Kumofs memo – Kumo Fast Storage

FURUHASHI Sadayuki

Contents

1 概要 **1**

1.1 Consistent Hashing 1

1.2 set/get/delete の保証範囲 2

1.2.1 set(key, value) 2

1.2.2 get(key) 2

1.2.3 delete(key) 2

2 インストールと実行 **3**

2.1 依存関係 3

2.1.1 コンパイル時に必要なもの 3

2.1.2 コンパイル時と実行時に必要なもの 3

2.2 コンパイル 3

2.2.1 configure フラグ 4

2.3 実行例 4

2.4 主な引数 4

2.4.1 kumo-manager 4

2.4.2 kumo-server 4

2.4.3 kumo-gateway 5

2.5 その他の引数 5

3 kumoctl **6**

3.1 stat 6

3.2 attach 7

3.3 attach-noreplace 7

3.4 detach 7

3.5 detach-noreplace 7

3.6 replace 7

3.7 backup 7

4 死活監視と再配置 **8**

4.1 ダウンの検出 8

4.2 接続の検出 8

4.3 ハッシュ空間の更新 8

4.4 レプリケーションの再配置アルゴリズム 9

5 レプリケーション **10**
5.1 set/delete の伝播 10
5.2 get のフォールバック 10
5.3 タイムアウト 10
5.4 リトライ 11

6 クロック **12**
6.1 クロックのフォーマット 12
6.2 データベースのフォーマット 12
6.3 レプリケーションでの利用 12
6.4 Manager 間の協調動作での利用 13

Chapter 1

概要

kumofs は key-value 型のデータを保存する分散ストレージ。key=>value を保存する **set**、key を取得する **get**、key を削除する **delete** の 3 つの操作をサポートする。

データを保持する **Server**、Server 群を管理する **Manager**、アプリケーションからのリクエストを Server に中継する **Gateway** の 3 種類のノードでシステムを構成する。

データは 3 台の Server にコピーされて保存される。2 台までなら Server がダウンしても動作し続ける。

データの取得は複数の Server に分散して行われ、Server を追加するほど性能が向上する。

システムを止めずに Server を追加したり、切り離したりすることができる。Server の追加と切り離しは自動的には行われず、**kumoctl コマンド**を使ってマニュアルで行う。

Server を追加したときと切り離したときに、レプリケーションされたデータの再配置が行われる。

Server 群は Manager によって死活監視されている。1 台か 2 台の Manager が起動していないと Server がダウンしたときにシステムが停止してしまう。また Server の追加と切り離しは Manager が起動していないと行えない。

1.1 Consistent Hashing

Consistent Hashing を使ってデータを分散して保存する*1。

Server がダウンしたときは、その Server の仮想ノードに **fault フラグ**がセットされる。set/get/delete は fault フラグがセットされた Server をスキップして行われる。つまり、通常動作時はレプリケーションは 3 つ作成されるが、1 台が fault 状態ならコピーは 2 つ、2 台が fault 状態ならコピーは 1 つしか作成されない key が存在することになる。fault 状態の Server が 3 台以上になると、get/set/delete が失敗し続ける key が存在することになる。

Server がダウンしても fault フラグがセットされるだけで、レプリケーションの再配置は行われない。fault フラグがセットされた Server が存在する状態で、kumoctl コマンドを使って **detach** コマンドを Manager に送信すると、fault フラグがセットされた Server がハッシュ空間から取り除かれる。同時にレプリケーションの再配置が行われ、すべての key に対してレプリケーションが 3 つ作成されるようにデータがコピーされる。

Server が追加されてもすぐにはハッシュ空間には追加されず、レプリケーションの再配置は行われない。新たな Server が起動している状態で、kumoctl コマンドを使って **attach** コマンドを Manager に送信すると、新しい Server がハッシュ空間に追加される。同時にレプリケーションの再配置が行われ、すべての key に対してレプリケーションが 3 つだけ存在するようにデータが移動される。

TODO: auto-replace

1.2 set/get/delete の保証範囲

1.2.1 set(key, value)

key=>value を保存する。保存できれば成功を返す。保存できなければエラーを返す。

既に key が保存されていたとき、set が成功した場合は key の値は確実に上書きされている。

set が失敗した場合は key の値は不定になっている。これは失敗したときにロールバックを行わないため。ロールバックを一貫性を損なうことなく行うための高級なアルゴリズムは実装されていない/使うと性能が低下してしまう。

