

MMOGゲームサーバ プログラミングの実際

～ データベースアクセスを題材にして ～

CEDEC 2009 9/1 11:20 ～

コミュニティーエンジン株式会社 中嶋謙互

株式会社ケイブ 巽謙聡 田中裕臣

株式会社ヘッドロック 中麦

(発表順、敬称略)

発表者紹介（敬称略・発表順）

- 中嶋@Community Engine



- MMOG向けミドルウェア”VCE”日本シェア50%

- 巽/田中@cave

- 真・女神転生IMAGINE

運営:ケイブ



- 中@Headlock

- パンドラサーガ、ai sp@ce

運営:ゴンゾロッソ



本レクチャーの題材

- MMOGサーバにおけるデータベースアクセスの実装法

ボトルネックや直しにくいバグ、煩雑なコードの温床。

MMOGでデータベースアクセスをどう書くのが良いのか、まだ定石だと言える方法はありませんが、実例を見て理解を深めたいと思います。

本レクチャーの構成

1. MMOGサーバ実装の一般論おさらい
2. 真・女神転生IMAGINEの場合
3. パンドラサーガの場合
4. まとめ

I. MMOGサーバ実装の 一般論おさらい

ここでいうMMOGとは

- Massively Multiplayer Online Game
- **永続化**されたひとつのゲーム空間を**数千人以上**が数ヶ月～数年以上の**長期**にわたり**共有**し続ける**リアルタイム**なゲーム
- World of Warcraft , FFXI, Ultima Online . . .

ゲーム結果が永続化されないMOタイプあるいはad-hocタイプのゲームや、多人数同時でもリアルタイムではないブラウザ三国志などのゲームはのぞく。
画面に何人出るかはあまり関係がない。

MMOG特有の要求

- **レスポンス**：Webだと2000ms待てるがゲームだと100msしか待てない。リアルタイム性が必要。
- **信頼性**：稀少性の高いアイテムが多く、データ価値が高いため、1～2分のデータ巻き戻りも許されない。
- **処理性能**：採算性のため、NPC(MOB)やPCを可能な限り大量・高密度に実装する必要がある。

CLI - Frontendは、帯域削減やレイテンシの認知上の改善に課題がもちろんあるが、Frontend-Backendほど悩まなくて良い。

MMOGの要求への対応（非同期化）

- 処理性能のため、スレッドではなく、コールバックを使った非同期のタスクシステムを用いる。
- ネットワークから受信したパケットも同様
- 信頼性とレスポンスを両立するため、永続化処理も、オンメモリのゲームデータを変更した後に非同期でやる

MMOGの要求への対応（外部化）

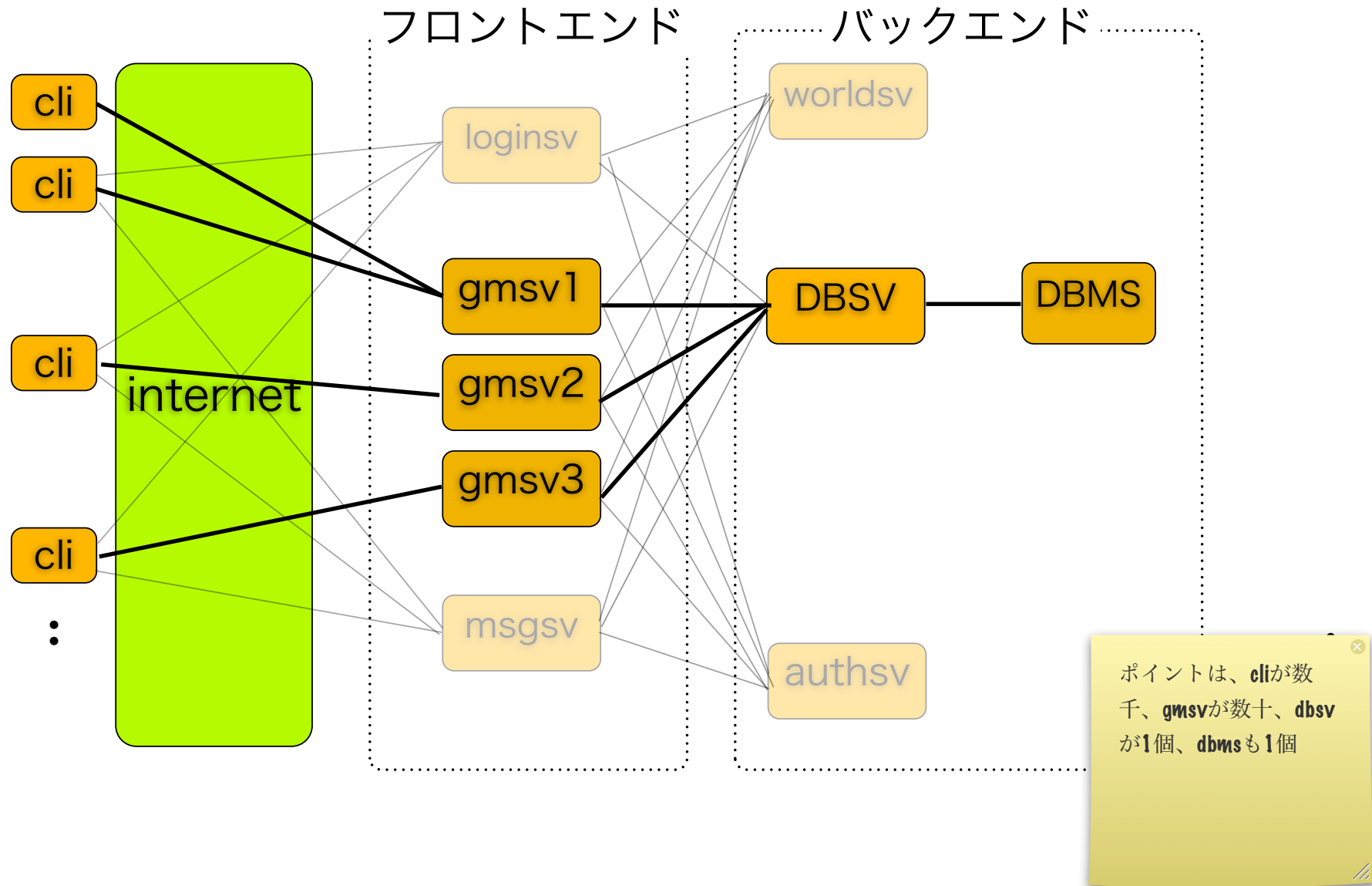
- ゲームロジックをできる限り大量・高密度化するために、遅延の大きい処理を徹底的に非同期化し、外部化する。
- ネットワーク送受信処理、暗号復号化処理、圧縮展開処理は外部化
- データベースやストレージへのアクセス機能も外部化
- 遅いI/Oを可能な限り外部化し、メモリ上の処理に集中

MOBやNPCなども外部化する傾向（Socket経由のI/Oはどうしても残るが、LAN内での少数TCPセッションを用いた通信は十分軽量）

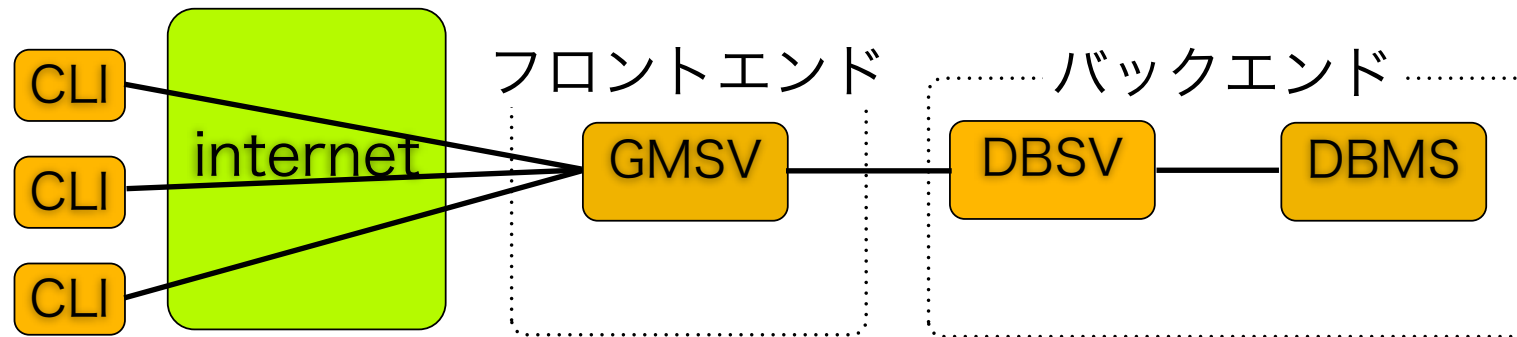
最も重要なI/O：DBアクセス

- **～1998** DBMSを使わずバイナリファイル等にキャラデータを保存
- **～2004** DBMSを使うがBLOBにデータを保存。DBMSをKey-Valueストレージとして使う。
- **～現在** DBMSを使い、多数のテーブルと多数のカラムにデータを保存。今日の実例でも40テーブル以上、合計数百カラム以上を使用。

