平成23年度　　卒業研究論文

分散キーバリューストア上での
トランザクションの実装

指導教官
松尾 啓志 教授
津邑 公暁 准教授

名古屋工業大学院　創成シミュレーション工学専攻
平成22年度入学　22413536番

熊崎 宏樹

目 次

第1章

はじめに

Webサービスを支えるバックエンドでは関係データベース（RDB）が広く使われている．SQLによる柔軟な問い合わせとトランザクションによる並行性制御をサポートし，ノウハウも多く蓄積されたRDBシステムは使い勝手の良いバックエンドである．しかしWebサービスがより多くのユーザーへよりリッチなサービスを提供するためにはRDBの性能がボトルネックになるという問題が起こっていた．

RDBで高い性能を得るための工夫は数多く行われているが，RDBのみではなくキーバリューストア（KVS）をキャッシュとして用いる事で補助的に性能を高める工夫も行われている．KVSとは文字列をkeyとして任意のバイト列をキーバリューペアとして対応付けて保存するタイプのストレージであり，その単純な機能のために高い性能とスケールアウト性能を得る事が出来る．しかしKVSはその単純さの為に同時に一つのキーバリューペアしか扱えないため，あらかじめ用途を限定した上でしか適用できないという問題がある．

そのため現状では扱いやすいデータモデルをサポートするRDBと，高速だが使いどころの難しいKVSを各自工夫して組み合わせて使う事が現在のWebサービスでは一般的となっている．しかしそのシステムではキャッシュの読み出し性能は稼ぐ事ができても，書き込み性能を改善する事はできず，RDBの書き込み性能の限界に達した際に全体を設計し直さなくてはならず移行に必要なコストは高い．

そこで本研究では分散KVSのスケールアウト性能を損なう事なく計算機を跨いだ分散トランザクションを実現する手法を提案する．これは多くの分散KVSが備えている

機能のみを用いてトランザクションを行い，途中でクライアントが離脱した場合でも活性及び一貫性を失わない手法である．この手法によって，RDB で鬼門となっていた書き込み性能のスケールアウト性を得ることができると考えている．

本稿の構成は次の通りである．第 2 章では必要な用語や研究の背景や関連研究を紹介しその問題点を示す，次に第 3 章にて提案手法を説明する，続く第 4 章にて実計算機での評価を行い特性や性能の考察を行う．

第2章

研究の背景

この章では本研究の前提となる従来手法や，関連した研究の紹介を行う．

2.1 キーバリューストア

キーバリューストア（以後 KVS）とは一意な文字列のキーに対して任意のバイナリ列であるバリューを対応付けて保存するタイプのデータストアである．連想配列と同様のセマンティクスをサポートするのみであるが，その単純さを利用した分散化による高いスケーラビリティや障害耐性を得ることができる [1].

KVS には多くの種類が開発されており，データモデルの拡張を行なった Redis や Cassandra[2] の他，kumofs，flare[14]，memcached，Voldemort，Okuyama，Hibari[15] などがある．

KVS はそのセマンティクスの単純さと引き換えに，データモデルや一貫性に不自由な点が多い．何故なら KVS は分散したロックを持たない事によって高いスケーラビリティと低レイテンシを両立しているという経緯があり，データ同士の一貫性において妥協しているためである．複数の KVS に対する一貫性の伴う操作が可能な KVS は少数である．そこで本研究では既存の多くの KVS をそのまま利用した上でトランザクションを実現する手法を提案する．

2.2 トランザクション

トランザクションとは複数の操作を一つの単位として不可分に行う物である [3]．一つのトランザクションの操作全体は完全に行われたか一切行われていないかのどちらかの状態のみしか実行されず（Atomic），矛盾の起きるような操作は行われず（Consistent），操作途中の状態は他のトランザクションから観測されず（Isolation）完了（コミット）したトランザクションの結果は永続化される（Durabule）という，ACID 特性が保障される．トランザクションによって並行システムの複雑性は隠蔽され，システムの効率のよい開発・拡張性・メンテナンス性のために広く持ちいられている．しかしトランザクションをスケールアウトさせる事は自明ではない．広く用いられているトランザクションの手法として 2-Phase Commit（2PC）[4] が挙げられる．これはコンセンサスを得る為に Prepare 状態という合意前の状態を作り，全ての参加者が Prepare 状態になった後で確定操作（コミット）を行うという手法である．この手法では複数のクライアントがトランザクションを行った場合でも適切に排他が行われる上，参加しているサーバの離脱に備えることも可能である一方で，操作途中にトランザクションの実行者（コーディネーター）が離脱した場合にデッドロックへ陥る場合があるため，タイムアウトによるアンロックなど個別の配慮が必要である．これに対しデッドロックの無いノンブロッキングなトランザクション手法として 3-Phase Commit（3PC）[5] も提案されている．こちらは 2PC の Prepare, Commit に加え Pre-commit という状態を用意する事で，コーディネーターが離脱した場合でもサーバ同士で合意形成をし，Pre-commit 状態になったノードの有無によってコミットかアボートへと状態を収束させ，トランザクションを進行させられるという手法であるが，Pre-commit 後にも通信が嵩む上，クライアントが途中で離脱した場合に参加者同士で合意形成を行う必要がありレイテンシが高くなってしまうという欠点がある．本研究では 3PC の欠点である合意形成やコミット情報の同期をキーバリューストアで補いつつスケーラビリティを実現する方法を提案する．

