SIGVerse（社会的知能発生シミュレータ）の基本機能拡張

構造説明書

2009 年 12 月
株式会社　数理システム

---

1. 概要

本稿は「SIGVerse（社会的知能発生シミュレータ）の基本機能拡張」のシステム設計，実装詳細について解説したものである．２章においてまずシステム構成について述べてシステム全体の概略を示し，３章に仮想世界の設定方法，４章においてユーザによるコントローラ実装方法について解説して，ユーザがシステムを使用するという視点での解説を行っている．５章においてシミュレーションサーバ，６章においてビューア，7 章において音声認識サービスプロバイダの実装についてそれぞれ詳細を述べ、８章でシミュレーションサーバと各種クライアント間のデータ転送について解説を行っている．

2. システム構成

「SIGVerse（社会的知能発生シミュレータ）の基本機能拡張」は，仮想世界を作成しその中でエージェントを自律動作させその振る舞いを観測するためのシステムである．

本システムはクライアント－サーバシステムであり，プロセス内に仮想世界の情報を保持し，クライアントからのリクエストに応じてデータの転送や各処理を実行するシミュレーションサーバと，サーバに接続して協調動作する各種クライアントにより構成されている．クライアントには，シミュレーションサーバに接続して仮想世界を視覚化するビューア，エージェントの動作を制御するコントローラ，シミュレーションサーバに不足している機能を補うサービスプロバイダがある．シミュレーションサーバは，サービスプロバイダのネームサービスを実装しており，コントローラがサービスプロバイダと直接通信できるよう必要な情報を提供する．

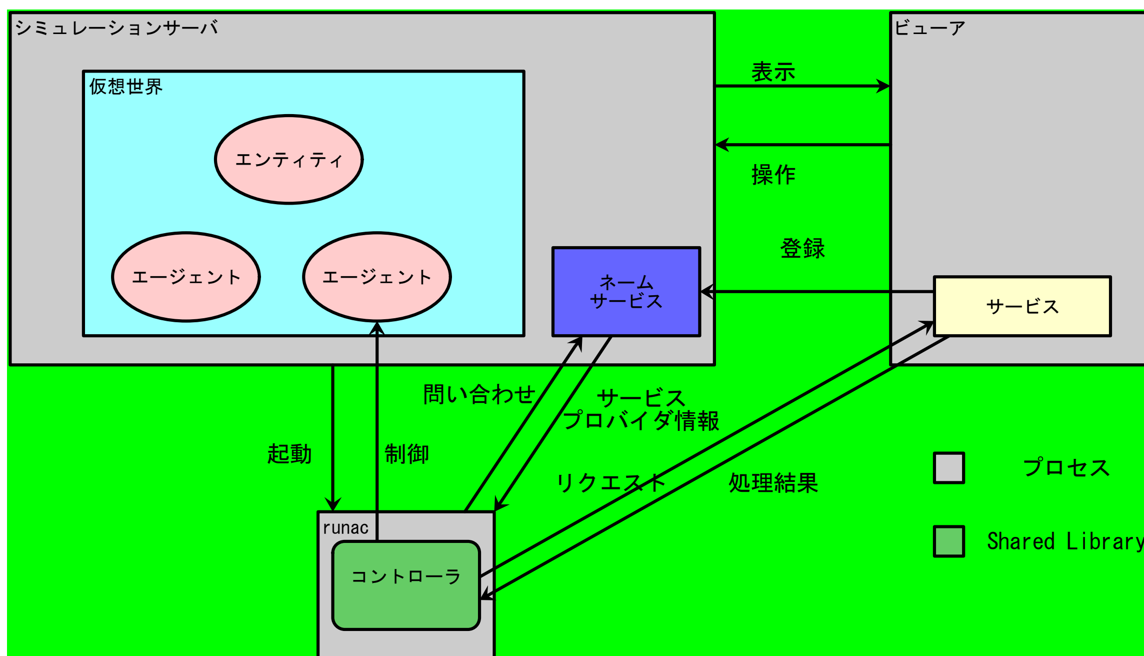

図 1　仮想世界シミュレータシステム構成図

2.1.　シミュレーションサーバ

シミュレーションサーバは仮想世界を表すデータをそのプロセス内に有している．仮想世界には人・ロボットなど自立して動作するエージェント(Agent)と，机・リンゴなど自立動作しないエンティティ(Entity)を登場させることができる．エージェント・エンティティをまとめてシミュレーションオブジェクトと呼ぶ．仮想世界には現実世界と同様に重力が働いており，シミュレーションオブジェクトには鉛直下向きの加速度が作用する.重力加速度ベクトルには任意の値を(場合によっては鉛直上向きの重力も)設定可能となっている.

2.2.　ビューア

ビューアはユーザとシミュレータの接点となり，ユーザはビューアを通じてシミュレーションの開始・停止の操作や，シミュレーション中の仮想世界の観測，場合によっては仮想世界への干渉をおこなうことができる．ビューアは起動時にシミュレーションサーバに接続し，シミュレーションサーバ内の仮想世界の状態を表示する．仮想世界内にシミュレーションオブジェクトを選択状態にすると，その状態を表示する.

2.3.　コントローラ

仮想世界に存在するエージェントにはその振る舞いを制御するコントローラを割り当てることができる．コントローラは C++により実装し，Shared Library としてコンパイルする．runac コマンドはコントローラの実装を含んだライブラリを動的にロードしてシミュレーションサーバ内の指定のエージェントに割り当て，エージェントを操作する.

2.4.　シミュレーションサーバとクライアントの通信

シミュレーションサーバは起動時にクライアント接続ポートをオープンする．ビューア，コントローラ，サービスプロバイダはこのポートに対してコネクト要求をする.

クライアントは，シミュレーションサーバとコマンドポートとデータポートの２本のコネクションを張る．コマンドポートはサーバからのリクエスト・コントローラメソッド呼び出しのパケットを受信するためのポートである．データポートは，リクエスト処理中またはコントローラメソッドの呼び出し中に，サーバまたは他のクライアントに対してパケットを送受信するためのポートである.

2.5.　サービスプロバイダ

シミュレーションサーバはクライアントからのリクエストに答えてデータの提供や各処理を行う．しかしながら，サーバマシンの機能的制限により，一部の機能をシミュレーションサーバにおいて実現できない．たとえば，任意の視点における画像取得や視界内のシミュレーションオブジェクトの取得など，画像処理に関する処理はシミュレーションサーバの動作する Linux 上での実装が難しい．そこでこれらの機能を実現するアプリケーションを他の環境において実装する．これを「サービスプロバイダ」と呼ぶ，現在の実装では，ビューアが Ogre の機能を利用してエージェント

視点での画像キャプチャー・エージェント検出など画像に関する機能を，julius-sp が音声認識機能を実装し，サービスプロバイダとしての役割を果たしている．
　サービスプロバイダは起動時にソケットポートを開き，クライアントからの処理リクエストを受け付ける．リクエストに対する処理を終了すると，クライアントとのソケットコネクションを切断する．

2.6.　ネームサーバ

　ネームサーバは，サービスプロバイダの IP アドレス・ポート番号・提供している機能を一元管理する．サービスプロバイダは起動時に自身の情報をネームサーバに登録する．コントローラはサービスプロバイダが提供している機能を使用する際，まずネームサーバに対し問い合わせを行い，使用したい機能を提供しているサービスプロバイダの IP アドレス・ポート番号を取得する．この情報を元にしてコントローラはサービスプロバイダに接続し，処理リクエストを送信して望む結果を得る．現在の実装では，シミュレーションサーバ本体がネームサーバとしての機能を果たしている．
　ネームサーバを使用することで，サービスプロバイダとクライアントの緩い連携が可能となり，システムの頑健性を保証し，一部のサービスプロバイダが異常終了しても，シミュレーションシステム全体が止まってしまわないような構成となっている．