現在よく使われるプロセス分割法



各サーバの役割と略称



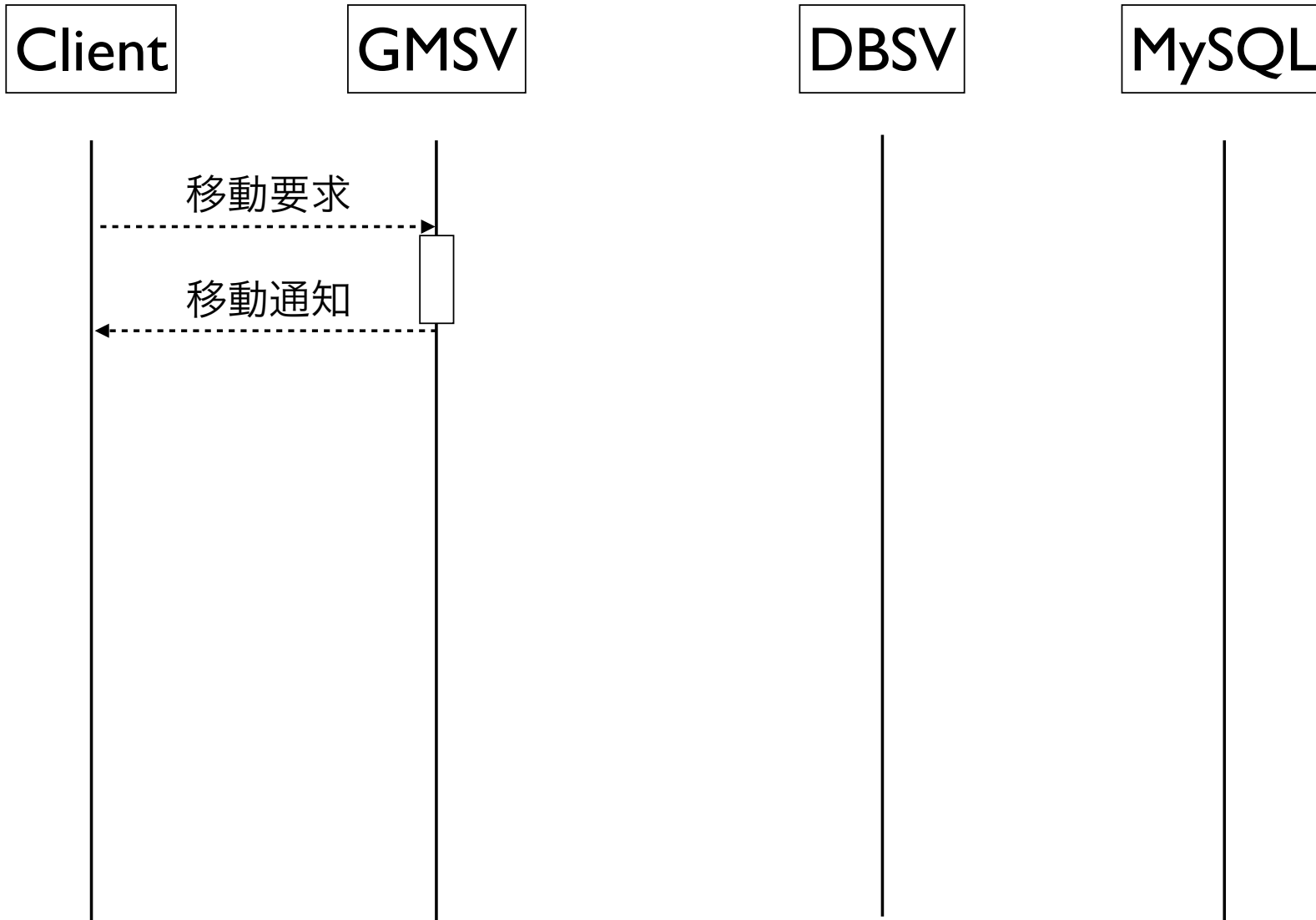
- CLI：ゲームクライアント
- GMSV：ゲームロジックを実装するメインのサーバ
- DBSV：GMSVからの書き込みを一元化・非同期化するサーバ
- DBMS：MySQLなど

DBアクセスの処理シーケンス

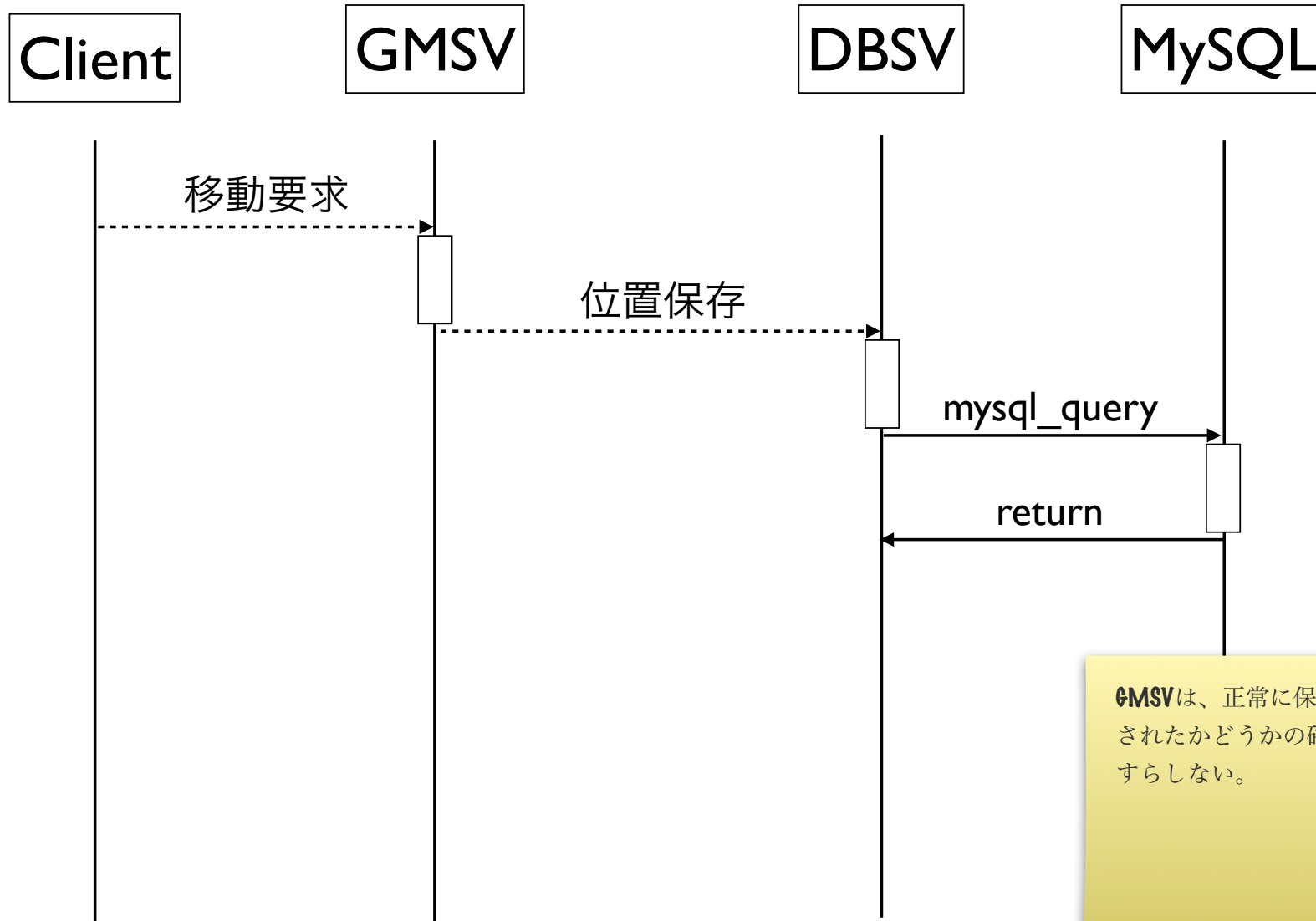
1. GMSVのみで完結
2. GMSV → DBSV 投げっぱなし
3. GMSV → DBSV → GMSV 1回往復でエラー確認
4. GMSV → DBSV → GMSV → DBSV .. 複数回往復

