情報システム技術論'08 — ＃2

久野　靖*

2008.4.17

1　通信の構造化

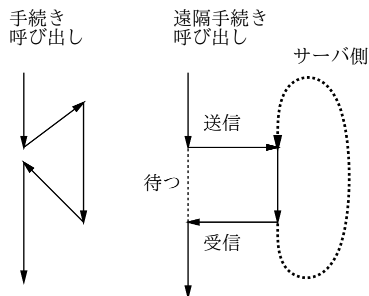
1.1　遠隔手続き呼び出し (RPC)

□ ネットワーク経由の通信は手順が面倒

- 前回やったように「ソケットを用意→接続を張る→送受信 (N 回) →切断」
- もっとプログラマに書きやすくできないか?
- プログラマにとってより「手慣れた」インタフェース (抽象化) は?

□ もちろん、手続き呼び出し (サブルーチンコール) は非常によく使うし、手慣れている。

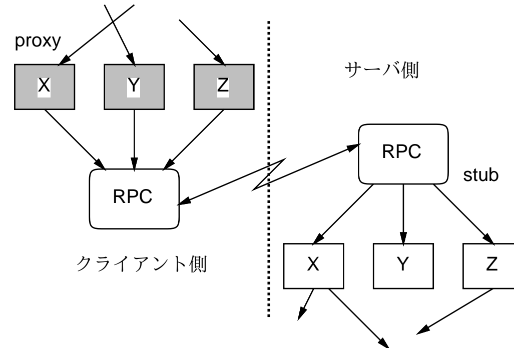
- ネットワーク越しの通信も手続き呼び出しと同様に考えることができる。「パラメタを渡して呼ぶ→処理を待つ→結果を受け取る」



- クライアント側:　パラメタ送信→待つ→結果受信
- サーバ側:　「パラメタ受信→処理→結果送信」の無限反復

□ 実際にはさらに次のような仕掛けが入ることが多い

- サーバ側に複数の手続きがあり、それを区別して呼び出せる (どれを呼ぶかを名前で指定して呼び出す)
- クライアント側/サーバ側に「呼ばれる手続き (プロキシ)」「呼ぶ手続き (スタブ)」が用意されていて、他の部分からは普通の手続きに見える。



- 「本物」の手続きはサーバの中にありこれがネット越しに呼ばれる

□ RPC ジェネレータ

- 上記のスタブ類を用意するのはすごく面倒に思えるが…
- 実際には「名前」「パラメタ個数とそれぞれの型」「返値の型」を指定してやれば、自動的に両側のスタブを生成してくれるジェネレータ (RPC ジェネレータ、スタブジェネレータ) が用意されている。
- パラメタや返値が複雑な型のときも、それを詰め込み (marshal)、取り出し (unmarshal) を行うコードが用意される。

□ RPC のバインディング

- このように RPC では一見手続き呼び出しのようにサーバ上のコードが呼び出せるが、実際には「ネットワークの向こう」への通信になっている。
- このため、クライアント側では「どのホスト宛接続する」という情報を供給する必要がある。
- サーバ側では RPC 呼び出し全体を管轄し、指定されたプログラムのスタブへの橋渡しを行うための「RPC サーバ」プログラムを動かしている必要がある。
- FreeBSD、Solaris などでは ONC RPC と呼ばれる RPC システムが動いていて、「rpcinfo」でサーバ側のプログラム登録状況が見られる。

*筑波大学大学院経営システム科学専攻

1.2 RPCのおもちゃを作る

□ では、前回の例題をちょっと直して、RPCの「おもちゃ」を作ってみる。サーバ内にdataという文字列変数が存在していて、その内容を読み出すget()とそこに書き込むput()というメソッドを遠隔呼び出しする。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.net.*;

public class Sample21 extends JFrame {
  JTextField f1 = new JTextField();
  JButton b1 = new JButton("Put");
  JButton b2 = new JButton("Get");
  JLabel l1 = new JLabel("start...");
  String host;
  public Sample21(String h) {
    host = h;
    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container c = getContentPane(); c.setLayout(null);
    c.add(f1); f1.setBounds(40, 40, 180, 30);
    c.add(b1); b1.setBounds(40, 80, 80, 30);
    c.add(b2); b2.setBounds(130, 80, 80, 30);
    c.add(l1); l1.setBounds(20, 360, 360, 40);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        put(f1.getText());
      }
    });
    b2.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        f1.setText(get());
      }
    });
  }
  private void put(String str) { rpc("p"+str); }
  private String get() { return rpc("g"); }
  private String rpc(String str) {
    try {
      Socket cs = new Socket("localhost", 4192);
      PrintWriter out = new PrintWriter(
        cs.getOutputStream(), true);
      BufferedReader in = new BufferedReader(
         new InputStreamReader(cs.getInputStream()));
      out.println(str); str = in.readLine(); cs.close();
    } catch(Exception ex) { l1.setText("!" + ex); }
    return str;
  }
  public static void main(String[] args) {
    new Sample21(args[0]).setVisible(true);
  }
}
```

- 今回からは「java Sample21 localhost」のように実行時にサーバのホスト名を指定するようにした(実験がやりやすいように)。
- 窓を作るあたりは同じ。
- ボタンは2つにし、それぞれのボタンはget()とput()でフィールドにサーバから値を読み出したりフィールドの値をサーバに書き込んだりする。
- get()とput()のスタブはrpc()という通信用メソッドを呼び出す。そのとき「何のメソッドか」を表す情報をくっつける。
- rpc()は前回やったようにしてサーバと通信する(文字列を送信し、文字列を受信する)。

□ 次はサーバ側。こちらも前回とほぼ同様。