Server はレプリケーション先の 2 台〜0 台のすべての Server にデータが受信されたことを確認してから Gateway にレスポンスを返す。どれか 1 台でもコピー処理が失敗したらエラーを返す。つまりアプリケーションに成功が返されたときは fault 状態でないすべての Server にレプリケーションがコピーされており、それ以降に古いデータが読み出されることはない。しかしディスクに書き込まれているとは限らない。

1.2.2 get(key)

key を set するリクエストが成功していた場合は、その key に対応する value を返す。set が失敗していた場合は、null か、set に失敗した value が返る。それ以外であれば null を返す。

key を set するリクエストが成功して value が保存されていたとしても、レプリケーションされたすべての Server の負荷が非常に高いために応答できない場合は、get がタイムアウトする可能性がある。

key が保存されていなかった場合はエラーにならないが、タイムアウトした場合はエラーになる。

1.2.3 delete(key)

key を削除する。

再配置処理を行っている間に delete を行うと、削除されないことがある。これはレプリケーションをコピーする Server 同士のやりとりが、Gateway が送出した delete リクエストと入れ違う可能性があるため。

*1[1]ハッシュ関数は SHA-1 で、下位 64 ビットのみ使う。仮想ノードは 128 台

[1] footnote4-1-anchor.tex

Chapter 2

インストールと実行

2.1 依存関係

2.1.1 コンパイル時に必要なもの

- g++ >= 4.1
- ragel
- bzaar >= 1.0

2.1.2 コンパイル時と実行時に必要なもの

- ruby >= 1.8
- libcrypto(openssl)
- Tokyo Cabinet >= 1.3.8

2.2 コンパイル

まず最新の MessagePack と mpio ライブラリをインストールする。

```
$ bzr branch lp:msgpack
$ cd msgpack
$ ./configure && make && make install
$ cd ruby
$ ./gengem
$ gem install gem/pkg/msgpack-*.gem

$ bzr branch lp:mpio
$ cd mpio
$ ./configure && make && make install
```

次に kumofs をインストールする。

```
$ ./configure && make && make install
```

2.2.1 configure フラグ

–**with-mpio=DIR** mpio がインストールされているディレクトリを指定する

–**with-msgpack=DIR** MessagePack がインストールされているディレクトリを指定する

–**with-tokyocabinet=DIR** TokyoCabinet がインストールされているディレクトリを指定する

–**with-tcmalloc**[**=DIR**] tcmalloc とリンクする

–**enable-async-replicate-set** set 操作でレプリケーションするとき他の Server からの応答を待たずに成功を返すようにする

–**enable-async-replicate-delete** delete 操作でレプリケーションするとき他の Server からの応答を待たずに成功を返すようにする

–**enable-trace** 画面を埋め尽くすほど冗長なデバッグ用のメッセージを出力するようにする

2.3 実行例

s1〜s4 の 4 台のホストでクラスタを構成し、**c1** と **c2** のホストで動作するアプリケーションから利用する例。
s1〜s4 で Server を動かし、**s1** と **s2** では同時に Manager も動かす。**c1** と **c2** では Gateway を動かす。

```
[s1]$ kumo-manager -v -l s1 -p s2    # Manager 同士は互いに指定する
[s2]$ kumo-manager -v -l s2 -p s1    # Manager 同士は互いに指定する
[s1]$ kumo-server -v -m s1 -p s2 -l s1 -s database.tch    # -m と-p で Manager を指定する
[s2]$ kumo-server -v -m s1 -p s2 -l s2 -s database.tch    # -l は常に自ホストのアドレス
[s3]$ kumo-server -v -m s1 -p s2 -l s3 -s database.tch    # -s はデータベース名
[s4]$ kumo-server -v -m s1 -p s2 -l s4 -s database.tch    # -v は冗長なメッセージを出力
[c1]$ kumo-gateway -v -m s1 -p s2 -c 11211    # 11211/tcp で memcache
[c2]$ kumo-gateway -v -m s1 -p s2 -c 11211    # text protocol を listen する
```

2.4 主な引数

2.4.1 kumo-manager

-l <address> 待ち受けるアドレス。**他のノードから見て**接続できるホスト名とポート番号を指定する

-p <address> もし存在するなら、もう一台の Manager のホスト名とポート番号を指定する

–**auto-replace** Server が追加・切断されたときに、マニュアル操作を待たずにレプリケーションの再配置を自動的に行うようにする。実行時でも kumoctl コマンドから変更できる