今日の例（真・女神転生**IMAGINE**, パンドラサーガでは、すべて**4**を避けている）

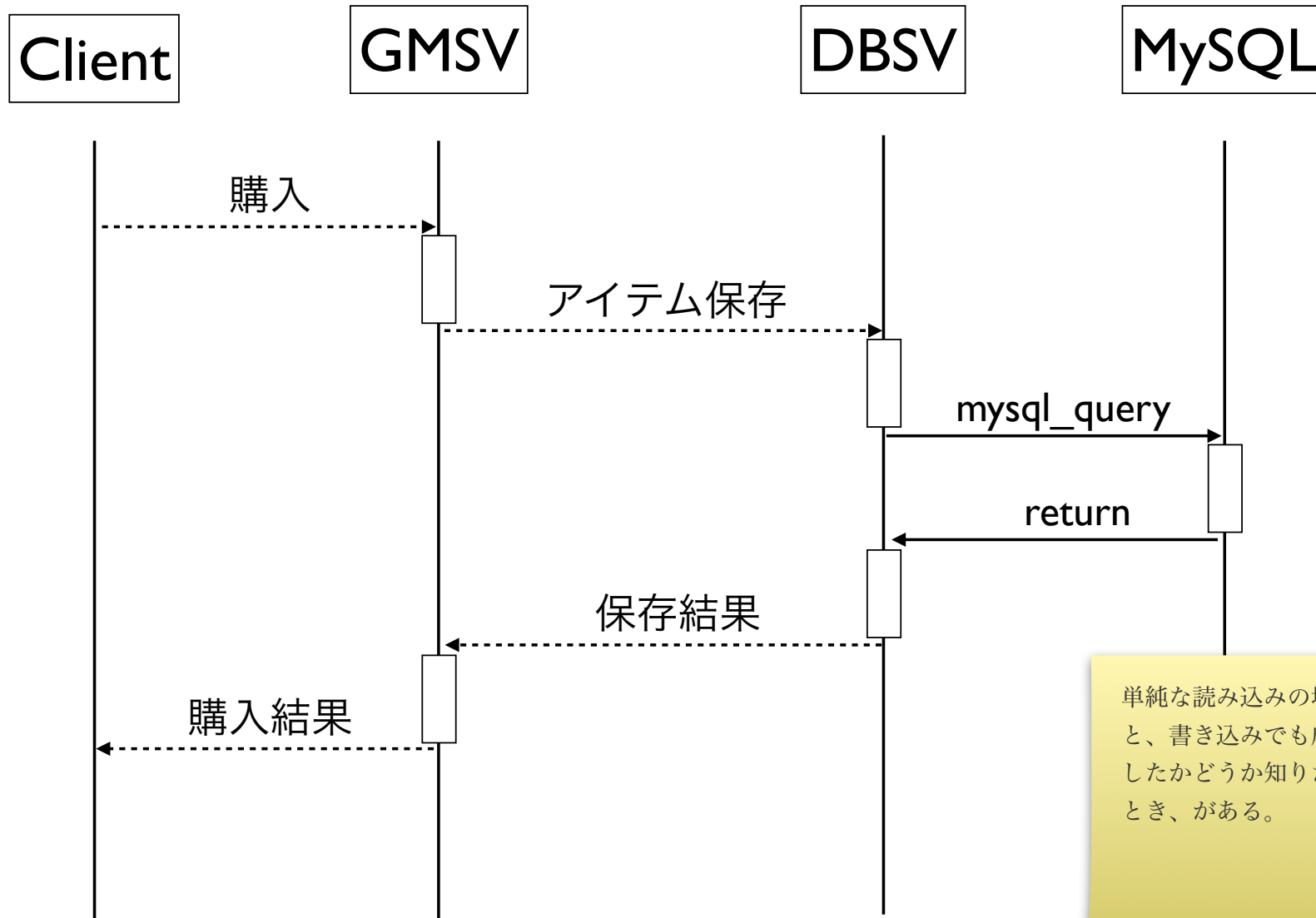
GMSVのみで完結



DBSVに投げっぱなし

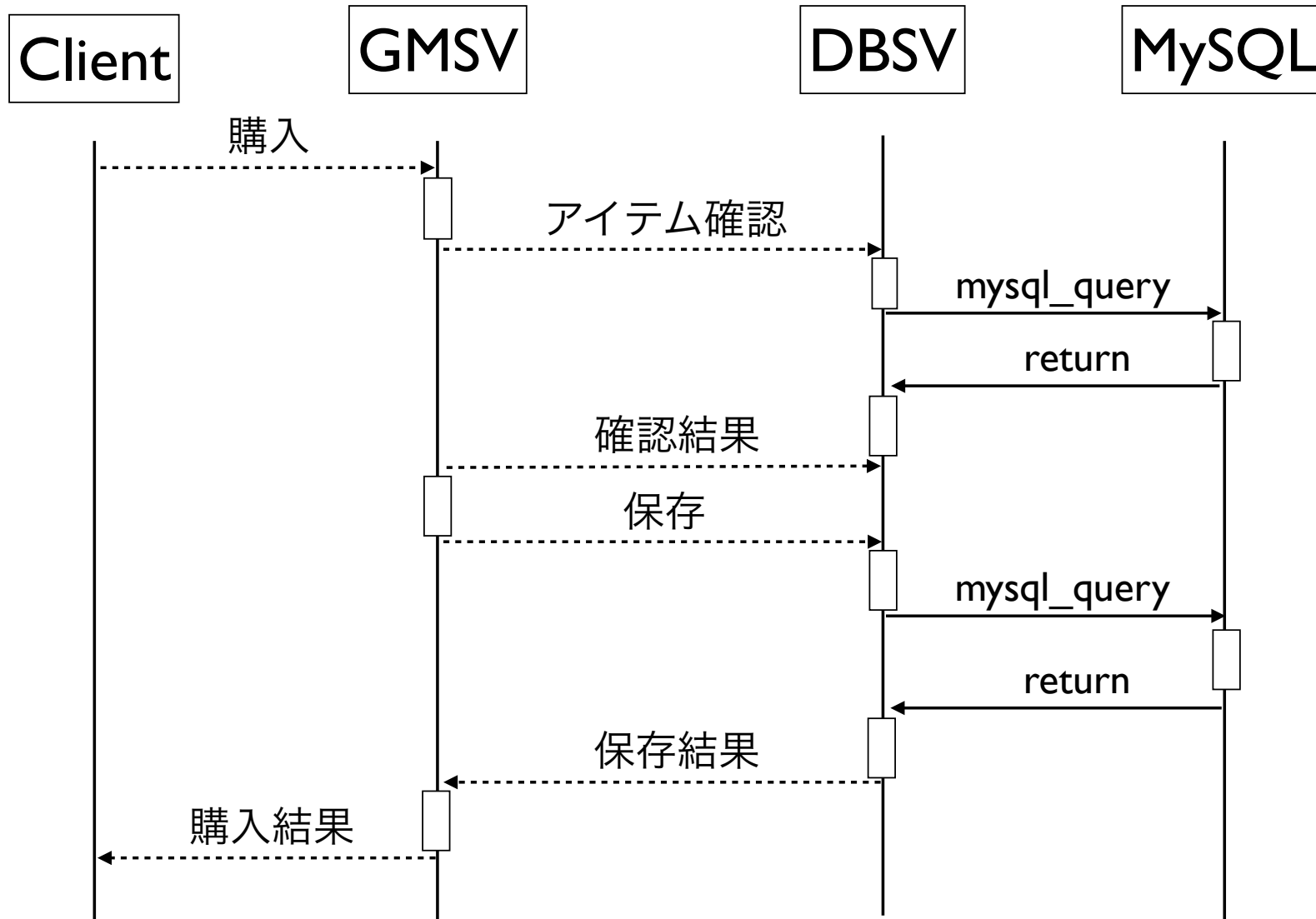


DBSVとGMSVで1 往復



単純な読み込みの場合
と、書き込みでも成功
したかどうか知りたい
とき、がある。

N回往復



関心

- GMSV, DBSV の連携部分の書き方は定説が存在しない。
- 非同期プログラミングでN回往復をするのはコールバックが入り乱れ、状態管理が煩雑で、デバッグしづらい。
- 実行効率、開発効率、安全性、堅牢性、レスポンス