2.3 トランザクショナルメモリ

トランザクショナルメモリとは並行プログラムのための並行性制御アルゴリズムである．元々はキャッシュコヒーレンスプロトコルの拡張としてハードウェアレベルで提唱されていた [6] が，後に同様のセマンティクスをソフトウェアレベルで実現する方法が提案され [7] 改良が続けられている．データベースなどのトランザクションと異なるのは ACID 特性のうち Durability が必要とされない点であり，そのためボトルネックがディスクの IO ではなくメモリの帯域や CPU に移っている．そのためアルゴリズムは低レイテンシでキャッシュコヒーレントにまで気を使った物も提案されている [8]．また様々な並行特性のトランザクショナルメモリが考案されており，Lock を利用した物 [9] から Lock を利用しないノンブロッキング特性を満たす STM[10] や更にその上位の Lock-free 特性を満たす STM[11] が提案されている．本研究ではこのトランザクショナルメモリのノンブロッキング特性に着目して分散キーバリューストアに応用し，コーディネータの離脱に耐性のある分散トランザクションを実現する．

2.4 関連研究

分散 KVS のようなデータストアの上にミドルウェアとしてトランザクションを実現する方法は幾つか提案されている．ここではそれらと本研究との違いを述べる．

2.4.1 sinfonia

sinfonia[12] は分散データストア上でトランザクションを行うミドルウェアである．2PC を行い耐障害性と一貫性を両立しているが，コーディネーターの離脱に備えるために各参加者にコミット済みの情報を埋めこまなくてはならない．本手法ではコミット済みの情報を一箇所に集約させるため，任意のキーバリューペアの情報を少ない無駄で一括に更新できる．

2.4.2 G-Store

G-Store[13] は分散 KVS の上で複数のキーにまたがるアトミック操作を実現するミドルウェアである．複数のキーバリューペアを動的にグルーピングして，その中でリーダーとなるキーを決め，そのキーへのアクセス権を持つクライアントが排他的に読み書きを行えるという物である．この手法では読み出すキーバリューペアの数が多いトランザクションでスケーラビリティが低下する恐れがあるが本研究の手法ではトランザクション中での読み出しにはロックを要求しないため，読み出しの多いワークロードに於いて有利である．

第3章

提案手法

本論文にて提案する手法は，既存の分散 KVS の機能のみを用いてトランザクションを行う物である．

3.1 概観

まず用いる機能と実現するトランザクションの特性について説明する．

3.1.1 利用する KVS の機能

本提案で必要としている KVS 側の機能を記述する．これらの機能が実装されていれば特定の実装の KVS によらずに本提案のトランザクションを実現可能である．

SET 指定したキーバリューペアを保存するコマンドである．キーとして文字列を使い，バリューとして任意の長さの文字列を KVS に保存できる．

GET 指定したキーに対応するバリューを獲得するコマンドである．文字列をキーとして問い合わせを行い対応するバリューが KVS から返送される．バリューと同時に論理クロックやタイムスタンプを獲得できる KVS 実装もある．以後の解説では論理クロックとタイムスタンプを区別せずタイムスタンプと呼ぶ．

CAS　*Compare And Swap* の略で「特定のキーが指定した条件を満たした場合にのみ書き込みが成功する」というコマンドである．その条件としてはそのキーへの書き込みを行ったタイムスタンプや論理クロックを指定する実装が多い．結果が成功したか否かが通知される．

ADD　指定したキーが存在しなかった場合にのみ保存が成功する保存コマンドである．既にキーバリューペアが保存されていた場合には失敗が通知される．

これらの他に，KVS は強い一貫性を保障している必要があり，読み書きの順序に追い越しが発生してはならない．そのため書き込んだ値がいつ読めるかの保障のない KVS や，書き込んだクライアントしかその値をすぐに読める保障のない KVS では利用できない．

また，KVS は永続化が可能な実装でなくてはならない．書き込んだ値は計算機の離脱・故障などによって失われてはいけない．そのためには保存時の他ノードへの複製かディスクへの書き込み，もしくはその両方を行ってデータの損失が起きない分散 KVS である必要がある．

これらの条件を満たす KVS は，例えば Flare[14]，Hibari[15]，membase[16] 等が挙げられる．

3.1.2　トランザクションの特性

本提案で実現するトランザクションの分類について説明する．

一貫性　本提案でのトランザクションは serializable consitency である．これは同時に行われた複数のトランザクションが何らかの順序で行われたのと同等の結果を保障するものである．しかしこのトランザクションは Opacity の条件を満たさない．つまり読み出すデータが一貫しない場合がある．もちろん一貫しないデータを読んだトランザクションは最悪でも commit 時に必ず abort されるため通常は問題がないが，一貫しない値を読んだままトランザクションが続行する可能性を配慮した使い方をする必要がある．この設計を選択した理由は後述する．

並行性　トランザクションはオブストラクションフリー [17] に実行される．この特性によってどのクライアントも任意のタイミングで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．この特性は任意のクライアントがいかなるタイミングで遅延・離脱した場合でもデッドロックさせることなく進行する事を保証する．しかし 2 つ以上のトランザクションが衝突し続けた場合には全体での進行が保証されない場合がある．

