:   }
```

トランザクショナルキーバリューペアを操作する前にキーバリューペアをオープンする．その際にロケータの値を書き換える必要がある場合に備え gets コマンドでロケータと同時にユニーク値も獲得する (13 行目)．獲得したロケータの中には new,old,owner のキー文字列が格納されている．オープン操作とは 3.2.2 に述べたように，所有権を自分に移す操作なので、既に自分に所有権が移っている場合には何もしない (16 行目)．所有権が他者にあった場合 (17 行目) には，その他者のステータス値を gets で獲得する (18 行目)．ここからの操作は 3.2.3 にて示した通り，ステータスが commited 状態ならば新しいバージョンを，abort 状態ならば古いバージョンを獲得し (20〜25 行目)，新しいロケータ上での old な値として採用するためである．もしステータスが active であれば (26 行目) 他のトランザクションが現在利用中であることを意味するので競合を解決する (28 行目)．　値を獲得できた場合，その値を自分の編集のため複製し (32 行目)，cas コマンドを発行しアトミックにロケータを書き換える．もし失敗した場合は既に他のクライアントによってロケータが書き変えられている事を意味するため，複製したキーバリューペアを削除し (38 行目) オープン操作を初めからやり直す．commited と abort のそれぞれの状態のキーバリューペアをオープンする際のロケータの遷移を図 4 と図 5 に示す．

衝突時に呼び出される競合解決操作

```
43: void resolve_contention(myID,otherID,otherUnique) {
```

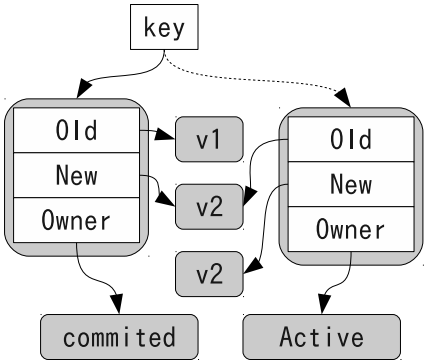

図 4　commited 状態のキーバリューペアをオープン

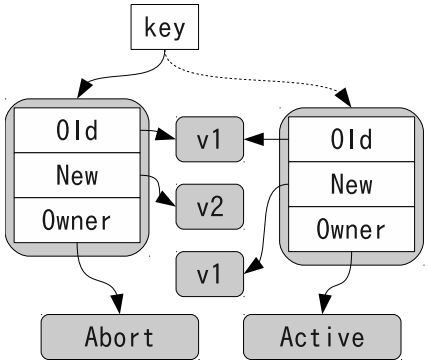

図 5　abort 状態のキーバリューペアをオープン

```
44:   waitMax = waitMap[myTransactionID]; // これが何度目の遭遇なのか調べる
45:   if(wait_max < BACKOFF_MAX){ // 待機時間が上限に達していないならば
46:     sleep(random(wait_max)); // バックオフ上限内でランダム時間待機する
47:     // 待機時間の上限を引き上げる
48:     waitMap[myTransactionID] = wait_max * 2;
49:   } else{ // 既に上限以上の待機時間になっていたら
50:     // 相手のトランザクションを強制的に abort させる
51:     kvs.cas(otherID,otherUnique, abort);
52:   }
53: }
```

衝突した場合は何らかの方法で競合を解決する．本実装ではバックオフ待機を用いる．この衝突がこのトランザクションに取って一定の回数以内であれば，回数に応じた上限以内での乱数時間で待機し (46 行目)，回数を超えていれば相手のトランザクションの状態を abort へ遷移させる (51 行目)．この abort 操作が成功した場合にはリトライ時にロケータを書き換えられる見込みがあり，失敗した場合も対象のトランザクションが commit 操作を完了，もしくは第三者のトランザクションが対象のトランザクションを abort へ遷移させたかのどちらかなので，リトライ時にロケータを書き換えられる可能性があるため，この関数を呼び出した後はリトライする (29 行目)．

open 済のトランザクショナルキーバリューペアへの操作