-c <port> kumoctl からのコマンドを受け付けるポート番号を指定する

2.4.2 kumo-server

-l <address> 待ち受けるアドレス。**他のノードから見て**接続できるホスト名とポート番号を指定する

-m <address> kumo-manager のホスト名とポート番号を指定する

-p <address> もし存在するなら、もう一台の Manager のアドレスを指定する

-s <path.tch> データを保存するデータベースのパスを指定する

2.4.3 kumo-gateway

-m <address> kumo-manager のホスト名とポート番号を指定する

-p <address> もし存在するなら、もう一台の Manager のアドレスを指定する

-c <port> memcache text protocol を待ち受けるポート番号を指定する

2.5 その他の引数

-o <path.log> ログを標準出力ではなく指定されたファイルに出力する

-d <path.pid> デーモンになる。指定されたファイルに pid を書き出す

-v WARN よりレベルの低いメッセージを出力する

TODO その他の引数
*1[1]ハッシュ関数は SHA-1 で、下位 64 ビットのみ使う。仮想ノードは 128 台

[1] footnote4-1-anchor.tex

Chapter 3

kumoctl

kumoctl コマンドを使うと Manager の状態を取得したり、コマンドを送ったりできる。
Ruby で書かれたスクリプト。実行するには gem で msgpack パッケージをインストールする。
第 1 引数に Manager のホスト名とポート番号を指定し、第 2 引数にコマンドを指定する。

```
$ kumoctl --help
Usage: kumoctl address[:port=19799] command [options]
command:
   stat                       get status
   attach                     attach all new servers and start replace
   attach-noreplace           attach all new servers
   detach                     detach all fault servers and start replace
   detach-noreplace           detach all fault servers
   replace                    start replace without attach/detach
   backup  [suffix=????????]  create backup with specified suffix
   enable-auto-replace        enable auto replace
   disable-auto-replace       disable auto replace
```

3.1 stat

Manager が持っているハッシュ空間を取得して表示する。

```
$ kumoctl localhost stat
hash space timestamp:
  Wed Dec 03 22:15:45 +0900 2008 clock 58
attached node:
  127.0.0.1:8000  (active)
  127.0.0.1:8001  (fault)
not attached node:
  127.0.0.1:8002
```

attached node はハッシュ空間に入っている Server の一覧を示している。**(active)** は正常動作中の Server で、**(fault)** は fault フラグが立っている Server を示している。

not attached node はハッシュ空間に入っていないか、入っているが (fault) 状態でまだ再 attach されていない Server の一覧を示している。

3.2 attach

stat で **not attached node** に表示されている Server をハッシュ空間に組み入れ、レプリケーションの再配置を開始する。

3.3 attach-noreplace

attach と同じだがレプリケーションの再配置を開始しない。再配置をしないまま長い間放置してはいけない。再配置を行わないと、エラーが積もって Gateway から最新のハッシュ空間を要求されたとき（後述)、Gateway が持っているハッシュ空間と Server が持っているハッシュ空間が食い違ってしまう。食い違うと set や delete がいつまで経っても成功しない。

3.4 detach

stat で **attached node** に表示されていて (fault) 状態の Server をハッシュ空間から取り除き、レプリケーションの再配置を開始する。

3.5 detach-noreplace

detach と同じだがレプリケーションの再配置を開始しない。再配置をしないまま長い間放置してはいけない。

3.6 replace

レプリケーションの再配置を開始する。

3.7 backup

コールドバックアップを作成する。バックアップは Server で作成され、元のデータベース名に suffix を付けた名前のファイルにデータベースがコピーされる。手元にバックアップを持ってくるには、rsync や scp などを使って Server からダウンロードする。

suffix は省略するとその日の日付（YYMMDD）が使われる。

Chapter 4

死活監視と再配置

4.1 ダウンの検出

Manager/Server 同士の接続では、あるノードにリクエストまたはレスポンスを送信しようとしたときに、そのノードとのコネクションが一本も存在せず、その上 connect(2) が 4 回*2 連続して失敗したら、そのノードはダウンしたと見なす。

Manager と Server は 2 秒間隔*3 で keepalive メッセージをやりとりしているので、いつも何らかのリクエストかレスポンスを送ろうとしている状態になっている。

connect(2) は次の条件で失敗する：

- 接続相手から明示的に接続を拒否された（Connection Refused）
- 接続相手からの応答がない時間が 3 ステップ*4 続いた。1 ステップは 0.5 秒*5

Gateway と Manager、Gateway と Server の接続は、TCP コネクションが切断されたらその Gateway はダウンしたと見なす。Gateway はデータの一貫性に何も影響しないのでこの実装になっている。