耐障害性　本提案でのトランザクションはロックを用いない．クライアントがコミット前に離脱した際には後発のトランザクションが非コミットな状態を確認し適切な abort 処理を行い，コミット後に離脱した場合は適切な後処理を行う．サーバ側の耐障害性は利用する KVS の実装に依存する．

3.2　詳細

提案手法は OSTM[11] の手法を KVS に応用したものである．

3.2.1　トランザクショナルキーバリューペア

トランザクション中に保護されるキーバリューペア（KVP）をトランザクショナルキーバリューペア（TKVP）と呼ぶ．任意のキーバリューペアをカプセル化し、TKVP として扱う事によってトランザクションを実現する．トランザクション中で書き換えるデータのみをこの TKVP で保護する．書き換えないデータはそのまま通常の KVP として扱う．

TKVP と KVP の関係

TKVP に対して実行可能な操作は Set，Get，Commit，Abort である．TKVP 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 TKVP の持ち主を表す物で，後からその KVP に触れ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を採用

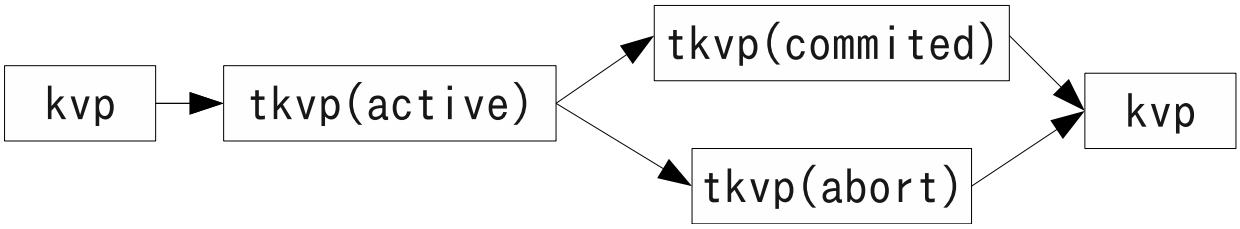

図 3.1: キーバリューペアの状態遷移

することが可能となる．

トランザクション中で書き込みが行われる場合，対象となる KVP は図 3.1 に描かれている通りに TKVP へ状態遷移する．非トランザクションでは KVP は通常の KVS での KVP として振る舞うが，トランザクション中は KVP は TKVP となり，*committed* か *abort* へ状態遷移が決定したのち再び KVP へと戻る．なお *committed* や *abort* や *active* については 3.2.1 にて解説する．トランザクション中で読み出ししか行わない KVP は一切書き換えも行わないため，TKVP へと遷移する事はない．

TKVP の状態

KVP から TKVP に遷移した後でも，TKVP 内部でも状態は遷移する．状態は *committed* や *abort* や *active* の 3 つのうちどれかである．

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，committed,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．

committed: トランザクションが操作を完了している事を意味する

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する

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．committed 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 3.2 に示す．

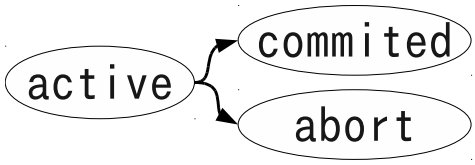

図 3.2: トランザクションステータスの遷移

TKVP の実体

TKVP は KVS 上には図 3.3 の形で保存される．key_a に対応して old と new と status の 3 つがシリアライズされた形で保存される．ここでの status は 3.2.1 にて紹介した値を取り，

committed: new を最新の値として採用

abort: old を最新の値として採用

というルールで TKVP から読み出す．new や old はシリアライズされた値そのものである．

3.2.2 トランザクションのアルゴリズム

トランザクションを行う際の手順を説明する．

図 3.3: TKVP の構造

トランザクションの開始

ADD コマンドを使って *Active* というステータスをバリューとして KVS に保存する．これは以後のトランザクションにて書き込みを行う際に，対外的に TKVP 所有を明示する為に用いる．またステータスと同時に writeset 情報も保存する，この値については後述する．保存の際にはキーはランダムな文字列を用いる．万が一既存のキーと衝突した場合には ADD コマンドは失敗するため，また別のランダムな文字列を生成して成功するまで繰り返す．また，Readset，Writeset も初期化する．Readset はトランザクション中で読み出した KVP の複製，Writeset は読み書きした KVP の複製である．

擬似コードを示す．

```
1  key random_add(value){
2    retry:
3      key_name = random_string();
4      /* add コマンドを発行して保存 */
5      if(kvs.add(key,value) == SUCCESS) return key_name;
6      goto retry; /* 成功するまで繰り返す */
7  }
8  string begin(){
9    my_status = kvs.random_add([ACTIVE, Empty]);
10   readset = new map();
11   writeset = new map();
12   return my_status;
13 }
```

なお `random_add()` で用いるランダムな文字列がユーザーの本来の名前空間を不必要に汚染することを避けるため，生成するランダムキーには Prefix をつけているが簡単のため省略した．