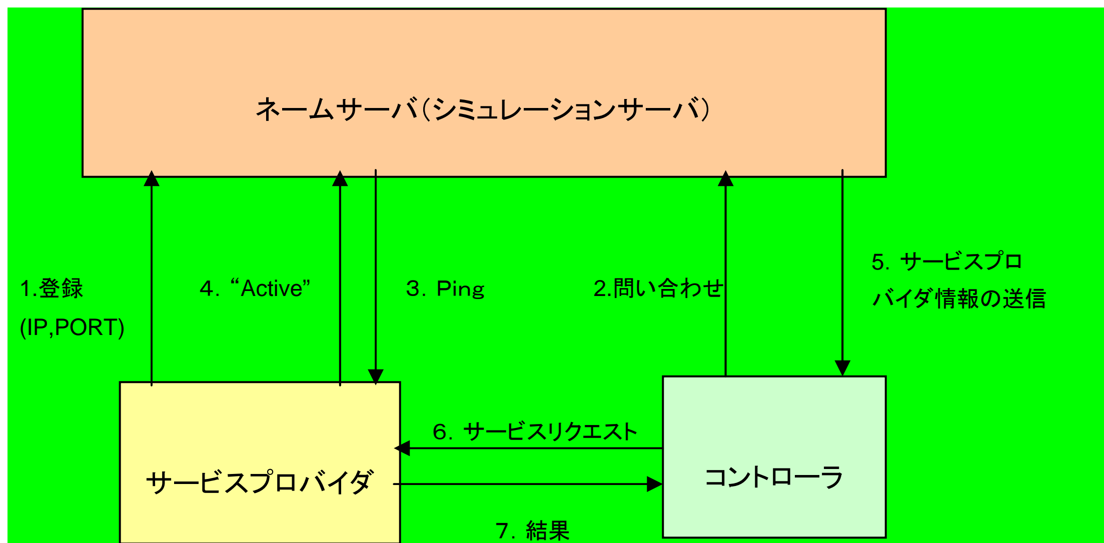

図 2 コントローラからのサービスプロバイダの使用

以下，コントローラからサービスプロバイダの提供する機能の呼び出し手順(図 2)について説明する．

1.　サービスプロバイダの登録

サービスプロバイダは起動時に，ネームサーバに対してサービスプロバイダ情報(自分の提供するサービス・IP アドレス・ポート番号)を登録する．

2.　コントローラからネームサーバへの問い合わせ
コントローラは，サービスプロバイダの提供する機能を使用する時，ネームサーバに対して，使用する機能を提供するサービスプロバイダ情報の問い合わせを行う.

3.　サービスプロバイダの稼働確認
２の要求に対してネームサーバは，登録済みのサービスプロバイダ情報から，コントローラの要求してきた機能を提供するサービスプロバイダの稼働状況を確認する.

4.　稼働状況の返答
サーバビスプロバイダは３の要求に対して，稼働している旨返答する.

5.　サービスプロバイダ情報の転送
コントローラの要求した機能を提供するサービスプロバイダ情報を転送する. コントローラはこの情報をキャッシュし，次回同一のサービスプロバイダを使用する際にはネームサービスへの問い合わせをしない.

6.　コントローラからサービスプロバイダへのリクエストの送信
サービスプロバイダ情報を元に，コントローラはサービスプロバイダにコネクションを張り，リクエストを送信する.

7.　処理結果の送信
サービスプロバイダはリクエストを処理し，その結果をコントローラに返送する. リクエストの処理が終了すると，コントローラ・サービスプロバイダ間のコネクションを閉じる.

3. 仮想世界とエージェント

3.1. 仮想世界の設定

仮想世界へのエージェントの配置，エージェントの初期状態などは世界ファイルにより指定する. 世界ファイルの詳細はマニュアルの「世界ファイルの作成」の項を参照のこと.

3.2.　エージェントの形状

エージェント・エンティティの形状ファイルの形式は，HAnim のデータ構造をそのまま XML に変換した独自フォーマットまたは OpenHRP 拡張を施した X3D フォーマットのファイルを使用できる.

3.3.　形状データから物理シミュレーションに適したデータ構造への変換

OpenHRP 拡張がなされた HAnim 形式のデータは，以下のような構造をしている.

```
+ Humanoid sample
  + Joint WAIST
    +Joint WAIST_JOINT0
      + Segment WAIST_LINK0
      + Joint WASIT_JOINT1
        + Segment WAIST_LINK1
    +Joint LLEG_JOINT0
      + Joint LLEG_JOINT1
        + Segment LLEG_LINK1
        + Joint LLEG_JOINT2
          …
    +Joint RLEG_JOINT0
      …
```

図 3 OpenHRP 拡張 HAnim データのサンプル

ODE ではセグメント → ジョイント → セグメント →…というように，セグメントとジョイントが交互にくる構造になっている必要がある．与えられた形状に対して ODE による物理シミュレーションを実現するため，以下のルールに従って OpenHRP 拡張 HAnim 形式のデータをツリー構造に変換し，サーバ内部で保持する．

I. ツリーのルートに”body”当名前のセグメントを追加する
II. ジョイントは兄弟ノードのセグメントと子ノードのセグメントを接続する
III. ジョイントの兄弟ノードがない場合，ブラインドセグメントを追加する

ブラインドセグメントはある程度の大きさを持つが，重さを持たず衝突判定の対象とならない．上記のルールに従うと，図 3 のデータは図 4 のようになる．

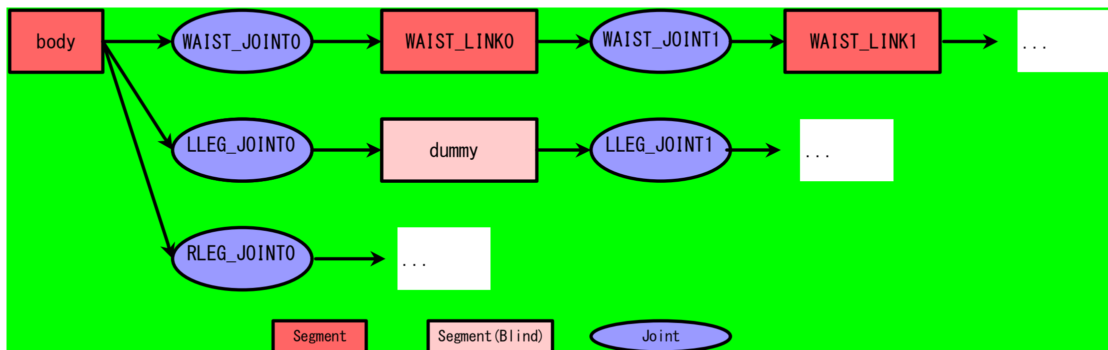

図 4 シミュレータ内での形状データの持ち方

4. ユーザによるコントローラの作成

ユーザはコントローラ開発用に提供されているライブラリを使用して，仮想世界内のエージェントを制御するコントローラを独自に実装することができる．コントローラは C++のクラスとして実装 Shared Library としてコンパイルする．runac コマンドは生成したコントローラを動的にロードして仮想世界内のエージェントにコントローラを割り当てる．

以下のサンプルでは MyController というコントローラクラスを作成している．

```
#include <Controller.h>

class MyController : public Controller
{
public:
	double onAction(ActionEvent &evt)
	{
		// ココにモデルの実装を書く
		return 0.0;
	}
}

extern "C"	Controller * createController ()
{
	return new MyController;
}
```

コントローラはControllerクラス(model/Controller.h)を継承して実装する．onActionはシミュレータで時間が進む毎にコントローラに対して呼び出されるイベントハンドラである．ここにエージェントの主な振る舞いを実装する．

4.1. イベントハンドラ

コントローラでは，シミュレーションの開始などのシミュレータのシステムイベントや，シミュレーション中に他のエージェントから話しかけられるなどのシミュレーション中に発生するイベントに対して，その反応をメソッドとして実装する．イベント毎に呼び出されるコントローラのメソッドが決められている．このようなイベントに対応付けられたメソッドをイベントハンドラと呼ぶ．イベントハンドラにはそれぞれ***Eventという型名の引数があり，この引数によりイベントの詳細を取得することができる．

イベントハンドラには以下のものがある．

◆ エージェントの初期化

onInit(InitEvent &evt);

シミュレーション開始前に一度だけ呼び出される．エージェントの初期化などに使用する．

	メソッド記述	説明
InitEvent	なし	-

◆ エージェントの動作

```
double      onAction(ActionEvent &evt);
```

シミュレーションステップ毎に呼び出されるメソッド．通常この関数内で，エージェントの動きの制御，エージェントの状態または周囲の監視を行う．返り値として次に onAction メソッドが呼ばれるまでの時間を返す．負の値を返した場合は，次のステップでも時間を置かずに同メソッドが呼び出される

	メソッド記述	説明
ActionEvent	double time()	シミュレーション時間を得る

◆ テキストの受信

```
void  onRecvText(RecvTextEvent &evt) {}
```

他のエージェントからテキストを受け取ったときに呼び出される．

	メソッド記述	説明
RecvTextEvent	const char *getCaller()	送り主名を得る
	const char *getText()	送られてきたテキストを得る
	Encode getEncode()	エンコードを得る(現在は ASCII のみ)

◆ 音声データの受信

```
void  onRecvSound(RecvSoundEvent &evt) {}
```

他のエージェントから音声データを受け取ったときに呼び出される．

	メソッド記述	説明
RecvSoundEvent	const char *getCaller()	送り主名を得る
	RawSound *getRawSound()	送られてきた音声データを得る

◆ メッセージの受信

```
void  onRecvMessage(RecvMessageEvent &evt) {}
```

他のエージェントまたは外部からメッセージ(命令)を受け取ったときに呼び出される．

	メソッド記述	説明
RecvMessageEvent	const char *getSender()	送り主名を得る
	int getSize()	送られてきた文字列数を得る
	const char *getString(i)	i 番目の文字列を得る

4.2. コントローラで使用可能なメソッド

コントローラでは次のメソッドが使用できる.