```
import java.net.*;
import java.io.*;

public class Sample21Server {
  static String data = "";
  public static void main(String[] args)
      throws Exception {
    ServerSocket ss = new ServerSocket(4192);
    while(true) {
      Socket cs = ss.accept();
      PrintWriter out = new PrintWriter(
        cs.getOutputStream(), true);
      BufferedReader in = new BufferedReader(
        new InputStreamReader(cs.getInputStream()));
      String line = in.readLine();
      switch(line.charAt(0)) {
case 'p': put(line.substring(1)); line = ""; break;
case 'g': line = get(); break;
default:
      }
      out.println(line); cs.close();
      if(line.equals("bye")) break;
    }
    ss.close();
  }
  static void put(String s) { data = s; }
  static String get() { return data; }
}
```

- 渡されて来た文字列の先頭を見て「どの手続き呼び出しか」を区分し、各手続きを呼び出す。
- get()とput()は先に述べたようにdataを読み書きする。

□ 演習: このプログラムを動かしてみよ。サーバを1つにして、複数の人が互いに文字列を共有できることを確認せよ。

□ 演習: このプログラム(サーバ1つ)を使って、「各自が思い描いた数値を合計する」のをやってみよ。何が問題か検討せよ。

□ 演習: 上記の問題を解決するような遠隔手続きとはどんなものか考えよ。実際にそれを組み込め。

1.3 ORBとCORBA

□ オブジェクト指向とORB

- ここまでは「手続き指向」だったが、オブジェクト指向の普及とともに、呼び出される対象は「オブジェクトのメソッド」と考える方が素直に思えるようになってきた。
- サーバ側には「複数のオブジェクト」があり、それぞれが固有のデータを持っている。そのオブジェクトのメソッドを RPC で呼び出すことで、その固有データに間接的にアクセスできる。
- その遠隔呼び出しの橋渡し (バインディング) → ORB(Object Request Broker) と呼ばれるようになった。

□ CORBA --- ORB の標準仕様。1990 頃からある。

- ORB としては色々なものが作られたが、相互運用性のために共通のものを作ろうという話になり、OMG(Object Management Group) が作った標準仕様が CORBA(Common Object Request Broker Architecture)
- CORBA 仕様に従っているなら、どのクライアントとどの ORB を組み合わせても呼び出し利用できる (はず)
- ただし Microsoft はこれにくみせず COM/DCOM を採用 (Windows 用の CORBA ソフトももちろん作られたが)。
- CORBA IDL(Interface Definition Language) --- どんなメソッドがどんなパラメタを持っているか、を指定する標準的な記法として使われるようになった。(スタブジェネレータに食べさせるのが本来の目的だったがどちらかというと人間が読む仕様記述の記法として…)

□ CORBA の現在…

- 汎用的すぎて面倒であり遅かった。そしてあまり普及しないうちに Web アプリケーションの時代になってしまった。
- Web アプリでは遠隔呼び出しに CORBA よりは XML-RPC や SOAP(Simple Object Access Protocol) など XML 系の技術を使う。Web アプリ側が接続先を指定できるので ORB はなくても済んでしまう。
- OMG は今は UML などの方に力を入れている (UML はメジャーになったのでそれなりに成功している)。
- それはそれとして、現在でも Java などから一応 CORBA は使えるようになっている。

2 Java RMI

2.1 Java RMI の概要

□ RMI(Remote Method Invocation) --- 遠隔オブジェクトに対してメソッド呼び出しを行うような機能全般。

□ Java では比較的初期からそのための機能が組み込まれている→ Java RMI。

- ORB、スタブジェネレータなども一通り揃っている。
- Java 言語の枠内で揃うようになっているので、CORBA より簡単。
- データだけでなく、コードも転送可能→大きな特徴。

□ 以下で先の例題をもとにした「文字列記憶サービス」を用いて RMI の必要事項を説明していく。

2.2 例題:文字列サービス

□ [1] インタフェース仕様

- Java RMI では CORBA IDL の代わりに Java 言語に備わっているインタフェース定義で遠隔オブジェクトの仕様を規定する。
- このインタフェース仕様は必ず「extends Remote」を指定する。これにより、このインタフェースは遠隔オブジェクトの仕様として使われることが宣言される。
- このインタフェースに現れるメソッドはすべて RemoteException という例外を発生させ得るように指定する。
- 実際、ネット経由で呼び出すのでネットワークが切断したらエラーになるから、そのような場合を必ず考慮するべき。(今回は例題を短くするためにさぼっている。)