値の読み出し

トランザクション中で必要な値を読み出す操作．既に Readset や Writeset に値を複製している場合はその複製を利用する．もし手元の複製と KVS との間で乖離している場合にはコミット時に検出可能なためここでは関知しない．複製がない場合には GET コマンドを用いて目的となる値とそのタイムスタンプを読み出す．読み出したターゲットが KVP ならば何もしないが，TKVP だった場合にはそのステータスに応じて以下のように分岐する．

committed: `new` を value として採用し、CAS コマンドで KVP へ書き換える

abort: `old` を value として採用し、CAS コマンドで KVP へ書き換える

active: 競合を解決する（方法は後述）

CAS コマンドが失敗した場合には GET コマンドの発行からやり直す．擬似コードをに示す．

```
value get(my_status, key){
  if(writeset.exists(key))
    return writeset[key];
  if(readset.exists(key))
    return readset[key];
  while(true){
    value = kvs.get(key)
    if(!is_tkvp(value)){
      /* 通常のKVPなら */
      readset[key] = value;
      return value;
    }

    /* TKVPなら */
    old, new, status = value;
    owners_status = kvs.get(status);
    if(owners_status == NULL){
      /* 他のスレッドによって既に後始末されている */
      continue;
    }
    if(owners_status == COMMITTED){
      /* COMMITTEDならばnewが最新の値 */
      committed_value == new;
    } else if(owners_status == ABORT){
      /* ABORTならばoldが最新の値 */
      committed_value == old;
    } else if (owners_status == ACTIVE){
      resolve_contention(status);
    }
    if(kvs.cas(key, committed_value)){
      /* 通常のKVPへとCASする */
      readset[key] = committed_value
      return committed_value;
    }
  }
}
```

値の書き込み

トランザクション中で値を書き込む操作は比較的複雑である．既にトランザクションが writeset に複製を持っている場合にはそこを編集するのみであるが，readset に複製を持っている場合には，複製とその値が一致している事を確かめた後，所有者を自分とした TKVP へ書き換える，もし一致しなかった場合にはアボートすることで衝突の早期検出を図っている．また解決済みの TKVP に遭遇した場合に KVP へを書き換える読み出しとは逆に，KVP から自分を所有者とした TKVP へと遷移させる．また自分がトランザクション中に急に離脱する事態に備えて，TKVP の持ち主を自分にする前に，トランザクション開始時に保存した自分のトランザクションステータスにキーを追記する事で，自分が離脱した場合でも第三者のトランザクションが適切に後始末ができるようにしている．

```
1  void set(my_status, key, value){
2    if(writeset.exist(key)){
3      /* 自分のStatusにターゲットとなるキーを追加 */
4      status, writeset = kvs.get(my_status);
5      if(status != ACTIVE){
6        /* 他のクライアントによってAbortさせられた */
7        throw AbortException;
8      }
9      result = kvs.cas(my_status, [ACTIVE, writeset + [key]])
10     if(result == false){
11       /* 他のクライアントによってAbortさせられた */
12       throw AbortException;
13     }
14   }
15
16   while(true){
17     got_value = kvs.get(key)
18     if(got_value == NULL){
19       /* ターゲットが存在しなかったら */
20       if(kvs.add(key, [NULL, value, my_status])){
21         /* add成功 */
22         writeset[key] = value;
23         return;
24       }
25       continue;
26     }
27
28     if(!is_tkvp(got_value)){
29       /* 通常のKVPならば */
30       if(readset.exist(key)){
31         if(readset[key] != got_value){
32           throw AbortException;
```

```
33        }
34      }
35      result = kvs.cas(key, [got_value, value, my_status]);
36      if(result == true){
37        /* KVP → TKVP の遷移に成功 */
38        delete readset[key];
39        writeset[key] = value;
40        return;
41      }else{
42        /* 他者によってKVPを書き換えられたのでやり直し */
43        continue
44      }
45    }
46
47    /* TKVPならば */
48    old, new, status = got_value;
49    if(status == my_status){
50      /* 既に自分が所有しているTKVPへの二度目以降の書き換え操作 */
51      owner_status, owner_writeset = kvs.get(my_status);
52      if(owner_status != ACTIVE){
53        throw AbortException
54      }
55      result = kvs.cas(key, [old, value, my_status]);
56      if(result == true){
57        writeset[key] = value;
58        return
59      }else{
60        /* 他のスレッドがTKVPを書き換えた */
61        throw AbortException;
62      }
63    }else{
64      if(readset.exists(key)){
65        /* 既にreadsetにある値ならば */
66        /* TKVPに遷移している時点で書き換えが発生している */
67        throw AbortException;
68      }
69      /* 他のトランザクションが所有しているTKVPへの操作 */
70      state, ref_list = kvs.get(status);
71      if(state == null){
72        /* そのステータスが既に消されていたらやり直し */
73        continue
74      }
75      if(writeset.(key)){
76        /* 既に自分がそのTKVPの獲得をしたのに他人が所有している */
77        throw AbortException;
78      }
79      if(state == ACTIVE){
80        /* 他のクライアントが書き込む最中なので競合解決へ */
81        resolve_contention(owner);
82        continue
83      }
84      /* TKVPの所有権を自分に移す */
85      if(state == COMMITTED){
86        next_old = new;
```

```
87        }else if(state == ABORT){
88          next_old = old;
89        }
90        result = kvs.cas(key,[INFLATE, next_old, tupled_new, my_status]);
91        if(result == true){
92          delete readset[key];
93          writeset[key] = value;
94        }
95        /* CAS失敗なら初めからやり直し */
96      }
97    }
98  }
```