4.2.1. シミュレーションオブジェクトの取得

```
SimObj *    getObj(const char *name);
```

名前を指定してシミュレーションオブジェクトを得る.

4.2.2. シミュレーション世界内のオブジェクト名の取得

```
bool        getAllEntities(std::vector<std::string> &v);
```

シミュレーション世界内のすべてのエージェントまたはエンティティ名を取得する.

4.2.3. 視野内のシミュレーションオブジェクトの検出

```
bool        detectEntities(std::vector<std::string> &v);
```

視野内にいるエージェントまたはエンティティを検出する．視野内に 1 つ以上のエージェントまたはエンティティを検出した場合 true を返し，引数で与えたコンテナに検出したものの名前が追加される.

4.2.4. エージェント視点からの画像データを取得する

```
ViewImage * captureView(ColorBitType cbtype, ImageDataSize size);
```

cbType ： 画像データタイプ(COLORBIT_ANY または COLORBIT_24)
size ： 画像サイズ(現在指定できるのは IMAGE_320X240 のみ)

現在の視点から見える画像データを得る．画像データの取得に失敗した時には NULL を返す.
ViewImage オブジェクトを介して画像データのデータ形式，サイズ，画像データを取得できる.

4.3. エージェントの状態の変更

コントローラの各イベントハンドラ内で getObj メソッドを呼び出すと，シミュレーションサーバから指定のシミュレーションオブジェクトの状態をロードし，コントローラ内でその状態を参照で

きるようになる．シミュレーションオブジェクトを変更すると，そのイベントハンドラの呼び出し終了時にシミュレーションオブジェクトの状態がシミュレーションサーバに転送され，変更内容がシミュレーションサーバ中のシミュレーションオブジェクトに反映される．

4.4. createController関数の実装

コントローラを実装した場合は createController 関数を実装する．この関数は自分で実装したモデルクラスのインスタンスを生成して返り値として返す．この関数は C 言語タイプの関数としてエクスポートするため，関数の前に extern "C" をつける．

5. シミュレーションサーバの実装

この章では本システムの核をなすシミュレーションサーバの機能と処理について解説する．

5.1 起動時の処理

シミュレーションサーバは起動時に以下の処理を行う．

クライアント接続ポートのオープン
ODE の初期化
クライアントリクエスト待ち受け用スレッドの起動
世界ファイル・エージェントファイルのロード
コントローラプロセスの生成

これらの処理が正常に行われてシミュレーションの準備が整った段階で，シミュレーションサーバは実行待ち状態になる．

5.2 シミュレーションの実行・停止

シミュレーションサーバは実行待ち状態のときにビューアからシミュレーション開始のリクエストを受信するとシミュレーションを開始する．シミュレーション停止，再実行，終了の操作もビューアを通じて行う．

5.3 シミュレーションステップの実行

シミュレーションを開始すると，シミュレーションサーバはシミュレーションステップと呼ばれる一連の処理を実行し，仮想世界内の時間を最小時間単位だけ進める．シミュレーションステップでは以下の処理が行われる．

ODE による衝突判定
ODE の時間を進める
クライアントからのリクエストの受信，それに対する処理の実行
コントローラに onAction の実行リクエストを送信する

クライアントとのコネクションがクローズされていたら除外する

5.4 コントローラの起動

世界ファイルにおいてエージェントにコントローラが割り当てられた場合，シミュレーションサーバは runac コマンドを子プロセスとして起動する．runac コマンドはコントローラを動的にロードして仮想世界内の指定のエージェントにアタッチし，エージェントを制御する．runac のコマンドラインオプションは次のようになっている．

```
runac -h <hostname> -p <port> -n <agentname> -l <controller>
```

ここここでは runac コマンドの実装について解説する．

・libdl によるコントローラのロードとコントローラオブジェクトの生成

シミュレータはコントローラの実装を含んだライブラリを libdl を使用してロードする．コントローラは，シミュレータで動的にロード・実行できるよう Shared Library としてコンパイルされているものとする．

ライブラリをロード後，ライブラリ内の createController 関数を呼び出し，コントローラインスタンスを生成する．

・サーバへのアタッチ

コントローラはシミュレーションサーバのクライアント接続ポートに接続し，指定のエージェントへのアタッチリクエストを出す．アタッチに成功すると，コントローラはシミュレータからのイベント待ち状態に入る．

・サーバからのコントローラのメソッド呼び出し

シミュレーションサーバでシミュレーションが開始されると，まずすべてのコントローラに対して onInit メソッドを実行するリクエストが送信される．その後仮想世界における時間が進み始め，シミュレーションステップを進める毎にすべてのコントローラに対して onAction メソッドを実行するリクエストが送信される．

6. ビューアの実装

この章ではビューアについて解説する．

ビューアはシミュレーションサーバとは独立した単独アプリケーションである．ネットワークを介してシミュレーションサーバと通信し，シミュレーションの状況をリアルタイムで表示する．現時点での対応プラットフォームは Windows のみである．

6.1. 通信処理

6.1.1. シミュレーションサーバとの通信

シミュレーションサーバとの通信は，SimServer クラスにより行う．SimServer クラスの行う具体的な処理は以下の通りである．

6.1.1.1. シミュレーションサーバへの接続（SimServer::connect）

シミュレーションサーバへ接続する．ホスト名とポート番号は画面下部の設定パネルで指定する．設定ファイル(startup.cfg)またはビューア起動時のコマンドラインオプションであらかじめ選択候補を指定しておくこともできる．

6.1.1.2. シミュレーションサーバからの切断（SimServer::close）

シミュレーションサーバとの接続を切る．

6.1.1.3. シミュレーションサーバへのデータ送信

6.1.1.3.1. シミュレーション制御コマンド (SimServer::sendSimCtrlCommand)

シミュレーション開始（SIM_CTRL_COMMAND_START），シミュレーション停止(SIM_CTRL_COMMAND_STOP)等の制御コマンドをシミュレーションサーバに送信する．シミュレーションサーバは受け取ったコマンドに応じて，シミュレーションの開始・停止等の処理を行う．

6.1.1.3.2. ビューアのアタッチ要求（SimServer::attachView）

シミュレーションサーバに，ビューアのアタッチ要求を送信する．アタッチ要求が受理されると，その後のビューアからのリクエストがシミュレーションサーバに受け付けられるようになる．

6.1.1.3.3. 現在のシミュレーション世界の取得要求 (SimServer::sendGetAllEntitiesRequest)

シミュレーションサーバに COMM_REQUEST_GET_ALL_ENTITIES 要求を送る．シミュレーションサーバはこの要求を受け取ると，現在の最新の世界情報をビューアに送信してくる．ビューアはその応答結果を受け取り，ウィンドウ上の仮想世界の表示を更新する．

6.1.1.4. シミュレーションサーバからのデータ受信 (SimServer::processOnePacket)

サーバからの通信パケットを処理する．通信パケットの種類としては

COMM_RESULT_ATTACH_VIEW（ビューアのアタッチ要求に対する承認応答）
COMM_RESULT_GET_ALL_ENTITIES（最新の世界要求に対する応答）
COMM_LOG_MSG（サーバからのログメッセージ）

などがある.

旧バージョン(Irwas)からのコード上での改良として，サーバから受け取ったバイナリデータをデコーダに渡す前に，パケットの開始，終了の判断をするようにした点がある．通信では，あるまとまったデータのかたまり（ここではパケットと呼ぶ）を受信する必要があるが，ソケットの枠組みそのものには，そのようなパケットの「始まり」「終わり」を判断する仕組みはない．このためパケットがいつ完全に届いたのかを判断することができず，コードが複雑になっていた.

新バージョンではこの点を改良するために Socket::getPacket()メソッドを導入した．SIGVerse (Irwas)のパケットは，先頭が 0xAB，0xCD の 2 バイトから始まり，0xDC，0xBA で終わる．getPacket()メソッドは，このデータパターンを判断し，完全なパケットが到着するまで待つ．また，ソケットからのデータリードの際に，パケットの長さも考慮して読み出すことにより，データの読みすぎも決して起こさないようにした．これにより getPacket()の呼び出し元は，getPacket()からの返り値のデータにより，デコードが確実に行えることが保障される.

getPacket()の導入により,ソケットからのデータ受信処理部分が簡明なコードになり,可読性，安定性，信頼度が著しく改善された.

6.1.2. サービスプロバイダ

サービスプロバイダとは，本来シミュレーションサーバで処理すべき処理だが，諸般の理由によりシミュレーションサーバでは処理できない場合に，シミュレーションサーバに代わって処理を肩代わりするアプリケーションのことである.