2. 真・女神転生IMAGINEの 場合

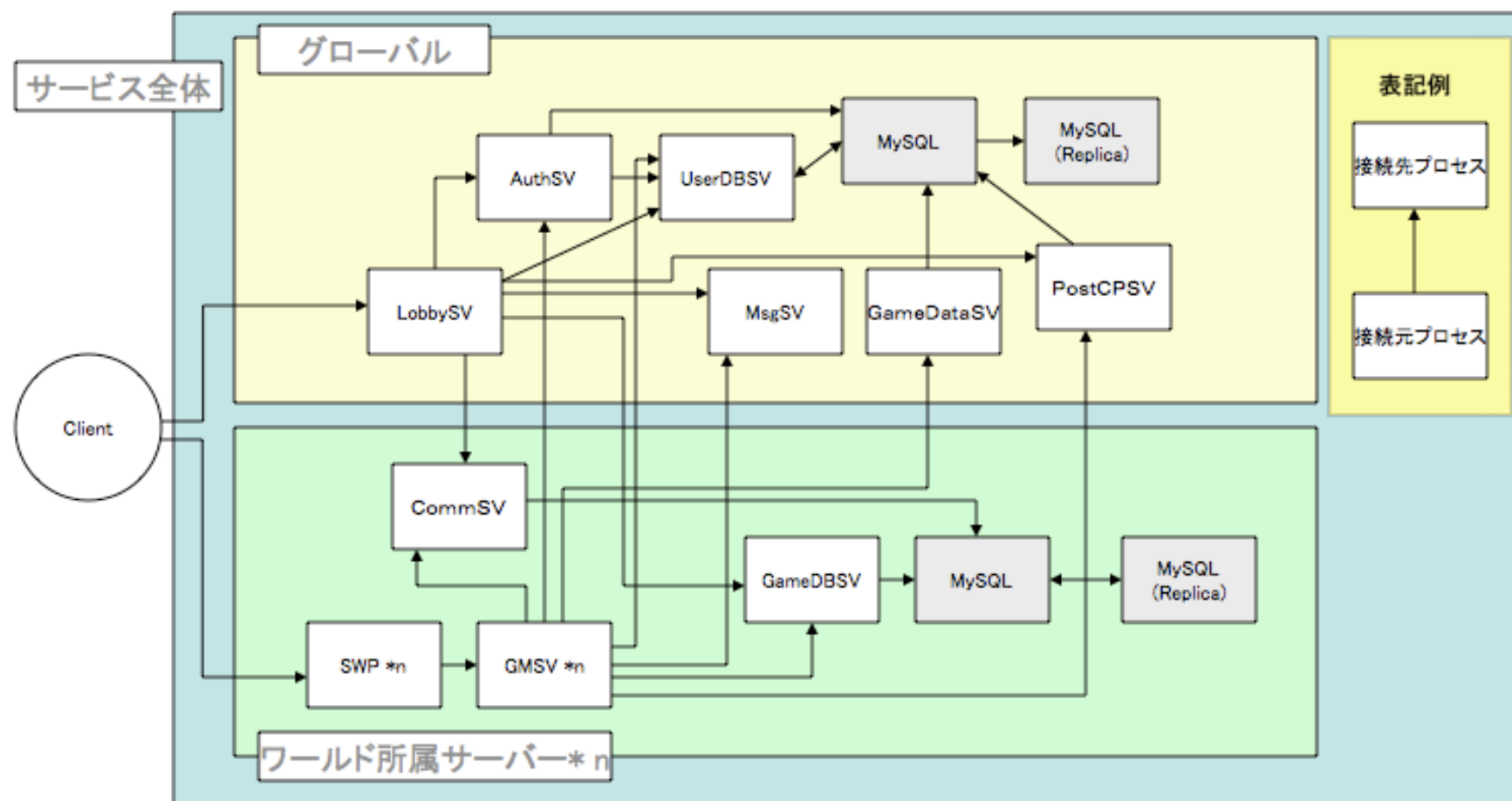
方針

- できるだけ往復の回数を少なく 1 回以下に。
- アイテム課金なので、DBSV側で、保存内容の確認を厳しくする。
- よく変更する部分をGMSVに集める。

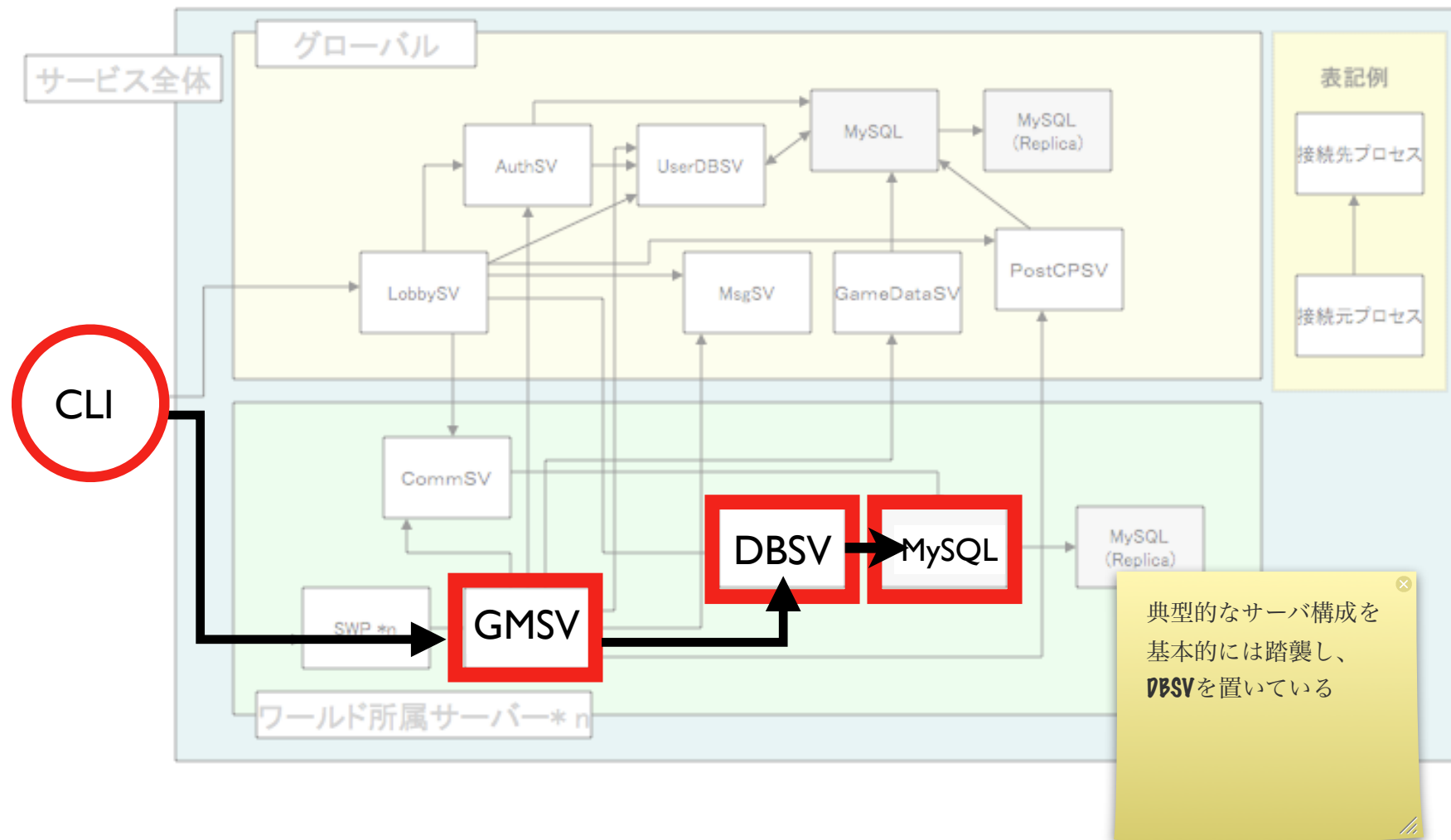
こう実装した

- アイテムの削除と生成に必要な情報をすべてセットにすることで、**すべてのリクエストを1往復以下**にした。
- DBSV側では、並列処理は行わず、同期的に処理
 - MySQLのトランザクションを用いて、削除、追加、更新がすべて成功したときのみDBに書き込み
- DBSVは1プロセスだけ。逐次処理