4.2 接続の検出

Manager/Server 同士の接続では、あるノードから接続を受け付けた後、そのノードから初期ネゴシエーションメッセージを受け取り、かつそのメッセージのフォーマットが正しければ、そのノードが新たに起動したと見なす。

こちらから接続するときは、最初に必ず初期ネゴシエーションメッセージを送信する。

Gateway と Manager、Gateway と Server の接続は、TCP コネクションが確立されたらその Gateway は新たに起動したと見なす。Gateway はデータの一貫性に何も影響しないのでこの実装になっている。

4.3 ハッシュ空間の更新

Consistent Hashing のハッシュ空間を更新できるのは Manager だけで、最新のハッシュ空間は常に Manager が持っている。

通常動作時には 1 種類のハッシュ空間しか存在しないが、レプリケーションの再配置を行っている間は 2 種類のバージョンが存在する。最新のもの（Server の追加/切り離しの更新が反映されている）は **whs**、1 つ前のバージョン（Server の追加/切り離しの更新が反映されていない）は **rhs** という名前が付いている。Manager と Server は 2 種類のハッシュ空間を持っており、Gateway は 1 種類しか持っていない。

Manager は kumoctl コマンドでレプリケーションの再配置を行うように指令されると、まず Server の追加/切り離しを whs に反映する。もう 1 台の Manager が存在すればその Manager に更新した whs を送信する。

次に認識しているすべての Server に whs を送信し、レプリケーションのコピーを行うようにコマンドを送る。Server は自分が持っている whs と Manager から送られてきた whs を比較し、必要なら他の Server にデータのコピーを行う（このときデータベースを上から下まで読み込む)。Server はコピーが終わったら whs を rhs にコピーする。

Server はすべてのデータを確認し終えたら、Manager にコピーが終了した旨を通知する。Manager はすべての Server でコピーが終了した通知を受け取ったら、whs を rhs にコピーする。また、認識しているすべてのサーバーにレプリケーションの削除を行うようにコマンドを送る。Server は whs を参照して、自分が持っている必要がないデータがデータベースの中に入っていたら、それを削除する（このときデータベースを上から下まで読み込む)。

Manager はレプリケーションのコピーを行っている最中に Server がダウンしたことを検知したら、すべての Server からレプリケーションのコピーが終了した通知を受け取っても、レプリケーションの削除を行わない。

Server はクライアントから get/set/delete リクエストを受け取ったとき、その key に対する割り当てノードが本当に自分であるか確認するために、get の場合は rhs を、set/delete の場合は whs を参照する。

4.4 レプリケーションの再配置アルゴリズム

TODO レプリケーションの再配置アルゴリズム

logic/srv_replace.cc:Server::replace_copy()

Chapter 5

レプリケーション

5.1 set/deleteの伝播

Gateway に set リクエストを送信すると、key をハッシュ関数に掛けてハッシュ空間から検索し、一番最初にヒットした Server に対して set リクエストが送信される。

set リクエストを受け取った Server は、まず key をハッシュ関数に掛けてハッシュ空間から検索し、自分が確かに最初にヒットする Server かどうか確かめる。そうでなければ Gateway に「ハッシュ空間が古いぞ」とエラーを返す。

次に Server は、自分の次の Server と次の次の Server にデータをコピーする。このときコピー先の Server に fault フラグが立っていたら、その Server にはコピーしない。

Gateway は set/delete が何回失敗しても、次の Server にフォールバックすることはない。set 先の Server が別の Server に切り替わるのは、Manager から新しいハッシュ空間を届いたときのみ。

以上の仕組みから、ある key を set/delete するときは必ず単一の Server を経由することになる。このためほぼ同時に set/delete されても必ず順序が付けられ、常に最新の結果がだけが残る。

5.2 getのフォールバック

Gateway は get リクエストがタイムアウトしたり失敗したりすると、ハッシュ空間上の次の Server にリクエストする。それでもタイムアウトしたときは次の次の Server にリクエストする。リトライ回数の上限に達するまで、最初の Server →次の Server →次の次の Server →最初の Server →…とリトライが繰り返される。

get は Manager から新しいハッシュ空間が届くのを待つことなくフォールバックする。

5.3 タイムアウト

Gateway でも Server でも Manager でも、リクエストを送ってから 10 ステップ（1 ステップは 0.5 秒*6）の間にレスポンスが返ってこないと、そのリクエストはタイムアウトしてエラーになる。