ビューアもサービスプロバイダとしての機能を持っている．ビューアの提供するサービスは 2 つある．一つは、エージェント視点から見える他エージェントの列挙(DetectEntities)である．もう一つは，エージェント視点からのスクリーンショットの撮影(CaptureViewImage)である.

6.1.2.1. ビューアのサービス提供メカニズムの構成

以下にビューアにおけるサービス提供のしくみを図で示す.

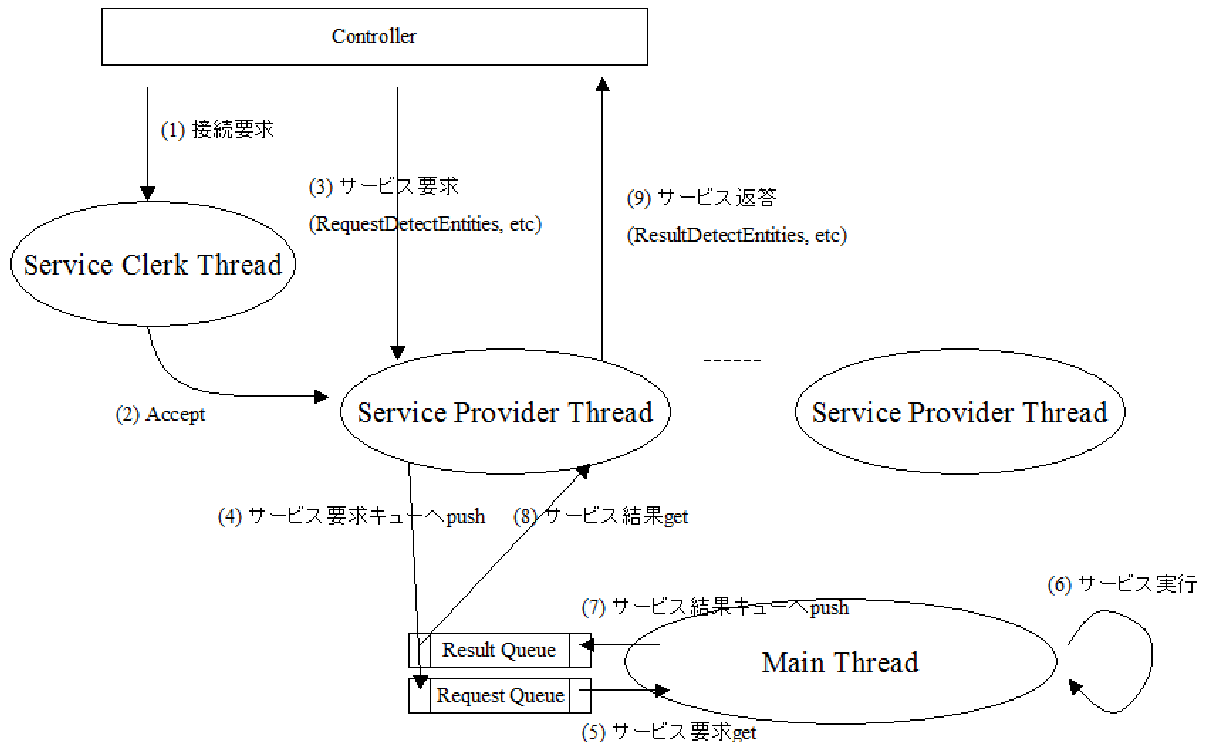

図 5 ビューアにおけるサービス提供のしくみ

(1) サービスクラークスレッドが，コントローラからサービス要求を受け取る．
(2) サービス要求を受け付けると，サービスクラークスレッドはサービスを実行するサービスプロバイダスレッドを起動する．
(3) サービスプロバイダスレッドはコントローラからサービス要求の具体的内容を受け取る．
(4) サービスプロバイダスレッドは，サービス要求をメインスレッドのリクエストキューにプッシュする．
(5) メインスレッドはサービス要求をリクエストキューから受け取る．
(6) メインスレッドがサービス要求を処理する．
(7) メインスレッドはサービスを処理した結果を，結果キューにプッシュする．
(8) サービスプロバイダスレッドが，結果キューから結果を取り出す．
(9) サービスプロバイダスレッドが，コントローラへ処理結果を返信する．

6.1.2.2. サービスクラーク (ServiceClerk)

サービスクラークとは，サービスの接続要求を受け付ける窓口である．コントローラは，シミュレーションサーバから受け取ったサービスプロバイダ情報を元に，サービスプロバイダに接続要求を送ってくる．

サービスクラークはコントローラからの接続要求を受け付ける(accept する) と，実際に処理を行うためのサービスプロバイダスレッドを開始させる．サービスクラークは，複数のコントローラからのサービス処理要求を同時に受け付けるため，接続要求が来るたびに新たなサービスプロバイダスレッドを起動する．

6.1.2.3. サービスプロバイダ (ServiceProvider)

サービスプロバイダは，スレッドとしてサービスクラークにより起動される．サービスプロバイダは，コントローラからのサービス要求を受け取り，該当する処理を行い，結果をコントローラに返信する．

本来コントローラらから受け取った処理要求は，サービスプロバイダスレッド内で処理するのが簡単であるが，ビューアの提供するサービス（DetectEntities，CaptureViewImage)を実行する Ogre（＝ビューアの採用している 3 次元グラフィックライブラリ）がスレッドセーフになっていないため，メインスレッドと異なるスレッドから直接 Ogre の機能を使うのは危険が伴う．そのため，処理要求と処理結果を蓄えるキューを用意してこれを回避している．

サービスプロバイダはコントローラからの処理要求を受け取ると，それを自分では処理せず，サービス要求キューにプッシュする．メインスレッド側は逐次サービス要求キューを監視しており，サービス要求があれば，Ogre の機能を使って処理する．得られた結果はメインスレッドにより再び結果キューに書き込まれ，サービスプロバイダスレッドがそれを取り出しコントローラに返す．このような過程を経てコントローラからの処理要求が実行される．

6.1.2.4. サービスキュー

サービスキューは 2 種類ある．1 つはサービス要求キュー，もう 1 つはサービス結果キューである．サービス要求キューへ書き込むのはサービスプロバイダスレッド，読み出すのはメインスレッドである．サービス結果キューの場合はこれが逆になる．

サービスキューに登録される内容には，要求されたサービスの種類，パラメータと，どのサービスプロバイダスレッドからの要求かを判別するためのスレッド識別子(THREAD_ID)が含まれる．サービスは複数の異なるコントローラにより同時に発生する可能性があるため，スレッド識別子により要求元を区別し，要求を出したコントローラに正しく結果を返すしくみになっている．

6.1.3. ビューアの提供するサービス

6.1.3.1. DetectEntities

DetectEntities は、指定されたエージェントの視点から見える他のエージェント、エンティティを列挙するサービスである．

DetectEntities は Ogre の PlaneBoundedVolumeQuery と RaySceneQuery クラスを併用して実現している．

PlaneBoundedVolumeQuery は，複数の平面で構成された錐体の中に存在する Ogre ノードを列挙する機能を持つ．この機能により，エージェントの視野ピラミッド内にある他のエージェントを探知することができる．

本システムではさらに RaySceneQuery を使い、ある物体の影に隠れて見えない物体の除去を行っている．RaySceneQuery は，ある点 p を始点とする半直線に交差しているすべての物体を列挙

することができる．RaySceneQuery を用いれば次のアルゴリズムで、隠れて見えない物体の除去を行うことができる.

(1) PlaneBoundedVolumeQuery により、エージェント a の視点から見えるすべての物体 e を列挙する.
(2)で列挙されたすべての e について以下を繰り返す
a から e へ向かう半直線を作り、RaySceneQuery を用いてこの半直線と交差するすべての物体 e' を列挙する.
(3) e'が a と e の間にあれば、e は e'に隠れて a から見えないため，e を探知した物体のリストから除去する.

6.1.3.2. CaptureViewImage

CaptureViewImage は、指定されたエージェントの視点から見た風景のスクリーンショットを撮り、画像のビットマップイメージを返すサービスである.

Ogre では、カメラにテクスチャイメージメモリを割り当てた状態でテクスチャのビットイメージを取り出せば、その時点でのカメラの視点からのスクリーンショットを得ることができる.

本システムではスクリーンショット撮影のための専用カメラ(MainWindow::m_pScrShotCam)を用意し、スクリーンショットを撮影している.

6.2. 形状表示

6.2.1. Ogreレンダーウィンドウの Win32 ウィンドウへの埋め込み

ビューアの根幹となる 3 次元物体形状の表示には，3 次元グラフィックライブラリである Ogre（オーガ）を使用している.

ビューアではメインウィンドウその他のすべての GUI 部品に Windows 標準のものを使い，Ogre は 3 次元形状のレンダリングエンジンとしてのみ使用している.