IMAGINE サーバ構成



IMAGINE サーバ構成



トランザクション例

- ショップでのアイテム購入

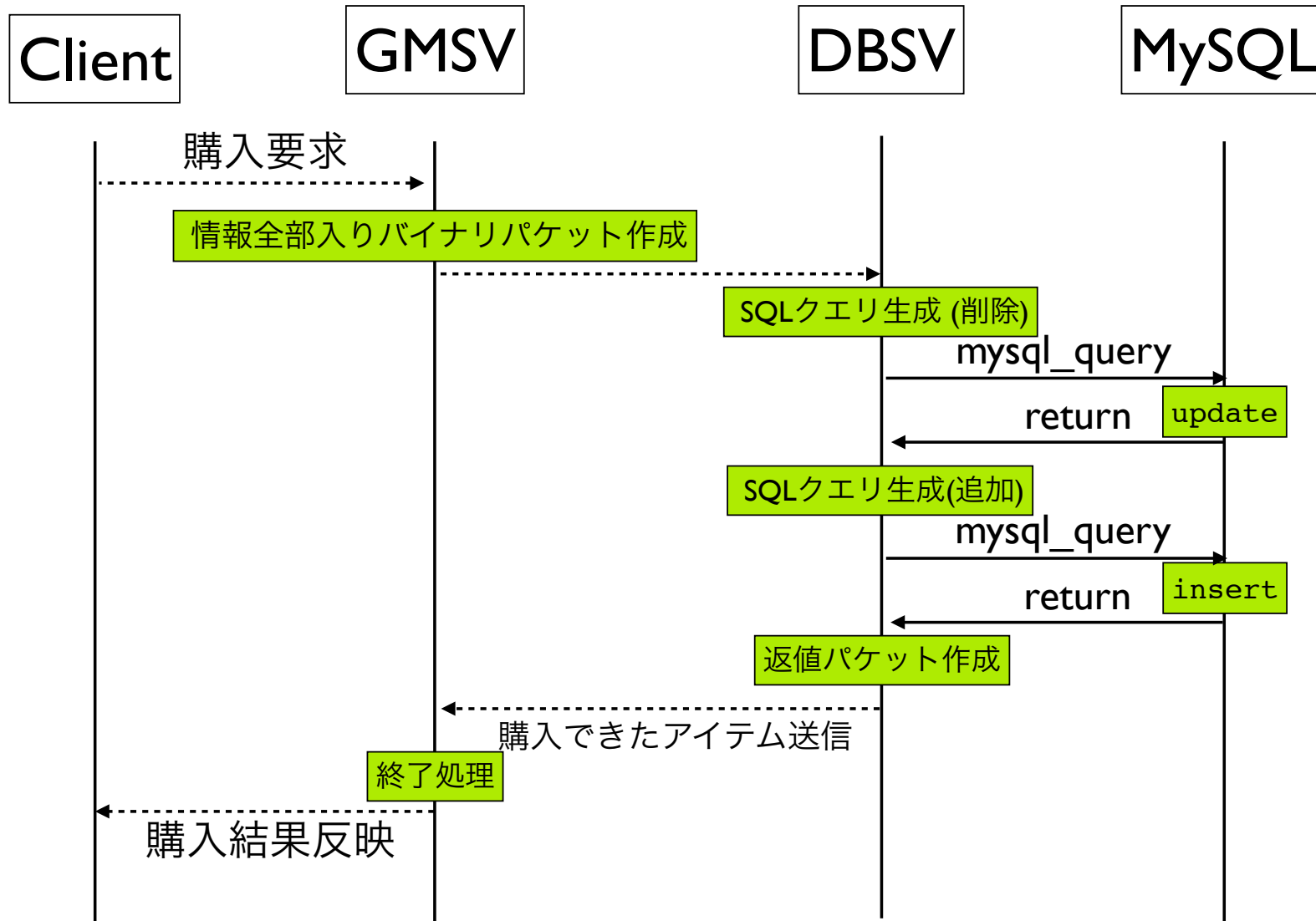
アイテム購入の考え方

- 購入とは：
 - 複数のアイテムを削除し、
 - 複数のアイテムを追加する。
- 削除、追加を1度に実行する“AlterItems”という汎用的なAPIをDBSVに実装。

1回のリクエストで全部やってしまうところがポイント

I 回往復で実装

2. 真・女神転生IMAGINEの場合



情報全部入りパケットの内容

- 生成するアイテム
 - 種類、個数、耐久値、エンチャント等、ゲームのアイテムに必要な全情報
- 削除するアイテム
 - ユニークID (の列)

GM5VからDBSVに
送るパケットの内容

バイナリパケット作成(GMSV)

```
CiPacketMaker pack(GDB_GSV::CS_AlterItems_Request);
```

```
pack.PackInt( add_count ); // 生成する個数 //
for( int i = 0; i < add_count; ++i )
{
    PackUShort( ItemIndex ); // アイテムインデックス //
    PackUInt( item.ItemID ); // アイテムID //
    PackUShort( item.Count ); // 所持数 //
    PackUShort( item.Endurance ); // 現在耐久値
    PackChar( item.MaxEndurance ); // 最大耐久値 //
    PackUShort( item.Exp ); // 熟練度 //
    PackChar( item.CareLevel ); // 手入れ度
    PackShort( item.Enchant[0] ); // エンチャント1 //
    PackShort( item.Enchant[1] ); // エンチャント2 //
}
```

追加するアイテム

```
pack.PackInt( delete_count ); // 削除する個数 //
for( int i = 0; i < delete_count; ++i )
{
    pack.PackLongLong(); // 削除するアイテムシリアル //
}
```

削除するアイテム

パケット受信(DBSV)

```
int CiAlterItemsRequest::Process(  
    int iConnectionID, CiUnPacker & unpack )  
{
```

まずUnpack

```
    int a_iRequestID;  
    int a_iCharSerial;  
    char a_iType;  
    unpack.UnPack( a_iRequestID );      // 要求の識別子  
    unpack.UnPack( a_iCharSerial );    // キャラクタシリアル
```

....

```
    // 指定したアイテムが存在してるか  
    // 持ちもののリストの位置指定は正しいか  
    // 個数が足りているか  
    // キャラが存在するか  
    // バッグ自体を削除しようとしていないか  
    // 既にあるアイテムをさらに生成しようとしていないか  
    // 持ち物の最大数を超えないか
```

オンメモリまたはSQLクエリを発行してチェックを徹底して行う

パケット受信続き(DBSV)

```
char a_pszSQL[QUERY_LENGTH_MAX + 1];
```

SQL生成

```
snprintf( a_pszSQL, sizeof( a_pszSQL ),
  "UPDATE item_container_part SET item_serial%d=%lld,item_id%d=
  %u,item_count%d=%u, item_endurance%d=%u, ....."
  " WHERE user_serial=%d AND char_serial=%d AND container_type=
  %d AND container_serial=%lld AND part_no=%u",
  i_iPos, i_Item.ItemSerial, i_iPos, i_Item.ItemID,
  i_iPos, i_Item.Count, i_iPos, i_Item.Endurance,
  .....
  i_iUserSerial, i_iCharSerial,
  i_iContainerType, i_iContainerSerial, i_iPartNo );
```

```
if( GetDBConnection().Query( a_pszSQL, strlen( a_pszSQL,
sizeof( a_pszSQL ) ) ) )
{
  ASSERT( 0 ); return -1;
}
```

```
return 0;
```

```
}
```

MySQL SQLクエリ発行

結果

- DBSV 1 プロセスで1000以上の同時接続に対応できている。
- 2年以上ほとんど修正せず、安定している。
- プレイヤー間のトレードも同様に実装しているのでそのまま並列化はむずかしい。現状ではワールドを追加して対応できるので、並列化の必要性がない。

3. パンドラサーガの場合

方針

- DBSVを極度に簡素化する。
- 必要なチェックは全部GMSV側でやってしまう。
- DBSVはGMSVが言ってきたことをただ受け入れる。

こう実装した

- 書き込みは、可能な限り投げっぱなし。
- GMSV側で疑似トランザクションを実装。
- DBSV側では、並列処理は行わず、同期的に処理。★
- DBSVは1 プロセスだけ。★
- 上記を実現するLibPSTというツールを用意した。