コミット

コミット操作はまず自身のReadset全てがKVS上の値と一致している事を確認する．この操作によってReadsetのスナップショットが獲得できた事になる．もしスナップショットが一致しなかった場合にはトランザクションをアボートし初めからやり直す．

そしてトランザクション開始時に保存したステータスをACTIVEからCOMMITTEDへCASコマンドを用いて書き換える事によってコミットは達成される．この操作はCASをサポートしたKVSにとっての1コマンドで行われる操作であり，不可分に行われる．この操作の完了の前後によって完全にトランザクションの結果が分岐される．CASコマンドが失敗した場合は他のクライアントによってAbortへ書き換えられた結果に他ならないので自身のトランザクションをアボートし初めからやり直す．この操作の途中でクライアントが離脱した場合も同様に他のトランザクションによっていずれはAbortへ書き換えられるためデッドロックする恐れはない．

CASコマンドが成功した場合にはトランザクションの中で書き換えた値全てをTKVPからKVPへとCASコマンドで書き換えていく．これにより後発のクライアントは最新の値を読むためにgetを一回発行するだけで良くなる．この操作を行う前にクライアントが離脱した場合，他のクライアントがstatusを読んで引き継ぐ．

擬似コードを以下に示す．

```
1  void commit(){
2    status, ref_list = kvs.gets(my_status);
3    if(status == null){
4      /* ステータスが消されている場合 */
5      /* 他のクライアントによってAbort及び消去がされている */
```

```
6       throw AbortException;
7     }
8     if(status != ACTIVE){
9       /* 他のクライアントによって Abort がされている */
10      throw AbortException;
11    }
12    /* スナップショットで正当性チェック */
13    snapshot = kvs.get_multi(readset);
14    if(snapshot != readset){
15      /* スナップショット不一致ならアボート */
16      kvs.cas(status, [ABORT, writeset])
17      throw AbortException;
18    }
19    result = kvs.cas(my_status, [COMMITTED, writeset]):
20    if(result == false){
21      /* cas 失敗なら他者からアボートされている事を意味する */
22      throw AbortException;
23    } else {
24      /* コミット成功 */
25      /* 自分の操作した TKVP を KVP に戻す */
26      for(key in writeset){
27        /* writeset にあるそれぞれのキーを書き換える */
28        old, new, status = kvs.get(key);
29        if(status == my_status){
30          /* 持ち主が自分ならば TKVP に書き換える */
31          kvs.cas(key, writeset[key]);
32        }
33        /* CAS に失敗した場合は他のトランザクションが代行しているので無視 */
34      }
35      /* ステータスを削除する */
36      kvs.delete(my_status);
37      return;
38  }
```

競合解決

競合が発生した場合には競合を解決する必要がある．トランザクションでの読み出し一貫性はスナップショットで解決しているため状態を書き換えない，よって Reader 同士及びは衝突する恐れがなく，Reader が続行中の場合の後発の Writer の衝突は必ず Writer が勝つため考えない．そのため競合が起きるパターンは以下の 2 つである．

Write-Read: 他のトランザクションが書き換えている最中の TKVP から読み出す

Write-Write: 他のトランザクションが書き換えている最中の TKVP に書き込む

これらの競合を解決しなくてはならない．CAS コマンドを用いてステータスを Abort へ書き換え，それ以前までにそのトランザクションが保持していた TKVP 全てを元の KVP へと書き換える．また Abort 状態にした際にはそのトランザクションが保持していた TKVP 全てを KVP へと書き換えた後にステータスを削除する必要がある．

ただし，衝突時に即座にアボートさせると開始されたトランザクションに自分がアボートされてお互いに進行しなくなってしまう状況（*Live Lock*）が発生しうる．そのため試行回数に応じた指数時間のバックオフ待機を行う事でその状況が起きにくいよう工夫を加えている．

以下に擬似コードを示す．

```
1  void resolve(other_status){
2    counter = 0;
3    while(true){
4      /* 指数バックオフ待機 */
5      sleep(0.001 * random(0, 1 << counter));
6
7      /* 現在のステータスを確認 */
8      status, ref_list = kvs.get(other_status);
9
10     /* 競合解決する必要が運良く無くなった場合 */
11     if(status != ACTIVE){
12       /* そのまま帰れる */
13       return;
14     }
15
16     if(count <= 10){
17       /* 待機上限以内なら待機を繰り返す */
18       count += 1;
19       continue;
20     } else {
21       /* 待機上限を超えたのでアボートさせる */
22       count = 0;
23       abort_success = kvs.cas(other_status, [ABORT, ref_list]);
24       if(abort_success == true){
25         for(key in ref_list){
26           old, new, owner_name = kvs.get(key);
27           if(owner_name != other_status){
28             /* 既に他のトランザクションによって横取りされているので無視 */
29             continue;
30           }
31           /* TKVP を KVP へと書き換える */
32           result = kvs.cas(key, old);
33         }
34         /* 全ての TKVP から所有権の剥奪を完了したのでステータスを消せる */
35         kvs.delete(other_status);
36       }
37     }
```

```
38   }
39 }
```