Win32 API により作成されたウィンドウへ Ogre のレンダリング結果を表示させるには以下のようにする.

```
// ------------------------------------
//     Ogre の描画対象を，すでに作成された
//     win32 のウィンドウにする
// ------------------------------------
NameValuePairList options;

options["externalWindowHandle"]
      = StringConverter::toString((size_t)hwndOgre);
```

```
// -----------------------------------
//     レンダーウィンドウ作成
// -----------------------------------
m_pRenderWindow = m_root.createRenderWindow(
       "embedded",
       800,
       600,
       false,
       &options);
```

上記のコードでは，"externalWindowHandle"というオプション文字列に，Win32 のウィンドウハンドル(hwndOgre)を代入することで，描画対象（ウィンドウ）と描画者（Ogre)を結び付けている．このオプション指定を Ogre のウィンドウ作成関数(createRenderWindow)に渡すと，Ogre でのレンダリング結果が Win32 の通常のウィンドウに表示されるようになる．

6.2.2. X3D形状データのロードと表示

ビューアでは X3D 形式で表現された物体形状を表示できる．モデリングツールにより作成したリアルな物体形状を X3D ファイル形式に変換しておけば，ビューアはその形状データを読み取って表示することができる．

6.2.2.1. X3D形式ファイルのパース

ビューアは形状表示に Ogre を使用しているが，Ogre は X3D 形式のファイルを直接読み込む機能を持っていない．そこで X3D 形式ファイルを読み込むパーサと，パーサから得た情報を Ogre システムに適合させ表示するしくみが必要になる．

X3D 形式のファイルのロードには，専用に開発した X3D 形状パーサー(CX3DParser クラス）を使用している．CX3DParser クラスを使えば，X3D 形式ファイルのパースは以下のように簡単に行える．

```
// ----------------------------------------------
//  X3D ファイルをパースする
// ----------------------------------------------
CX3DParser parser;

if (!parser.parse("shape.x3d"))
{
       // ロードエラー発生
       exit(1);
}

// ロード完了
```

X3D のパースが成功すれば，CX3DParser クラスのメンバ関数を通じて，物体の詳細情報を得ることができる．

6.2.2.2. Ogreマニュアルオブジェクトの作成

CX3DParser クラスにより取り出せる X3D の形状情報は，物体の頂点の位置，面の構成情報などの数値データである．これを実際の画面で目に見える形として表示するためには，これらの情報から Ogre マニュアルオブジェクトを作る必要がある．

Ogre マニュアルオブジェクトとは，プログラム中で動的に生成した形状を表示するしくみである．このしくみを利用すれば、X3D のような、本来 Ogre が対応していない形状でも、Ogre で表示することが可能である．

本システムでは、X3D クラスの createOgreManualObjectNodeFromShapes メソッドにより、Ogre マニュアルオブジェクトを生成している．

```
// -----------------------------------------
//  X3D 形式から Ogre マニュアルオブジェクトを作成
// -----------------------------------------
Ogre::SceneNode *shapeNode
= x3d.createOgreManualObjectNodeFromShapes(mgr, node, visObjElemName);
```

6.2.3. SimObjの属性値による表示形状の切り替え

6.2.3.1. 状態値

本システムのような，実世界を模写したシミュレーションシステムにおいては，場合によって物体形状の外見をリアルタイムに変更したいという要望が生じることがある．

例えばシミュレーション世界内にテレビがあった場合，エージェントがスイッチを入れればテレビ画面には何かの画像が表示され，スイッチを切れば画面は消えるだろう．また，部屋の中にライトがある場合，スイッチを入れれば点灯し，切れば消灯するだろう．このように，シミュレーションの進行に応じて動的に表示物体が「変化」することが必要になる．

本システムでは，この「状態変化に応じた表示形状の変化」に対応している．シミュレーション世界上のエンティティは，表示変更に使う属性名をあらかじめ宣言しておく．この宣言は，属性名と，その属性値に対応して使う形状ファイルの名前を列挙することで行う．表示変化に使うと宣言された属性の属性値を変化させると，あらかじめ定義された属性値に対応した形状ファイルに切り替える．

状態値として使う属性名の宣言は，visStateAttrName 属性で行う．visStateAttrName は Entity.xml で以下のように宣言されている．

```
<attr name="visStateAttrName" type="string" group="visState"
value="visual"/>
```

例えば，TV.xml エンティティで，switch という属性を表示状態属性として使いたい場合は，まず switch を以下のように宣言しておく．

[TV.xml]

```
<define-class name="TV" inherit="Agent.xml">
      <attr name="switch" type="string" group="visState"/>
</define-class>
```

そして，世界ファイル内で，switch 属性を表示状態属性として使うことを宣言する.

[MyWorld.xml]

```
<world name="MyWorld">
      <instanciate class="TV.xml">
            <set-attr-value name="name" value="TV_0"/>
            <set-attr-value name="dynamics" value="false"/>

            <set-attr-value name="x" value="-20.0"/>
            <set-attr-value name="y" value="82.0"/>
            <set-attr-value name="z" value="-250.0"/>

            <set-attr-value name="visStateAttrName" value="switch"/>
            <set-attr-value name="switch" value="off"/>
      </instanciate>
</world>
```

宣言した表示状態属性はコントローラのプログラム中で動的に変更することが可能である. コントローラ中での状態変更は即座に画面に反映される.

```
double TVController::onAction(ActionEvent &evt)
{
      SimObj *o = getObj("TV_0");
      o->setAttrValue("switch", "on");
}
```

6.2.3.2. 状態値と形状ファイルの対応

表示状態属性として宣言された属性の値に対して，どの形状ファイルを使用するかの指定は，XML 中で行う．例えば以下の例では，Apple クラスの表示に使う形状として apple.x3d を指定している.

```
<define-class name="Apple" inherit="Entity.xml">
      <x3d>
            <filename>apple.x3d</filename>
      </x3d>
</define-class>
```

状態値と形状ファイルの対応の指定は、この書式に状態指定の記述を付け加えることで行う.

例えば、TV クラスの表示として、switch 属性の値が"on"の場合に"TV_on.x3d"を、"off"の場合に"TV_off.x3d"を使用する場合、以下のように記述する.

```
<define-class name="TV" inherit="Entity.xml">
	<x3d>
		<filename>TV_on.x3d</filename>
		<state name="switch" value="on"/>
	</x3d>
	<x3d>
		<filename>TV_off.x3d</filename>
		<state name="switch" value="off"/>
	</x3d>
</define-class>
```

シミュレーションサーバは、世界ファイルを読み込んだ際に、世界内に存在するクラスと、その表示時に使われる形状ファイルとの対応データベースを作成する．このデータベースの内容は，ビューアがシミュレーションサーバに送るアタッチ要求への返答に含めてビューアに通知される.

6.2.3.3. 状態値の変化に応じて表示形状を変えるしくみ

ビューアはサーバから転送された世界(SimWorld)を最初に表示する際に，SimWorld に含まれるすべての SimObj から Ogre 形状ノードを作り出す. SimObj のクラス名と、それに対応する X3D ファイル名のリストは，シミュレーションサーバからビューアへのアタッチ承認返答に含まれている．クラスに表示状態属性が定義され、複数の X3D 形状を切り替える可能性がある場合、ビューアはそれら複数の形状ファイルそれぞれから Ogre シーンノードを作成し、属性値と Ogre シーンノードのマップを作成する.

SimObj の属性値が変更されると，その属性値に対応する Ogre シーンノードを表示状態にし，それ以外の属性値に対応する Ogre シーンノードは非表示状態にする．これにより指定された属性値に対応する X3D 形状のみが表示されるようになり，属性値を変化させることであたかも動的に物体の見た目が変化したように見える.

6.2.4. OpenHRPヒューマノイド形状データのロードと表示

本システムでは OpenHRP 仕様に基づくヒューマノイドデータでの物理シミュレーションの実行および表示をサポートしている．OpenHRP の形状データは標準の X3D 形式に、独自の PROTO ノードの定義を追加したものとなっている．以下に OpenHRP 仕様での拡張ノードについて簡単に述べる.

6.2.4.1. OpenHRPの形状データの概要

Humanoidノード

OpenHRP 形状のルートとなるノード．Joint，Segment ノードは、Humanoid ノードの子ノードとなる.