★のところは、
IMAGINEと共通のところ。

サーバ構成

- 女神転生IMAGINEと基本的には同じ。
- GMSV - DBSV - MySQL の構成を採用。

中核となるLibPST

- ゲームデータの永続化コードを自動生成するツール。
- 目的：DBSVのコーディング作業を省略してGMSVのコーディングに集中できるようにすること。

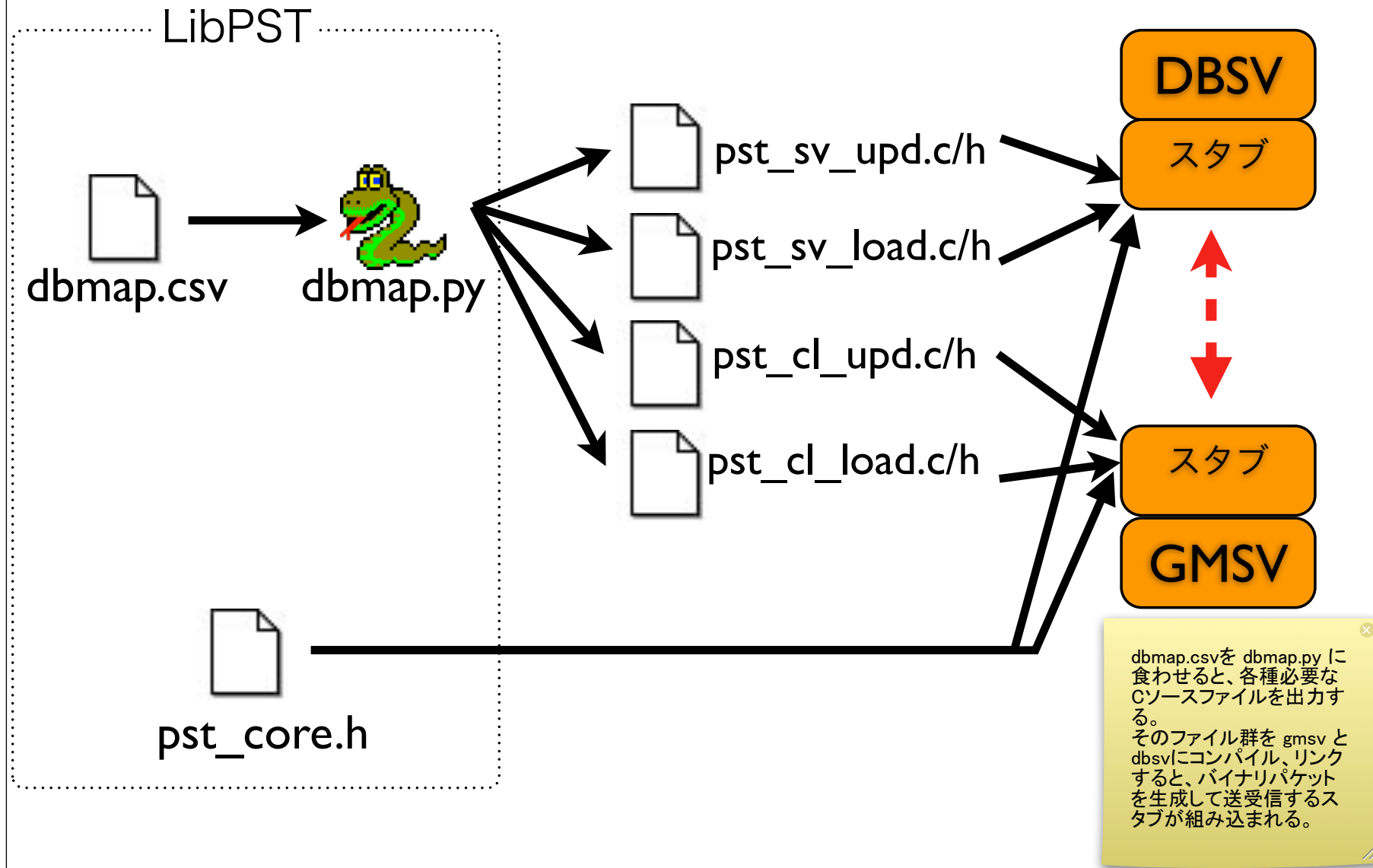
LibPSTの特徴

- 必要な最低限の機能だけに絞っている
- 機能
 - O/R Mapper
 - トランザクション
 - DB構成バージョン管理



pythonコードひとつ、
入力フォーマット1
個、あとはいくつかの
Cヘッダ

実装の概要



O/R Mapper

- テーブルと構造体定義が一対一に対応している
- DBにアクセスするには出力された構造体を引数に渡す

CSV入力ファイル

テーブル名など

```
#op,db_table,auto_increment  
entry_table,chara,Y
```

型

主キー

インデクス

```
#op,db_table,db_field,type,key,index  
entry_field,chara,charaid,int32,Y,N  
entry_field,chara,userid,int32,N,Y  
entry_field,chara,name,string,N,N  
entry_field,chara,mapid,int32,N,N  
entry_field,chara,x,int32,N,N  
entry_field,chara,y,int32,N,N  
entry_field,chara,z,int32,N,N  
entry_field,chara,money,uint64,N,N
```


出力：構造体定義

dbmap.csv



pst_common.h

```
entry_field,chara,charaid,int32,Y,N
entry_field,chara,userid,int32,N,Y
entry_field,chara,name,string,N,N
entry_field,chara,mapid,int32,N,N
entry_field,chara,x,int32,N,N
entry_field,chara,y,int32,N,N
entry_field,chara,z,int32,N,N
entry_field,chara,money,uint64,N,N
```

```
typedef struct pst_chara
{
    int32_t charaid;
    int32_t userid;
    const char *name;
    int32_t mapid;
    int32_t x;
    int32_t y;
    int32_t z;
    uint64_t money;
} pst_chara_t;
```

各カラムがCSVに 1 対 1 対応

出力：操作関数

```
int pst_sv_load_chara(  
    struct pst_sv_load *load,  
    int32_t charaid  
);
```

主キーを用いたload

```
int pst_sv_load_chara_by(  
    struct pst_sv_load *load,  
    const char *where  
);
```

自由なwhere文を挿入

```
int pst_sv_load_chara_by_userid(  
    pst_sv_load_t *load,  
    int32_t userid  
);
```

インデクスを使う関数

出力：パケット生成関数(GMSV)

```
int pst_cl_create_chara(  
    struct pst_txn *_txn,  
    int32_t charaid,  
    int32_t userid,  
    const char *name,  
    int32_t mapid,  
    ...  
    uint64_t money  
)  
{  
    pst_cl_txn_t *txn = pst_cl_get_txn(_txn->txnid);  
  
    PST_SET_INT32(txn, 1);  
  
    PST_SET_INT32(txn, charaid);  
    PST_SET_INT32(txn, userid);  
    PST_SET_STRING(txn, name);  
    PST_SET_INT32(txn, mapid);  
    ...  
    PST_SET_UINT64(txn, money);  
  
    return 0;  
}
```