3.2.3 トランザクションのユーザーインターフェース

KVS に対し 3.2.2 に説明した手法を直接利用する場合，同様のパターンを何度も記述する必要がある．予め定形パターンとして定義しておく事でユーザーにとって見通しの良いトランザクションを記述できるインタフェースを用意した．擬似コードを以下に示す.

```
1  bool transaction(fn){
2    my_status = begin();
3    /* ステータスとsetter,getterを結びつける */
4    setter = bind:set(my_status, _1, _2);
5    getter = bind:get(my_status, _1);
6    try{
7      return fn(setter, getter);
8    }catch(AbortException){
9      return false;
10   }
11 }
```

このような形で一連のトランザクション中でのトランザクションのステータス変数を隠蔽する事でユーザーは以下のようにトランザクションを利用することが出来る.

```
1  /* 利用例 */
2  amount = 10;
3  from_account = "accountA";
4  to_account = "accountB";
5
6  /* トランザクションを定義する */
7  bool move_money(setter, getter){
8    from = getter(from_account);
9    to = getter(to_account);
10   setter(from_account, from - amount);
11   setter(to_account, to + amount);
12 }
13 while(true){
14   /* */
15   result = transaction(move_money);
16   if(result == true){
17     /* 成功するまで繰り返す */
18     break;
19   }
20 }
```

3.3 ゾンビトランザクション

本提案でのトランザクションは Serializable Isolation を達成しているが，それはコミットやアボートされたトランザクションに限っての話であり，実行中のトランザクションはその限りではない．実行中のトランザクションは書き換え済みの値しか読み出さないが，トランザクション T1 が A → B の順で読み出しを行う際，別のトランザクション T2 がその間に割り込んで A,B に書き込みを行なった場合，時系列のズレた A，B を読み出してしまう．T1 がその後に A に書き込みを行う場合には Readset との一致比較（3.2.2 の 64 行目）が行われるため検出可能であり，T1 がそのままコミットを行おうとする場合でも Snapshot 比較（3.2.2 の 14 行目）での不一致を検出してアボートされる，一貫性の無い値に基づいたトランザクションがコミットされる事は無いが，一貫性の無い値に基づいてトランザクションが続行されてしまう事自体は回避できていない．これは利用者が何らかの前提に基づいて本ミドルウェアを利用した場合にゼロ除算や無限ループを引き起こす可能性がある．

対策としてグローバルカウンタを用いてトランザクションの時系列を比較するという手法は存在するが [9] 分散 KVS 上で負荷がグローバルカウンタに偏る原因となるため，今回は実装を見送った．そのため本ミドルウェアを利用する際にはゾンビトランザクションが発生しないよう配慮する必要がある．

3.4 間接化

本提案でのトランザクションは，バリューを保存する際には初めの Set で 1 回，コミット後に通常 KVP に書き換える為にもう 1 回で合計 2 回書き込む必要があり，大きなバリューを扱う場合にその通信量がボトルネックになる可能性がある．そのため大きなバリューを扱う場合には動的に間接化を用いるようにした．これはバリューを保存する際に間接化されている事の目印と，間接化後のバリューを保存しているキーをペアで保存する物であり，バリューを KVS に送る回数を 1 回に削減できる．間接化の分レイテンシがトレードオフとなるが，バリューの大きさによってどちらかが有利になる場面は異なると見込んでいる．

第4章

評価

このシステムを AmazonEC2 上の仮想マシン上に実装して評価を行なった．用いた環境は以下の通りである．

仮想マシンマネージャ	Xen
CPU	20ECU
Memory	7.5B
KVS	memcached 1.4.13
提案実装言語	python2.7

図 4.1: EC2 仮想マシン

トランザクションのベンチマークを行う種目として，キーバリューペアを数値を一つ保持した口座と見立てて複数用意し，トランザクションを用いる事でその口座間で数値の総量を変える事無く残高の一部を移動させるという操作を考える．そこで発行される memcached コマンドの一例は以下の通りである．

```
1  add(status, 'active')
2  get(from_account) # 振り込み元
3  get(to_account) # 振り込み先
4  cas(status, ['active', from_account]) # 他のクライアントによる後処理のために書き込むKVP 名を記憶
5  cas(from_account, [from_old, from_new, status]) # 操作前の額と操作後の額の両方をステータスを保存
6  cas(status, 'active,␣[from_account,to_account]') # 他のクライアントによる後処理のために書き込むKVP 名を記憶
7  cas(to_account, [to_old, to_new, status]) # 操作前の額と操作後の額の両方をステータスを保存
8  cas(status, ['commit', [from_account, to_account]) # コミット
```

```
 9   cas(from_account, from_new) # 直接KVP として読み出せるよう戻す
10   cas(to_account, to_new) # 上に同じ
11   delete(status) # ステータスを消去する
```

実際には cas の直前に gets で unique 値の獲得操作を行っているためこの例に gets 操作が加わる上，各 cas や add コマンドが失敗した場合にはリトライを行うためこれ以上に増えることは起こり得る．この評価では操作対象の口座として 10 万口座を用意した．クライアントとして 10 台の計算機を使用し，全口座の中からランダムに 2 つ選び出して残高の一部を移動させる．memcached として使うサーバ台数を 1 台から 10 台の間で変動させて評価を行なった．その結果を図 4.2 に示す．