Jointノード

関節を表すノード．基点からの変位(translation)と，回転(rotation)を持つ．Joint ノードは，子として 0 個以上の Joint ノードと，高々1 個の Segment ノードを持つ．

Segmentノード

関節に繋がる体の部位を表す．(例えばジョイントが肩の関節の場合，セグメントに相当するのは上腕部分，ジョイントが膝の関節の場合はセグメントに相当するのは脛，という具合)
Segment ノードは，物体の形状のみ持つ．親からの位置の変位(translation)や回転(rotation)は持たない．

Segment ノードには通常，モデリングツールで作成した複雑な物体形状データが置かれる．

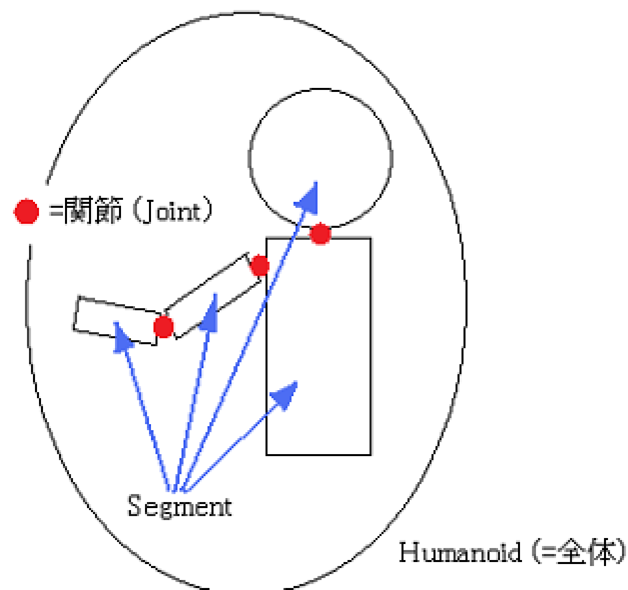

図 6 OpenHRP のノード

6.2.4.2. OpenHRP形状データの特徴

ジョイントにおける回転の指定は，3 次元の回転軸ベクトルと，その軸周りの回転角を与える形式になっている．つまり，ある 1 つのジョイントでは，任意の回転を表現することはできず，ある一つの軸周りの回転しか表現できない．一方、ジョイントは自分自身の子として，ジョイントを持つことができる．そこで OpenHRP においては、複雑な回転は、複数のジョイントを連続して設置して表現する．

6.2.4.3. OpenHRP形状データの表示

OpenHRP 形式のヒューマノイドは、腕、手、足など個別に動く複数の部位から成る．シミュレーションサーバは、これらの部位の位置、回転角等を計算する．ビューアはそれら各部位の情報を絶対座標系で受け取って表示する．

シミュレーションサーバから送られてくるデータは以下のような形式をしている.

名前	translation	rotation
"HEAD"	(0.0 0.0 0.0)	(1.0 1.0 1.0 0.52359)
"WAIST_LINK0"	(0.0 10.0 0.0)	(0.0 1.0 0.0 -1.04720)
"LARM_LINK0"	(-5.0 3.0 0.0)	(1.0 0.0 0.0 2.61799)
...	...	...

ここで"HEAD", "LARM_LINK0"などの名前は，各パーツの名前を表し，translation, rotation はそのパーツの絶対座標と回転を表している.

6.2.5. VisObj

VisObj はビューア上での「1 つの物体形状」を表現するクラスである．ビューアで扱う形状のうち，静止物体（単なる X3D ファイル）とヒューマノイド（OpenHRP 仕様に基づくヒューマノイド形状を表現する X3D ファイル）とでは若干性質が異なるため，それぞれ VisObjStatic と VisObjHumanoid という派生クラスで表現している.

VisObjStatic は静止物体（机，テレビ，ランプ，りんご等）に対応する．また，表示属性の変更に対応するため，属性値とそれに対する形状データ(Ogre シーンノード)のマップを持っている.

VisObjHumanoid は OpenHRP 形式のヒューマノイドを表現するクラスである．VisObjHumanoid はパーツ名と，パーツ名に対応する Ogre シーンノードのマップを持っている．シミュレーションサーバから送られてくるデータには，パーツ名ごとにそのパーツの位置(translation)，方向(orientation)が入っている．これはヒューマノイドの各部位（腕，頭，足など）ごとに位置や向きが変わるためである．なお、VisObjHumanoid は表示属性による表示形状の変更には対応していない.

6.2.6. VisWorld

VisWorld はビューアでの表示世界全体を表すクラスである．世界中に存在するすべての VisObj は VisWorld が保持している.

VisWorld はシミュレーションサーバから受け取ったシミュレーションオブジェクト(SimObj)から VisObj を生成するファクトリーを備えている．このファクトリーは，後述するビデオファイルの再生時に使われる.

6.3. 録画・再生

ビューアは，シミュレーションサーバから逐次受信，表示しているシミュレーション状況を，ビデオテープのようにファイルに録画し，再生する機能を持っている．録画データは，拡張子が.svd という独自フォーマットで記録される．記録した.svd ファイルをビューアで読み込み，再生すれば，記録した状況が画面上で再現できる．また，再生の一時停止，巻き戻し，ステップ再生等も行うことができる．この様子を概念的に示した図を以下に示す.

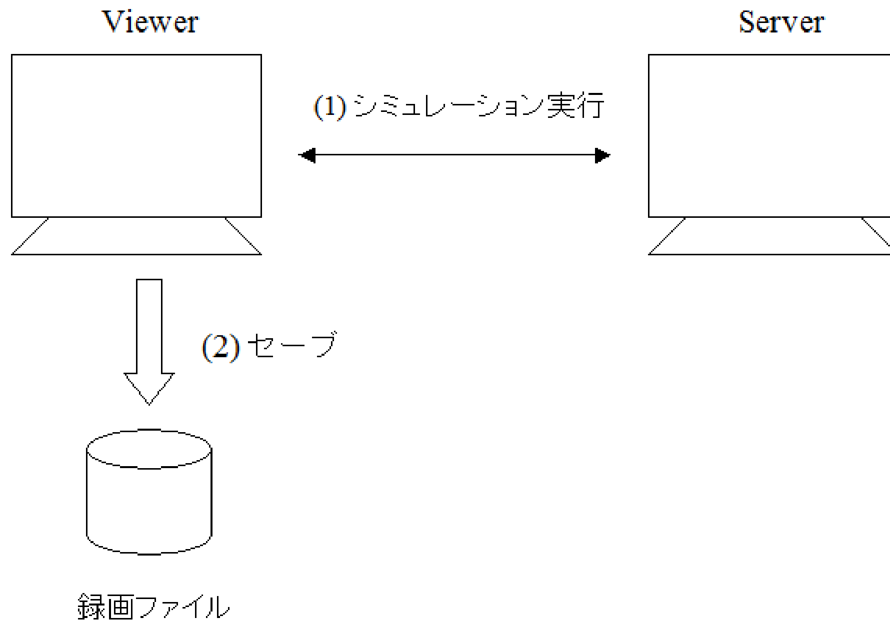

図 7　録画時の処理

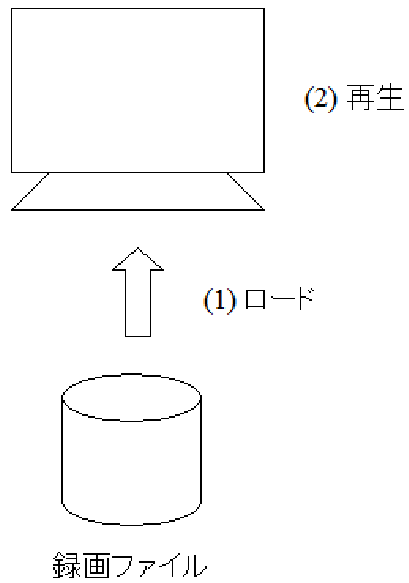

図 8　再生時の処理

なお，再生処理は完全にビューア側のみで行われるため，再生時にシミュレーションサーバに接続する必要はない．

6.3.1. 録画・再生処理の実装

6.3.1.1. シリアライズ

シミュレーションサーバから送られてくる世界データは，SimWorld と呼ばれるオブジェクトに格納されている．SimWorld はある瞬間での，すべてのエンティティ，エージェントの情報を保持している．

録画処理では SimWorld が含む情報をシリアライズ（直列化）して，単一のビットイメージに変換する．シリアライズを行った結果は，ある特定の大きさのメモリブロックになる．これをバイナリファイルとして逐次的にファイルに書き込んでいけば，シミュレーションの進行状況をすべてファイルに記録することができる．

以下にシミュレーションの進行状況をファイルに記録する概念的コードを示す.

```
// シリアライザー
Sgv::Serializer se;

for (;;)
{
	// シミュレーションサーバから新しい世界を得る
	SimWorld *w = getNewWorldFromSimServer();

	// シリアライズを行い，バイナリデータ化する
	se.serializeWorld(time, w);

	// 記録終了
	if (quitRecording()) break;
}

// ファイルに書き出す
se.writeToFile("sample.svd");
```

6.3.1.2. デシリアライズ

.svd ファイルに記録されたシミュレーション状況を再生するには，記録されたデータを読み込み，ある時点での世界(SimWorld)の状況を復活させる必要がある．この処理をデシリアライズと呼ぶ.

デシリアライズは Serializer クラスの deserializeWorld メソッドにより行う.