バッファ確保

関数のID

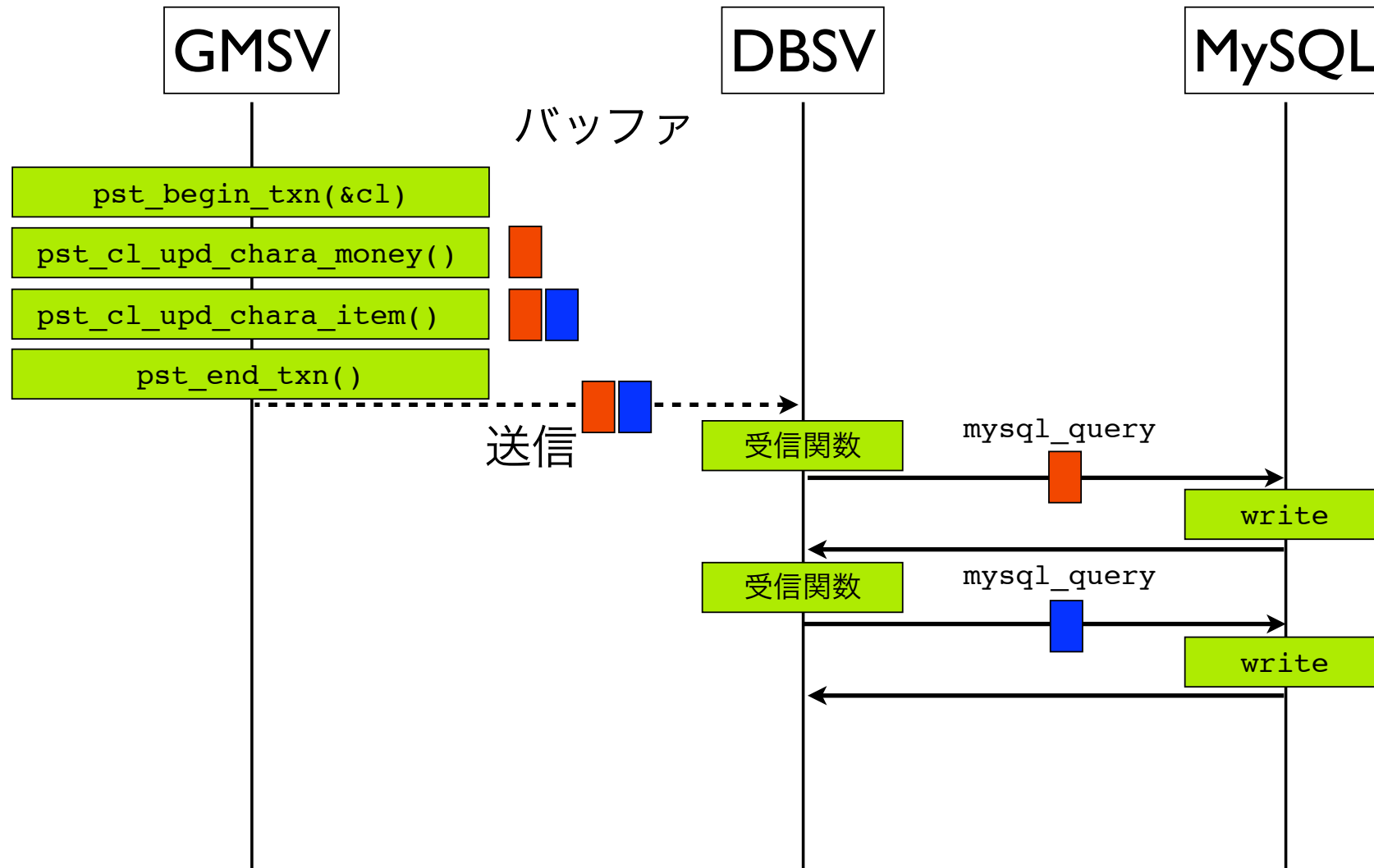
引数を順次
シリアライズ

トランザクション

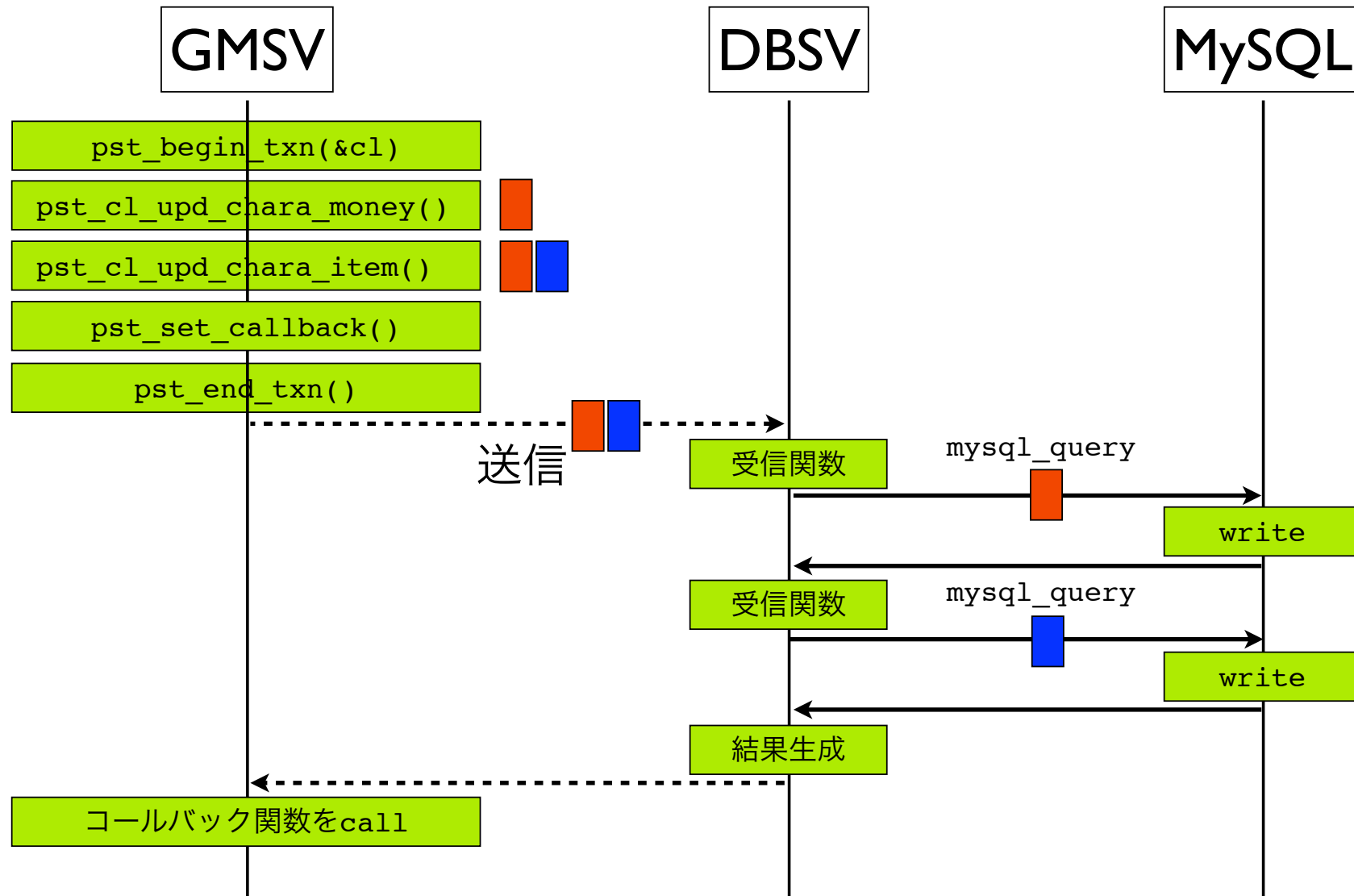
1. `pst_begin_txn()` : バッファ確保(ネスト可)
2. `pst_cl_create|upd|load_テーブル名()` : バッファリング
3.
4. `pst_end_txn()` : 送信+バッファ開放

`pst_begin_txn()`, `end`関数
は`pst_core`で定義され
ている。

トランザクション：投げっぱなし

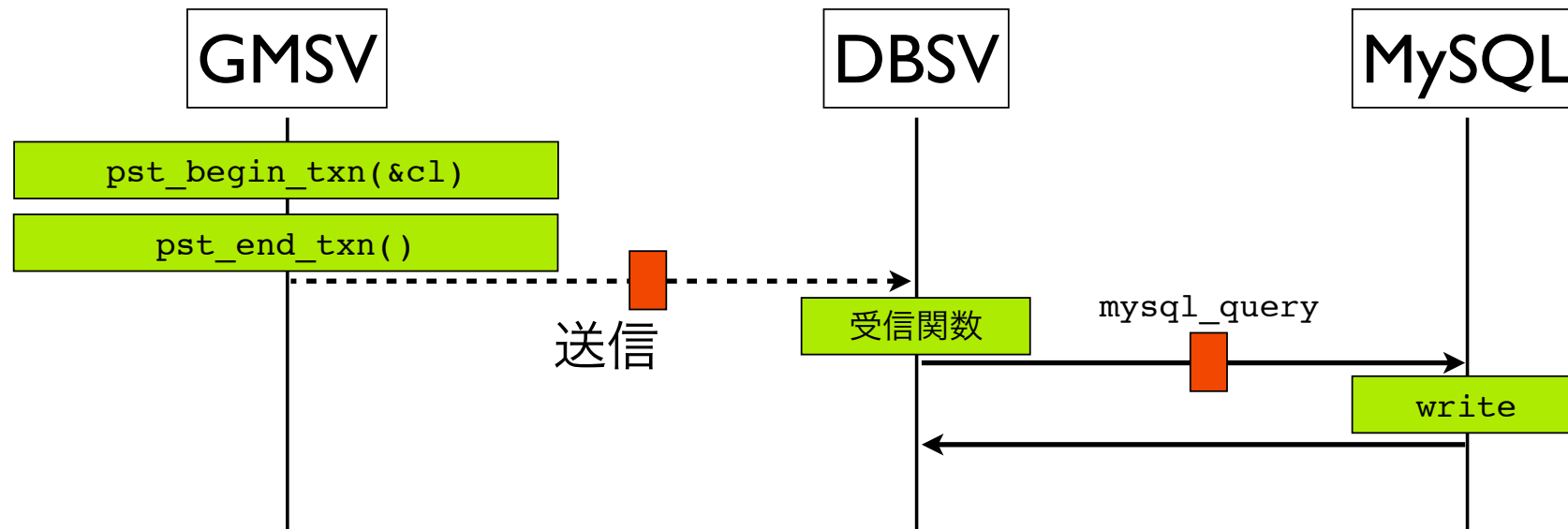


トランザクション：返り値あり



例 1 : キャラ座標更新 (GMSV)

```
if (mapidが違う || 以前の保存から2000m離れた)  
{  
  client_begin_txn(cl);  
  
  pst_cl_upd_chara_x(&cl->txn, cl->chara.id, pos->x);  
  pst_cl_upd_chara_y(&cl->txn, cl->chara.id, pos->y);  
  pst_cl_upd_chara_z(&cl->txn, cl->chara.id, pos->z);  
  
  client_end_txn(cl);  
}
```



例 2 : 店でアイテム購入 (GMSV)

```
int shop_buy(client_t* cl, int itemid, int amount, int price)
{
    //xxx所持数制限などメモリ上で事前チェック

    ...
    //xxxメモリ上で整合性が取れたのでtxn開始OK

    client_begin_txn(cl);

    client_update_money(cl, cl->money - price);

    item = item_create(&cl->txn, itemid, amount);
    client_item_add_to_inventory(cl, item);

    client_end_txn(cl);

    return 0;
}
```

最上位トランザクション開始

所持金減額

アイテム追加

例 2 : 店でアイテム購入 (GMSV)

```
//所持金更新
int client_update_money( client_t *cl, uint64_t money )
{
    //xxx所持金不整合などメモリ上で事前チェック
    ...
    //xxxメモリ上で整合性が取れたのでtxn開始OK

    client_begin_txn(cl);
    cl->money = money;

    pst_cl_upd_chara_money(&cl->txn, cl->chara.id, money);

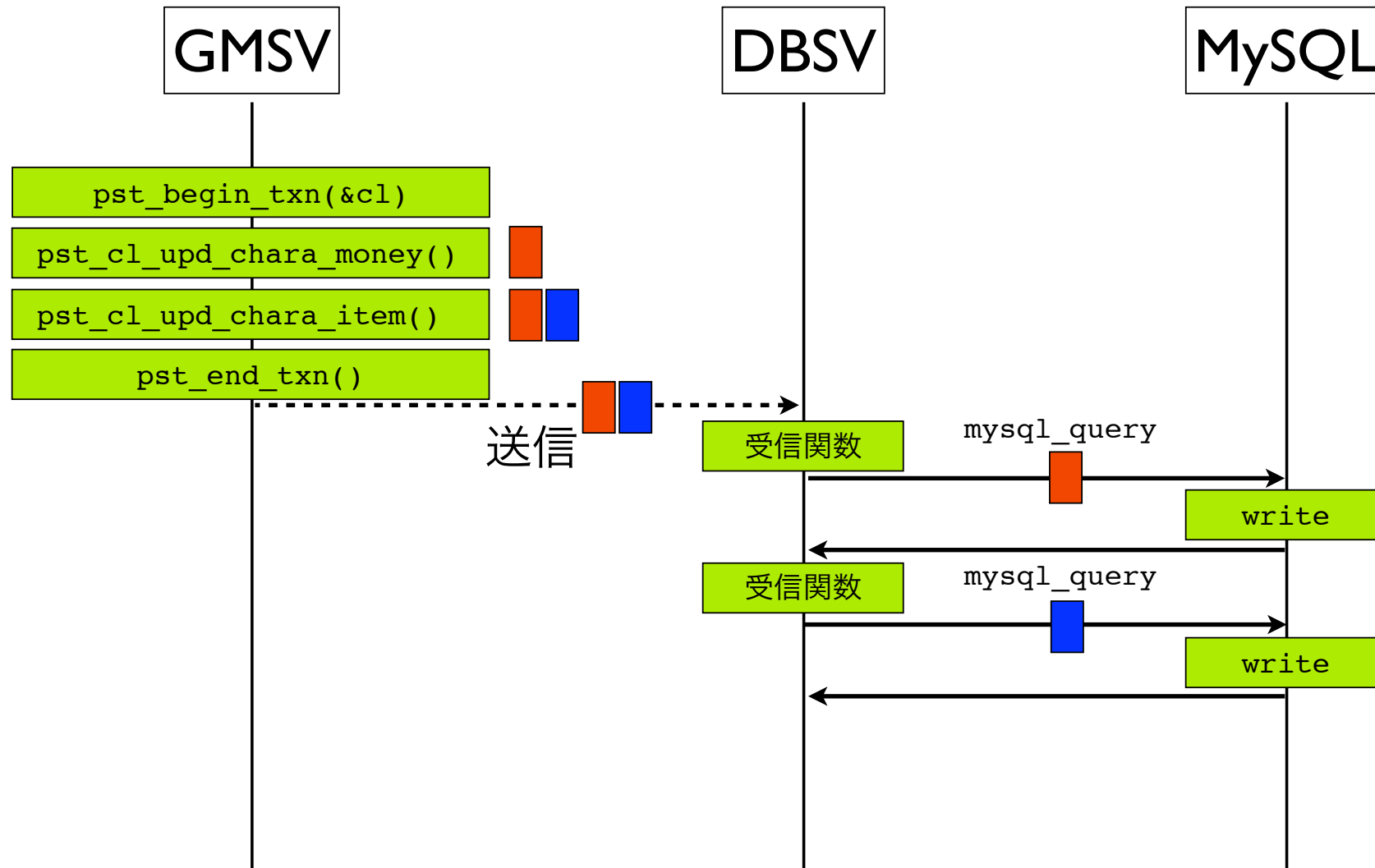
    client_end_txn(cl);
    return 0;
}
```