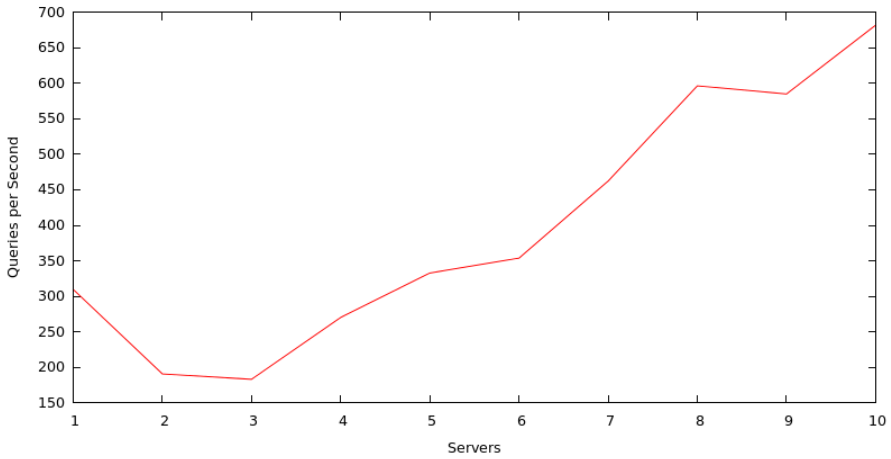

図 4.2: クライアント数 10 台での結果

台数を増やすごとに大まかに性能は向上しているがクライアント 1 台での性能が 2 台での性能より大幅に大きく調査が必要な結果となった．今回は充分な実験時間を取れず平均値を出すに至れていない．

次にクライアント台数を大幅に増やして 90 台で評価を行なった．それを表したグラフが 4.3 である．クライアントが増えた結果としてクライアントが増えたので相対的にトランザクションの衝突が増加し性能が極端に低下している．

そこでクライアント台数に伴い口座の件数も 300 万件へ増やして検証を行なった．結果が 4.5 である．こちらは最大で 1400 トランザクションを毎秒行えている．しかし性能が安定せずサーバ台数を増やした場合に性能が劣化している場合もある．再現性が

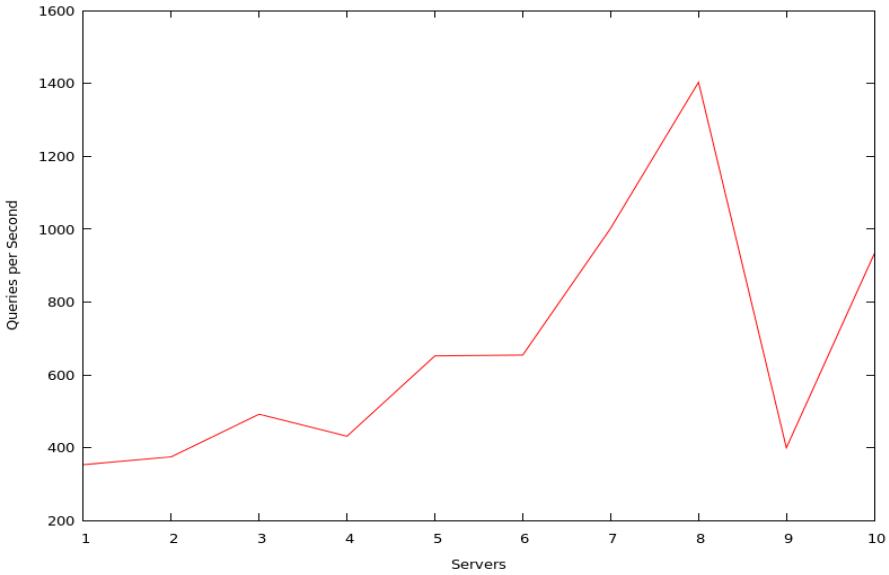

図 4.3: クライアント数 90 台での結果 1

無い現象のため，トランザクション衝突などの乱数に要因のあると見られるが詳しい検証はまだ行えていない．

次にクライアントとサーバを同一の計算機に共存させた場合のベンチマークを取った．クライアントとして 50 台を用いてそのうちの 1 台から 15 台までをサーバとしても利用した環境でスループットを計測した．こちらは 7000 トランザクション毎秒付近まで性能が出ている．元々サーバの性能が飽和するまで使っていなかったケースにおいて高い性能を出せている．しかしサーバが 1 台の場合に 500 トランザクション毎秒程度の性能を示している点は改良の余地があると見られ今後の研究で明らかにしていきたい．