以下に概念的コードを示す

```
Sgv::Serializer se;

// 録画ファイルを読み込む
```

```
se.readFromFile("sample.svd");

// デシリアライズを行い，バイナリデータから世界を復活させる
SimWorld *w = se.deserializeWorld(0, time);
```

上記のような処理を行うことで、ビューアは.svd ファイルに記録されたビットイメージから世界イメージを復活させる．復活させた世界を連続して表示させることで，あたかもシミュレーションが進行中であるかのような状況を作り出している．

6.3.1.3. データの圧縮

シミュレーションの録画・再生を、シリアライズ・デシリアライズにより単純に行った場合，記録されるファイルのサイズが録画時間に比較して大きめになる．そこでシリアライズされたデータの書き込み前にデータを zlib ライブラリにより圧縮し，ファイルサイズを圧縮する処理を行っている．

この圧縮処理を加えた結果，.svd ファイルのサイズは圧縮処理なしの場合と比較して 1/50 程度まで小さくすることができた．

6.4. 音声認識インターフェース

6.4.1. OpenALによる音声取り込み

マイクからの音声の取り込みには OpenAL ライブラリを使用している．OpenAL は Creative 社製のサウンド関連のライブラリである．

OpenAL の API を使用し，マイクから入力された音声データを.wav 形式で記録する．

6.4.2. 音声データの送信

OpenAL により取り込まれた音声データ(.wav)は，その音声データを送る対象のエージェント名と共に，Raw データ（バイナリデータ）としてシミュレーションサーバに転送される．

以下に転送部分のコードの概略を示す．

```
bool SimServer::sendRawSound(RawSound *sound, const char *sendTo)
{
	// send sound to agent
	CommInvokeMethodOnRecvSoundEncoder enc(
		1.0,
		"SIGViewer",
		sendTo,
		*sound);

	if (enc.send(m_pSock->sock()) < 0) return false; // error
```

```
        return true;        // succeeded
}
```

音声データ(sound)はInvokeOnRecvSoundEncoderクラスによりバイナリデータ化され，シミュレーションサーバへ送られる．シミュレーションサーバは受け取った音声データを、送信先のコントローラへ転送する．コントローラはonRecvSoundイベントにより、この音声データを受け取ることができる．

7. 音声認識サービスプロバイダの実装

本システムは音声認識サービスとして、京都大学で開発された音声認識アプリケーションjuliusを使用した音声認識サービスプロバイダsigjspを用意している．ここではsigjspの動作のしくみについて説明する．

7.1. 音声認識サービスプロバイダの起動

sigjsp は独立したアプリケーションである．音声認識サービスを使う場合、シミュレーションサーバを起動した後、sigjsp を起動しておく．sigjsp は起動時に、自分が音声認識サービスを提供するサービスプロバイダであることをシミュレーションサーバに通知する．

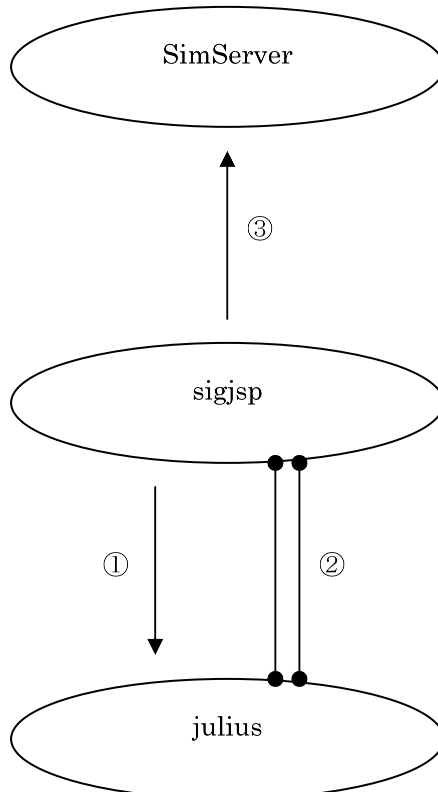

図 9. 音声認識サービスプロバイダの起動時の処理

sigjsp の起動時の処理の流れは以下の通りである.

（1）julius をモジュールモードで起動する.
（2）sigjsp は julius との通信用ポート（モジュールポート、adinnet ポートの 2 本）を開く
（3）sigjsp は自分が音声認識サービスを受け付けることを、シミュレーションサーバに通知する．その後コントローラからの認識要求に対する待機状態に入る.

7.2．音声認識サービスプロバイダの利用

音声認識サービスプロバイダにより、コントローラは音声データから、音声認識処理を行うことができる．ここではコントローラが音声認識依頼を出し、音声認識結果を得るまでの処理の流れを示す.

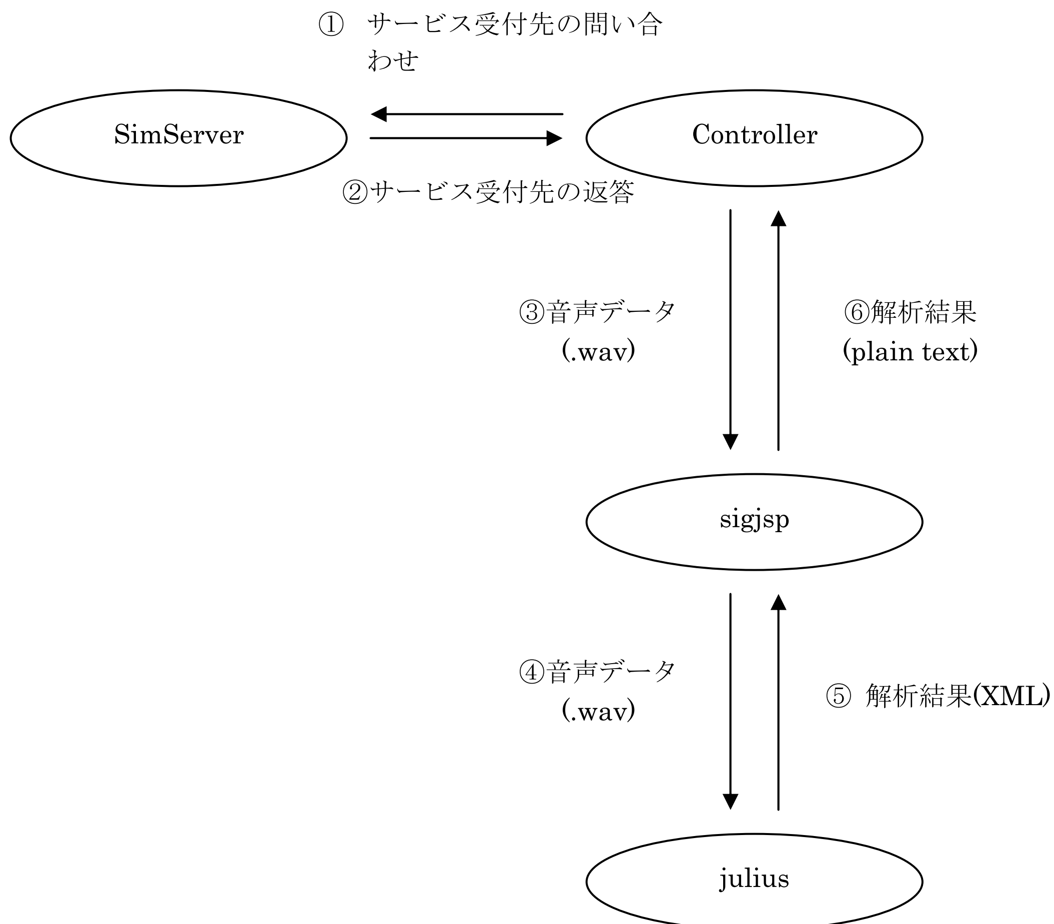

図 10. 音声認識サービスプロバイダの認識要求時の処理

（1）コントローラは音声認識サービスを提供するプロバイダ情報をシミュレーションサーバに問い合わせる.
（2）シミュレーションサーバは音声認識サービスを提供するプロバイダ情報をコントローラに返信する.
（3）コントローラはプロバイダ情報に従って、sigjsp へ音声認識要求を送信する.

（4）音声認識要求を受け取った sigjsp は、julius へ音声データを送信する.
（5）julius は受け取った音声データを元に認識を実行し、結果（XML 形式）を sigjsp に返信する.
（6）sigjsp は XML から文字列を抽出し、プレーンテキストに変換したものをコントローラへ返信する.