```
54: value get(myTransactionID, key){
55:   locator = kvs.get(key);
56:   old, new owner = locator.split();
57:   if(owner == myTransactionID) // 既に自分がアクセス権を獲得していたら
58:     return kvs.get(new); // 最新の値を読み出す
59:   else // 他者が所有権を保持しているなら
60:     throw abortException; // 自身は abort されている事を意味する
61: }
62: void set(myTransactionID, key, Value){
63:   locator = kvs.get(key);
64:   old, new owner = locator.split();
65:   if(owner == myTransactionID) // 既に自分がアクセス権を獲得していたら
66:     kvs.set(new, value); // 上書き
67:   else // 他者が所有権を保持しているなら
68:     throw abortException; // 自身は abort されている事を意味する
69: }
```

値を読み書きする際は，予めオープン操作を行いアクセス権を自分へ移してから行わなくてはならない．それにも関わらずトランザクショナルキーバリューペアが自分のトランザクションを指していない場合 (59,67 行目) は他のクライアントにより abort された場合しか有り得ないためトランザクションのやり直しを促す (60,68 行目)．正しく自身のトランザクション ID を指している場合 (57,65 行目) は new が指しているバージョンに対して読み書きを行う (58,66 行目)．

トランザクションのコミット操作

```
70: void commit(myTransactionID){
71:   status, unique = kvs.gets(myTransactionID);
72:   if(status == abort) // 既に他のクライアントによって状態を変えられている
73:     throw abortException
74:   result = kvs.cas(myTransactionID, unique, commited);
75:   if(result == SUCCESS)
```