ネストされたトランザクション開始

```
int client_item_add_to_inventory( client_t *cl, uint32_t itemid)
{
    // 同様に トランザクションを使う (省略)

}
```

例 2：店でアイテム購入



例 3 : キャラクター作成(GMSV)

```
//キャラクター作成
int client_create_chara_recv(
    client_local_t *ct, //!< 要求元クライアント
    const char *name //!< 作成するキャラクター名
)
{
    //初期パラメータ等をチェック
    ...

    pst_txn_t txn = {};
    pst_begin_txn(&txn);

    pst_cl_create_chara( &txn, 0, ct->userid, name, 0, 0, 0, 0, 0);

    pst_set_txn_callback(&txn, client_create_chara_exec,
        (void *)ct->userid);

    pst_end_txn(&txn);

    return 0;
}
```

create要求

コールバック登録。名前の重複で失敗することがあるため。

例 3 : キャラクター作成(GMSV)

//キャラクター作成結果のコールバック関数本体

```
int client_create_chara_exec(  
    void *argp,  //!< ユーザID  
    int result  //!< 結果  
)  
{
```

クライアントを探す

```
    client_local_t *ct = (client_local_t *)  
                        client_lookup_by_userid((int)argp);  
    if(!ct || ct->state != CL_STATE_CREATE)  
    {  
        return PND_ERR_GENERIC;  
    }
```

```
    proto_msg_chara_create_r_send(ct->circ, result, chara->slot);
```

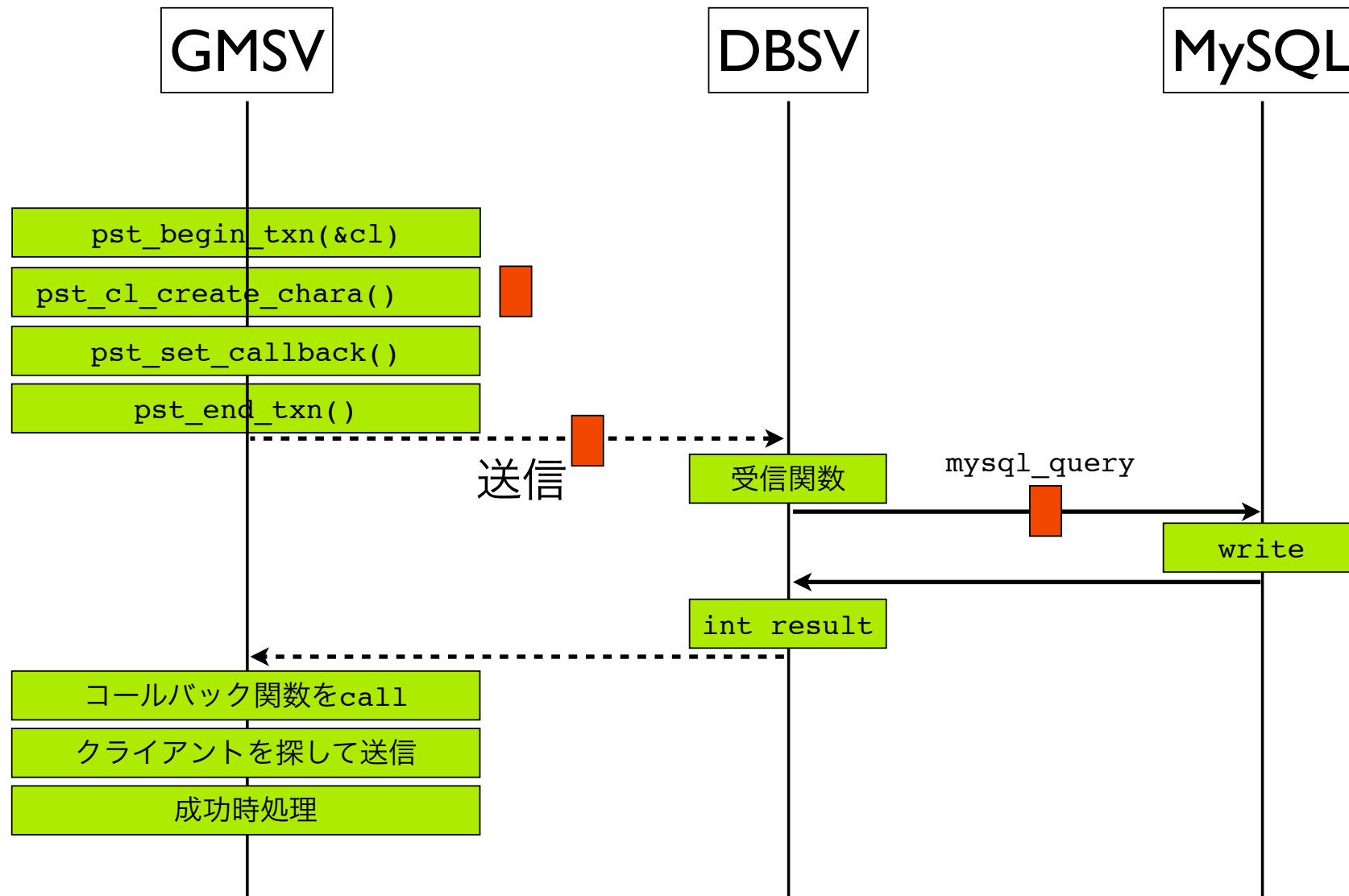
クライアントに送信

```
    if(result > 0)  
    {  
        chara->id = result;  
        ...  
    }
```

成功時処理

```
    return 0;  
}
```

例 3 : キャラクター作成



結果

- DBSV 1 プロセスで同時接続3000、200トランザクション/秒
- MySQLのCPU負荷は30%程度、DBSVはほぼ負荷なし
- カラム変更したときには、クライアント変更、GMSV、DBSV、プロトコル、テーブル定義修正、の作業があったが、後ろ3つを一回にできた。
- パンドラサーガ、aisp@ceで使われ、十分な性能とメンテナンス性、開発効率を実現できている。

4.まとめ

共通点と相違点

- 共通点
 - DBSVは1 プロセス、並列化無しの同期処理
 - すべてのリクエストを1 往復以下にしている。
 - 入り乱れるコールバックを回避している。
- 相違点
 - トランザクションの扱い：パンドラサーガでは、擬似的にGMSVでトランザクションを実装し、DBSVの単純化している。

もとの関心

- GMSV, DBSV の連携部分の書き方は定説が存在しない。
- 非同期プログラミングでN回往復をするのは煩雑で、デバッグしづらい。
- 実行効率、開発効率、安全性、堅牢性、レスポンス

関心に対しては・・・

- MMOGでは、最初に全部読み、
- できる限りメモリ上で処理をし、
- 差分は「投げっぱなし」で保存する
- この方針によって
 - GMSVでのオンメモリのゲームロジック作成に専念できる度合いが高まり、開発効率は向上した。
 - 非同期なのでレスポンスは良く保てている。
 - 実行効率と堅牢性は今のところ足りている。

著作権等について

- ・ 本資料に記載されているデータ等の無断での転用転載はご遠慮ください。
- ・ 本資料に記載されている会社名、製品名は、各社またはその提供元の商標または登録商標です。
- ・ 本資料に記載されている製品「真・女神転生IMAGINE」、「パンドラサーガ」、「aisp@ce」は、以下の各社の著作物です（Webからの転載）。
- ・ 真・女神転生IMAGINE： COPYRIGHT (C) ATLUS / (C) CAVE
「真・女神転生IMAGINE」は、株式会社アトラスからのライセンス契約に基づいて、株式会社ケイブが独自に開発、及び運営を行っております。
- ※ 「女神転生」、「真・女神転生」及び「女神転生IMAGINE」は、株式会社アトラスの登録商標です。
- ・ パンドラサーガ： (c)2006-2009 GR・GDH / イズミプロジェクト Developed by HEADLOCK Inc.
- ・ aisp@ce (c)2007 『ai sp@ce』 製作委員会

質疑応答など

- ご静聴ありがとうございました。
- メールでも質問ください

ringo@ce-lab.net