8. データ通信

この章ではシミュレーションサーバとクライアント間のデータ通信について解説する．シミュレーションサーバまたはクライアントは，転送するデータまたはリクエストをバイナリ化し，それにヘッダ・フッタを付与したパケットと呼ばれるデータ形式で転送される．パケット内のバイナリから元のデータが復元できるようになっている．また，転送データのサイズが大きい時には，そのデータを複数に分割して転送することも可能である．

8.1. パケット構成

パケットはヘッダ，データ領域，フッタにより構成される．バケットタイプ（転送するデータのタイプ）に応じたデータがデータ領域に格納される．

カテゴリ	項目名	データタイプ	値
ヘッダ	パケット開始トークン	符号なし 2 バイト	0xabcd
	パケットサイズ	符号なし 2 バイト	
	パケットタイプ	符号なし 2 バイト	
	パケット数	符号なし 2 バイト	
	パケットシーケンス番号	符号なし 2 バイト	
	転送タイプ	符号なし 2 バイト	
	転送先	文字列	
	転送範囲	実数	
データ	データ領域	任意	
フッタ	パケット終了トークン	符号なし 2 バイト	0xdcba

データ転送時，充分小さいデータを転送する場合は 1 つのパケット内に収まる．この場合パケット数は 1 に設定される．画像データなどデータが大きい場合は複数のパケットに分割して送信される．この場合バケット数に 1 より大きい値が設定され，分割パケットの順にパケットシーケンス番号が 0 から振られる．

8.2. パケットタイプ

ここではパケットの種類とパケットタイプ毎に転送されるデータについて解説する．

8.2.1. リクエストとリザルト

シミュレーションサーバとクライアント間には必要に応じて，クライアントからサーバ，サーバからクライアントにリクエストパケットが転送され，これにより連携動作している．リクエストパ

ケットを受け取った側はそれに応じた処理を行い，処理の結果があればリザルトパケットとして転送元に送り返す.

8.2.2. コントローラメソッドの呼び出し

エージェントの動きを制御するコントローラに対して規定のパケットを送信すると，ネットワーク越しにコントローラの onInit, onAction などのメソッドを呼び出すことができる.

8.2.3. パケットタイプ一覧

クライアントのアタッチリクエスト

リクエスト	リクエストデータタイプ リザルトデータタイプ	転送データ
ビューのアタッチ	COMM_REQUEST_ATTACH_VIEW	ビューの名前
	COMM_RESULT_ATTACH_VIEW	結果 エラーメッセージ
コントローラのコマンドポ ートのアタッチ	COMM_REQUEST_ATTACH_CONTROLLER	エージェント名
	COMM_RESULT_ATTACH_CONTROLLER	結果 エラーメッセージ
コントローラのデータポートのアタッチ	COMM_REQUEST_CONNECT_DATA_PORT	エージェント名
	COMM_RESULT_CONNECT_DATA_PORT	結果 エラーメッセージ
サービスプロバイダのアタッチ	COMM_REQUEST_PROVIDE_SERVICE	サービスプロバイダの名前
	COMM_RESULT_PROVIDE_SERVICE	結果 エラーメッセージ

データの取得・変更

リクエスト	リクエストデータタイプ リザルトデータタイプ	転送データ
シミュレーション内のオブジェクト名を取得	COMM_REQUEST_GET_OBJECT_NAMES	取得オブジェクトタイプ
	COMM_RESULT_GET_OBJECT_NAMES	オブジェクト名
指定のオブジェクトを取得	COMM_REQUEST_GET_ENTITY	シミュレーションオブジェクト名
	COMM_RESULT_GET_ENTITY	シミュレーション時間 シミュレーションオブジェクト
オブジェクトをすべて取得	COMM_REQUEST_GET_ALL_ENTITIES	なし
	COMM_RESULT_GET_ALL_ENTITIES	シミュレーション時間 シミュレーションオブジェクト(複数)
オブジェクトの状態を変更	COMM_REQUEST_UPDATE_ENTITIES	シミュレーション時間 シミュレーションオブジェクト(複数)
	なし	–

画像の取得	COMM_REQUEST_CAPTURE_VIEW_IMAGE	画像データタイプ, １画素あたりのビット数 画像サイズ
	COMM_RESULT_CAPTURE_VIEW_IMAGE	画像データタイプ １画素あたりのビット数 画像サイズ 画像データ
シミュレーションオブジェクトの視認	COMM_REQUEST_DETECT_ENTITIES	なし
	COMM_RESULT_DETECT_ENTITIES	検出したエージェント名（複数）

ヒューマノイド物理シミュレーション

リクエスト	リクエストデータタイプ リザルトデータタイプ	転送データ
ジョイントの接続	COMM_REQUEST_CONNECT_JOINT	ジョイント名 エージェント名 エージェントパーツ名 ターゲット名 ターゲットパーツ名
	COMM_REQUEST_RELEASE_JOINT	エージェント名 ジョイント名
ジョイントトルク計算	COMM_REQUEST_GET_JOINT_FORCE	エージェント名 ジョイント名
	COMM_RESULT_GET_JOINT_FORCE	結果 エラーメッセージ

コントローラメソッドの呼び出し

メソッド名	データタイプ	転送データ
onInit	COMM_INVOKE_CONTROLLER_ON_INIT	なし
onAction	COMM_INVOKE_CONTROLLER_ON_ACTION	シミュレーション時間
onRecvText	COMM_INVOKE_CONTROLLER_ON_RECV_TEXT	シミュレーション時間 発信者名 送信先 テキスト エンコード 到達範囲
onRecvSound	COMM_INVOKE_CONTROLLER_ON_RECV_SOUND	シミュレーション時間 発信者名 送信先 音声データ
onRecvMessage	COMM_INVOKE_CONTROLLER_ON_RECV_MESSAGE	発信者名 送信先 メッセージ（複数行）

ネームサービス

リクエスト	リクエストデータタイプ リザルトデータタイプ	転送データ
サービス提	COMM_NS_QUERY_REQUEST	サービス識別子

供先の情報	COMM_NS_QUERY_RESULT	結果 エラーメッセージ
存在確認	COMM_NS_PINGER_REQUEST	サービス識別子
	COMM_NS_PINGER_RESULT	有効（アクティブ）フラグ

音声認識

リクエスト	リクエストデータタイプ	転送データ
	リザルトデータタイプ	
音声認識	COMM_REQUEST_SOUND_RECOG	音声データ
	COMM_RESULT_SOUND_RECOG	認識結果テキストデータ

X3DDB

リクエスト	リクエストデータタイプ	転送データ
	リザルトデータタイプ	
X3DDB の転送	COMM_REQUEST_X3DDB	なし
	COMM_RESULT_X3DDB	X3DDB

その他

リクエスト	データタイプ	転送データ
ログの転送	COMM_LOG_MSG	ログレベル テキスト

8.3. パケットの転送

本システムではコントローラ間でソケットコネクションを張り，直接データ通信を行う機構を備えていない．そのため，コントローラが他のコントローラに対してデータの送信を行う場合，シミュレーションサーバを介したデータ転送が行われる．

転送が必要な場合，送信元コントローラはリクエストに転送情報を付けてサーバに送信する．シミュレーションサーバは転送情報に基いて適切なコントローラを選び，パケットに変更を加えずそのまま転送する．転送したパケットがリクエストまたはコントローラ呼び出しで返り値を必要とする場合，シミュレーションサーバは転送先からのデータを待ち，受信したデータをそのまま送信元に転送する．

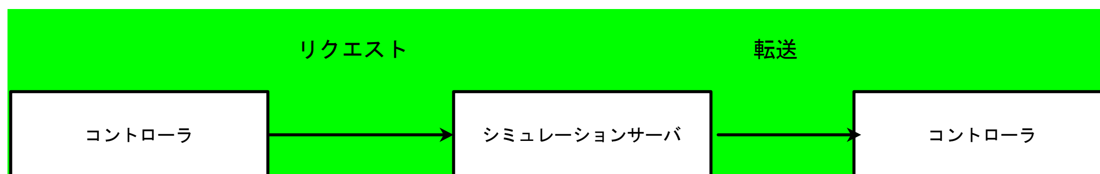

図 11 パケットの転送（リザルトなし）

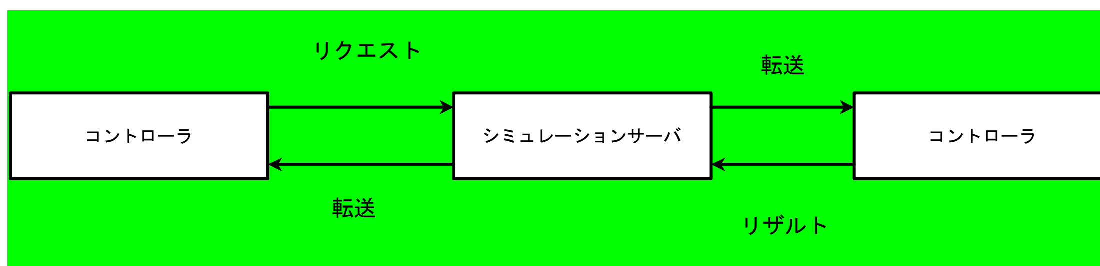

図 12 パケットの転送（リザルトあり）