```
76:     return;
77:   else // 他のクライアントにより abort されたならば
78:     throw abortException;
```

トランザクションをコミットする際は，自身のトランザクションステータスが abort になっていない事を確認 (72 行目) した上でを cas コマンドによって commited へと書き換えるだけである (74 行目)．これにより，このトランザクションがアクセス権を所有している複数のトランザクショナルキーバリューペアでの論理的な最新のバージョンが一斉にそれぞれの new value の指しているバージョンとなる．もし自分以外のトランザクションによりロケータが書き換えられている場合 (72 行目) は，その前に必ずこちらのステータスの値を abort へ遷移させている (51 行目) のでこちらでの cas コマンドは必ず失敗 (77 行目) し，一貫性の無い状態を commit する事態は発生しない．自トランザクション以外からのトランザクションステータスへの操作は abort への遷移以外存在しないため，cas コマンドの失敗はそのまま abort を意味する (78 行目)．

3.4 既存手法との比較

本提案手法ではどの瞬間にクライアントが離脱してもキーバリューストア内に保存されたトランザクションステータスを用いて後進のクライアントが競合を解決するのでクライアントの離脱のみを特別な場合としいて実装する必要は無い．本手法では競合マネージャの実装によってタイムアウトの他に過去の作業量やユーザ定義の優先度などを判断材料に含める事が出来るため，より高いスループットを実現する可能性がある．またロックによる待機が行われないので，クライアント間に性能差がある場合も高性能なクライアントがより多くのリソースを活用できる可能性がある．

4. 評　　価

3 章で示した手法を Ruby 言語で実装し評価を行った．評価環境は表 1 の通りである．分散キーバリューストアはオープンソースで公開されている Flare[7] を用いた．通常のキーバリューストアと提案手法のそれぞれで合計 1000000 個のキーを保存する操作を 10 回行った平均値を描いたグラフを図 6 に示す．横軸がキーバリューストアとして稼働するサーバの台数であり縦軸が秒間あたり合計で何キーを保存できたかを表している．
クライアント側の各計算機ではそれぞれ 8 つの Ruby プロセスを立ち上げローカルで起動している Flare のプロキシーを経由して保存対象のノードへ直接保存を行う．

CPU	Intel Core2 Quad 2.67Ghz
メモリ	DDR2-800 4GB
ネットワークインタフェース	1000BASE-T
サーバ台数	1〜5 台
クライアント台数	10 台

表 1　評価環境

以下では考察トランザクション無し，すなわち通常のキーバリューストアでの操作では 4 台を超えたところから性能が向上しなくなった．これはクライアント側での負荷量から見てクライアントの台数不足による飽和状態と考えられる．一方でトランザクション有りでのベンチマークではサーバの台数が増えるごとにわずかに性能が低下している．これはトランザクションの制御にかかる CPU 負荷が高い事でクライアントの負荷が飽和状態に陥っており，増加した分のサーバの台数効果ではなく負荷分散に掛かるオーバーヘッドが飽和状態をさらに悪化させている事が考えられる．今後クライアントの台数を増やし，再評価を行う必要がある．

5. 関 連 研 究

キーバリューストア上でトランザクションを実現する研究として scalaris[8],WebShere[9],

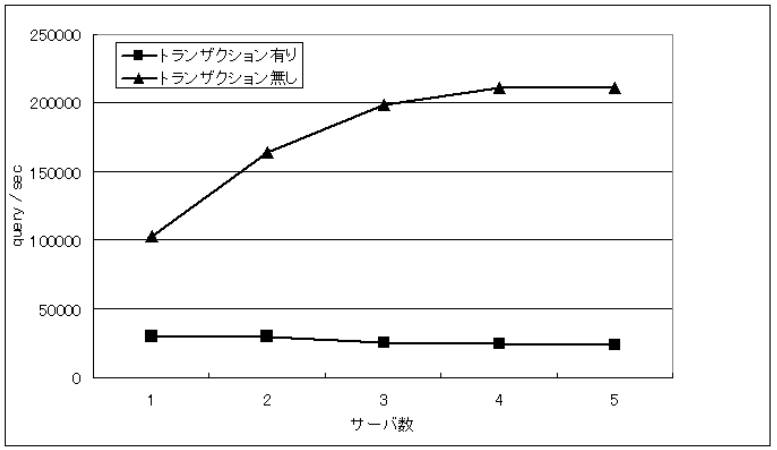

図 6　サーバ台数とスループットのグラフ

などが挙げられるが，いずれもロックベースである．ロックベースなトランザクションは前述したように，クライアントが多く離脱する状況では性能が大きく低下する可能性がある．

6. まとめと今後の課題

一般的な分散キーバリューストアを用いてオブストラクションフリーなトランザクションを行うクライアントライブラリを Ruby で実装し評価した．サーバの台数が増えても性能がほぼ劣化しない事を確認した．

今後の課題として，現時点の実装ではトランザクションステータスや古いバージョンのキーバリューペアがデータストア内部に蓄積されてしまうため定期的に参照されていないキーを削除するガベージコレクションが未実装である．また，より良い競合マネージャの実装や，利用ケースに合わせた間接参照の削減や一貫性の強度を抑制しての性能の最適化，実際にトランザクションを利用したミドルウェアの実装などを行っていく予定である．

参 考 文 献

1) David Karger, Alex Sherman, Andy Berkheimer, Bill Bogstad, Rizwan Dhanidina, Ken Iwamoto, Brian Kim, Luke Matkins, and Yoav Yerushalmi. Web caching with consistent hashing. *Comput. Netw.*, Vol.31, pp. 1203–1213, May 1999.
2) DangaInteractive. memcached http://www.memcached.org/. 2010.
3) Facebook. Cassandra http://cassandra.apache.org/. 2010.
4) Danga Interactive. memcached protocol http://code.sixapart.com/svn/memcached/trunk/server/doc/protocol.txt.
5) Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *International Conference on Distributed Computing Systems*, pp. 522–529, 2003.
6) Maurice Herlihy, Victor Luchangco, Mark Moir, and William N.III scherer. Software transactional memory for dynamic-sized data structures. In *Symposium on Principles of Distributed Computing*, pp. 92–101, 2003.
7) Masaki Fujimoto. Flare http://labs.gree.jp/top/opensource/flare.html, 2011.
8) scalaris: a distributed key-value store. http://code.google.com/p/scalaris/.
9) IBM. Websphere http://www-06.ibm.com/software/jp/websphere/apptransaction/extremescale/. 2009.