```
import java.rmi.*;

interface Sample13IF extends Remote {
  public String exec(String s) throws RemoteException;
}
```

□ 解説:

- Remote インタフェースである。
- メソッドは文字列を 1 つ受け取り 1 つ返す exec() のみ。
- このメソッドは RemoteException を発生させ得る。

□ [2] サーバ側コード

- サーバ側に置かれたオブジェクトは extends UnicastRemoteObject と指定されている必要がある。
- サーバ側に置かれるオブジェクトは [1] で宣言したインタフェースを実装する必要がある。
- このクラス名を指定して Java 付属のスタブジェネレータ rmic を実行することでクライアントスタブ (Java ではスケルトンと呼ぶ)、サーバスタブ (Java では単にスタブと呼ぶ) が生成される。
- 呼び出し方は次の通り (☆のところにクラス名を指定)

```
% /compat/linux/usr/local/Java/j2sdk1.4.2_10/bin/rmic ☆
```

□ サーバ側コードはクラス本体部分と main() に分けて説明する。まずクラス本体部分から:

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.io.*;

public class Sample13Server
    extends UnicastRemoteObject implements Sample13IF {
  String data;
  public Sample13Server() throws RemoteException {
    data = "?";
  }
  public String exec(String s) {
    data = data + s; return data;
  }
  public String toString() {
    return data;
  }
```

□ 解説 (サーバクラス):

- extends UnicastRemoteObject の指定必要。
- implements Sample13IF の指定必要。
- 変数 data は記憶する文字列を保持。

```
  public static void main(String[] args)
      throws Exception {
    BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
    Sample13Server srv = new Sample13Server();
    Registry reg = LocateRegistry.createRegistry(4918);
    reg.bind("str1", srv);
    System.out.print("command> ");
    while(!in.readLine().equals("bye")) {
      System.out.println(srv.getData());
      System.out.print("command> ");
    }
    System.exit(0);
  }
}
```

□ 解説 (main):

- コマンド入力のための入力ストリーム用意。
- サーバオブジェクトを作る。
- レジストリ (登録サーバ) をポート 4918 で作る。
- レジストリにサーバを str1 という名前で登録。
- コマンド入力ループで、bye という行が入力されるまで 1 行入力ごとに現在のサーバの文字列表現を書き出す。

□ [3] クライアント側コード

- クライアントはこれまでと同じく GUI を持つ。
- 初期設定のところでサーバのリモートオブジェクトを取得
- あとはこれに対するメソッド呼び出し→ RMI によりサーバで実行

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.rmi.*;