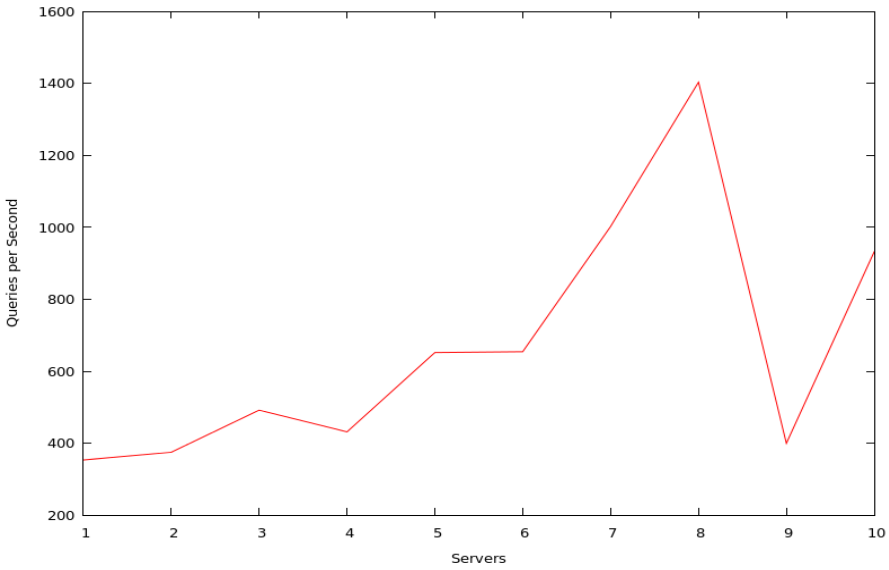

図 4.4: クライアント数 90 台での結果 2

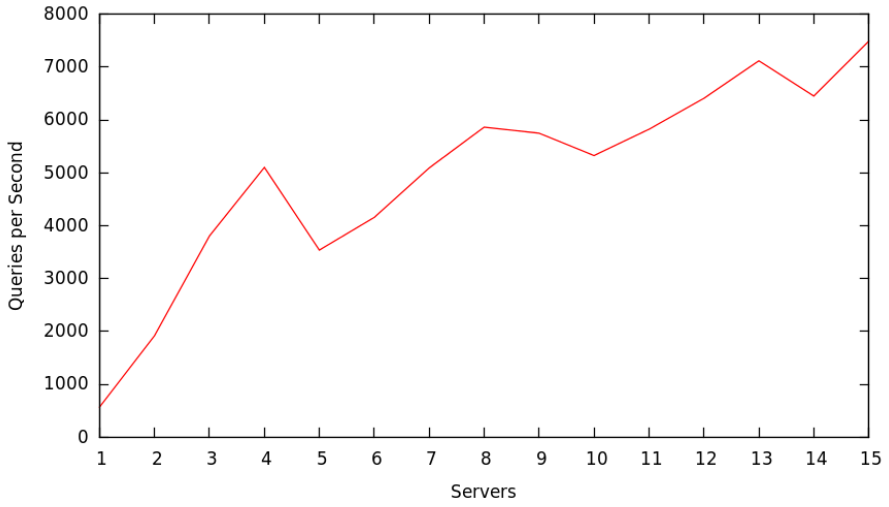

図 4.5: サーバとクライアントが同一の場合の性能

第5章

まとめと今後の課題

本研究では分散KVSをベースにした分散トランザクションの実装を行なった．障害耐性とスケーラビリティの両立を狙ってSPoFとHotSpotの排除が可能な設計とした．今後の課題としてはトランザクションの書き込みをコミット時まで遅延することでレイテンシとスループットのトレードオフを図れるため動的に戦略を変更するようアルゴリズムに改変を加えるといったトランザクションそのものの改良や，一貫性が得られるという利点を活かしてB木などをキーバリューストア上に実現するといったトランザクションの活用法を考案するという発展などが考えられる．

謝辞

本論文の作成にあたり，終始適切な助言を賜り，また丁寧に指導して下さった松尾啓志教授に感謝します．また，日々の研究生活に注ぐ意欲と情熱を支えて下さった研究室の皆様に感謝します.

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] Facebook. Cassandra http://cassandra.apache.org/. 2010.

[3] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of The ACM*, Vol. 19, pp. 624–633, 1976.

[4] Jim Gray. *Notes on Data Base Operating Systems*. 1978.

[5] Dale Skeen. Nonblocking commit protocols. In *International Conference on Management of Data*, pp. 133–142, 1981.

[6] Maurice Herlihy, J. Eliot B. Moss, and B. Moss. Transactional memory: Architectural support for lock-free data structures. In *International Symposium on Computer Architecture*, pp. 289–300, 1993.

[7] Nir Shavit and Dan Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pp. 204–213, 1995.

[8] Aravind Natarajan and Neeraj Mittal. False conflict reduction in the swiss transactional memory (swisstm) system. In *International Parallel and Distributed Processing Symposium/International Parallel Processing Symposium*, pp. 1–8, 2010.

[9] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In Shlomi Dolev, editor, *Distributed Computing*, Vol. 4167 of *Lecture Notes in Computer Science*, pp. 194–208. Springer Berlin / Heidelberg, 2006. 10.1007/11864219_14.

[10] 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.

[11] Tim Harris and Keir Fraser. Language support for lightweight transactions. *Sigplan Notices*, Vol. 38, pp. 388–402, 2003.

[12] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. *SIGOPS Oper. Syst. Rev.*, Vol. 41, pp. 159–174, October 2007.

[13] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pp. 163–174, New York, NY, USA, 2010. ACM.

[14] Masaki Fujimoto. Flare http://labs.gree.jp/top/opensource/flare.html, 2011.

[15] Scott Lystig Fritchie. Chain replication in theory and in practice. In *Proceedings of the 9th ACM SIGPLAN workshop on Erlang*, Erlang '10, pp. 33–44, New York, NY, USA, 2010. ACM.

[16] DangaInteractive. couchbase http://www.couchbase.com/. 2012.

[17] 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.