プログラムから見て TCP コネクションが確立しているか否かはタイムアウトには関係しない。コネクションが確立していなくても時間以内に再接続してレスポンスが返れば正常通り処理が続行され、コネクションが確立していても時間以内にレスポンスが返ってこなければタイムアウトする。

Gateway は Server に送ったリクエストがエラーになった回数が 5 回*7 以上失敗すると、Manager から最新のハッシュ空間を取得する。

5.4 リトライ

Gateway は set は最大 20 回*8 まで、delete は最大 20 回*9 まで、get は最大 5×(レプリケーション数==3 + 1) 回*10 までリトライする。限界までリトライしても失敗したらアプリケーションにエラーが返される。

*1[1]ハッシュ関数は SHA-1 で、下位 64 ビットのみ使う。仮想ノードは 128 台

*2[2]–connect-retry-limit で指定

*3[3]–keep-alive-interval 引数で指定

*4[4]–connect-timeout-steps 引数で指定

*5[5]–clock-interval 引数で指定

*6[6]–clock-interval 引数で指定

*7[7]–renew-threashold 引数で指定

*8[8]–set-retry 引数で指定

*9[9]–delete-retry 引数で指定

*10[10]係数は–get-retry 引数で指定

[1] footnote4-1-anchor.tex
[2] footnote4-2-anchor.tex
[3] footnote4-3-anchor.tex
[4] footnote4-4-anchor.tex
[5] footnote4-5-anchor.tex
[6] footnote4-6-anchor.tex
[7] footnote4-7-anchor.tex
[8] footnote4-8-anchor.tex
[9] footnote4-9-anchor.tex
[10] footnote4-10-anchor.tex

Chapter 6

クロック

データベースに保存されているすべての value や、ハッシュ空間には、クロック（=タイムスタンプ）が付与されている。value 同士やハッシュ空間同士でどちらが新しいかを比べるために利用している。

ref:Lamport Clock の解説[1]

6.1 クロックのフォーマット

クロックは 64 ビットの整数で、上位 32 ビットには UNIX タイム（精度は秒）、下位 32 ビットには Lamport Clock が入っている。

UNIX タイムが上位に入っているので、Server/Manager 同士の時刻が 1 秒以上ずれていると、Lamport Clock に関係なく間違った比較が行われてしまう。

6.2 データベースのフォーマット

データベースに value を保存するとき、先頭の 64 ビットにクロックを付加して保存する。またその次の 64 ビットも予約してあるが、使っていない。

```
Database entry format
Big endian
+--------+--------+-----------------+
|   64   |   64   |       ...       |
+--------+--------+-----------------+
clocktime
         reserved
                  data
```

6.3 レプリケーションでの利用

Server から別の Server にデータをコピーするとき、後から来た set リクエストのレプリケーションが、先に来た set リクエストのレプリケーションを追い抜いて先行してしまうことが発生し得る。Server はレプリケーションを受け取ったとき、既に保存されている value のクロックと新たに届いた value のクロックを比べ、新たに届いた方が新しかった場合のみデータベースを更新する。

レプリケーションの再配置を行うとき、ほとんどの場合はレプリケーションされたどの Server も同じデータを持っているが、set が失敗していた場合は異なるデータを持っている可能性がある。このときどの Server が持っているデータが最新なのか比べる必要があり、クロックを利用して比較する。

[1] http://funini.com/kei/logos/clock.shtml

6.4 Manager 間の協調動作での利用

Manager が 2 台動作しているとき、どちらが持っているハッシュ空間が最新なのかを比べる必要がある。ハッシュ空間を更新するときに更新した時のクロックを付与しておき、比較するときにこのクロックを利用する。

*1[2]ハッシュ関数は SHA-1 で、下位 64 ビットのみ使う。仮想ノードは 128 台

*2[3]–connect-retry-limit で指定

*3[4]–keep-alive-interval 引数で指定

*4[5]–connect-timeout-steps 引数で指定

*5[6]–clock-interval 引数で指定

*6[7]–clock-interval 引数で指定

*7[8]–renew-threashold 引数で指定

*8[9]–set-retry 引数で指定

*9[10]–delete-retry 引数で指定

*10[11]係数は–get-retry 引数で指定

[2]footnote4-1-anchor.tex
[3]footnote4-2-anchor.tex
[4]footnote4-3-anchor.tex
[5]footnote4-4-anchor.tex
[6]footnote4-5-anchor.tex
[7]footnote4-6-anchor.tex
[8]footnote4-7-anchor.tex
[9]footnote4-8-anchor.tex
[10]footnote4-9-anchor.tex
[11]footnote4-10-anchor.tex