public class Sample13 extends JFrame {
  Sample13IF srv;
  JTextField f1 = new JTextField();
  JButton b1 = new JButton("Exec");
  JLabel l1 = new JLabel("start...");
  public Sample13() throws Exception {
    srv = (Sample13IF)
      Naming.lookup("rmi://localhost:4918/str1");
    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container c = getContentPane(); c.setLayout(null);
    c.add(f1); f1.setBounds(40, 40, 180, 40);
    c.add(b1); b1.setBounds(240, 40, 80, 40);
    c.add(l1); l1.setBounds(20, 360, 360, 40);
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          f1.setText(srv.exec(f1.getText()));
        } catch(Exception ex) { l1.setText("!" + ex); }
      }
    });
  }
  public static void main(String[] args)
      throws Exception {
    new Sample13().setVisible(true);
  }
}
```

□ 解説:

- Naming.lookup() でホスト、ポート、名前を指定→そのホストの RMI レジストリ (先に main() で起動したもの) に接続してリモートオブジェクトを取得
- ボタン押した時は単に srv.exec() を呼ぶだけでよい (簡単)。

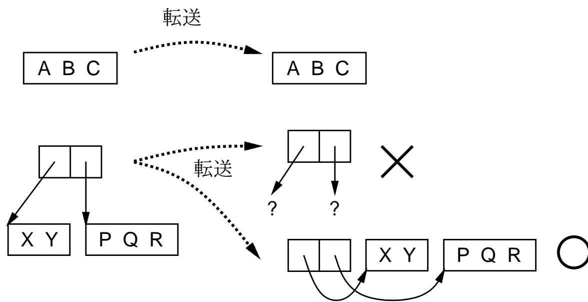
□ 演習: 文字列サービスを動かして前の演習と同様に確認してみよ。

3　オブジェクトとコードのモビリティ

3.1　モビリティ(可動性)とその内容

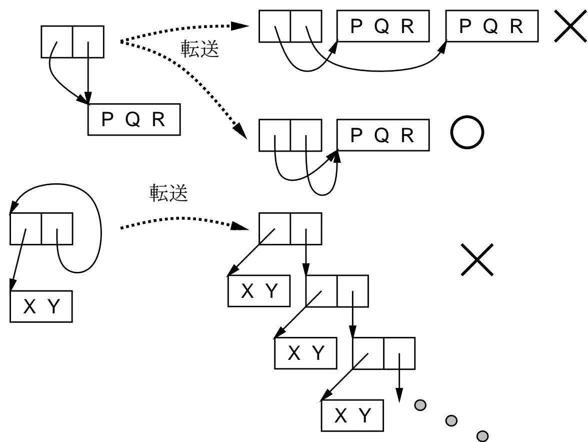
□ モビリティ(mobility)→ネットワークを通じて「××」が移動可能なこと。

- 通常のデータ(テキストとか)が移動するのは当り前だが…
- 「通常」とは?　たとえばポインタは?　→ポインタは「特定のマシンの上でのメモリアドレス→そのまま別のマシンに移動させても「とんでもない番地を指すポインタ」になるだけ。
- 正しくは「ポインタの指す先のデータ」もコピーする必要がある。



□ 構造のあるデータのコピーは必ずしも簡単ではない…

- 共有部分があったらどうするか?　→ その形を保ってコピーしたい。
- ループがあったらどうするか?　→ その形を保ってコピーしたい。



- ☆ポインタを順次たどっていくが、既に転送に着手したのと同じものに到達したら、新たに転送する代わりに「既に転送した何番目と同じ」という情報を送り、復元時にもそれに応じて復元。

□ コピー vs 参照セマンティクス

- コピーするということは、行った先と手元は別物になるということ。
- それでよい場合もあるが、オブジェクトは勝手に複製されてほしくないという場合もある。
- その場合、オブジェクトは動かずにずっとどこかに存在し、そのオブジェクトを指す参照はあちこちに渡して、それらの参照からオブジェクトを呼び出して利用、という考え方もある← CORBA など。
- しかしその方法しかないとなると、すべて RMI のようになるため遅いという問題がある。
- CORBA では言語が様々なのでどうにもならないが Java では「コピー」も「参照」も可能。参照は先にやったのがそれ。ではコピーは…

□ Java の場合は…

- Serializable インタフェースを implements で指定したクラスのオブジェクト→直列化可能(ネットワーク経由での転送、ファイルへのそのままの格納が可能)
- そのときに前記☆のような制御は用意されている。さらに特別な処理をしたい場合はそれぞれのオブジェクトにメソッドを用意することで制御可能。
- 直列化可能なクラスのインスタンス変数は基本型(単純なデータ)または再び直列化可能なクラス型でなければならない(送れる必要があるから)。
- インスタンス変数に transient というキーワードを指定しておくとその変数は送らない(復元したときは null になる)。そのため上記の制約もない。

□ コードモビリティ(モバイルコード)

- 受動的なデータだけならただ送ればよいが、オブジェクト指向言語では「データ+メソッド」がオブジェクト。
- このため、オブジェクトのデータを送ったが送った先にそのオブジェクトのメソッドのコードがなかった、では済まされない。
- 1 つの方法…あらかじめ必要なクラスのコードは用意させる。それがある状態でのみ送ることとする(普通のアプローチ)。
- 別の方法…オブジェクトを送る時、送り先にそのクラスのコードがまだなければコードも送る(Java のアプローチ) ← コードモビリティ

□ なぜ Java でこれができるか?

- もともと「アプレットのための言語」→コードをネット経由で転送することを前提として開発された言語+環境 ← i-アプリもそう

- 具体的には、Java バイトコードを送ってそれぞれの実行環境の JVM (バーチャルマシン) で実行→バイトコードは CPU 種別などに影響されずすべて共通→これを送れば OK。
- ただし、「コードが送られる」ことにはセキュリティ上の危うさがあるのでそのための工夫も必要 (今回は詳しくはやらない)。

3.2 例題:文字列サービス改訂版

□ [1] インタフェース仕様

```
import java.rmi.*;
import java.io.*;

interface Sample14IF extends Remote {
  public Data get() throws RemoteException;
  public void put(Data d) throws RemoteException;
  interface Data extends Serializable {
     public String exec(String s);
  }
}
```

□ 解説:

- インタフェース Sample14IF の中に Data というインタフェースを定義している。このインタフェースは extends Serializable と指定しているので、これを実装するオブジェクトは転送可能。
- Data オブジェクトにはこれまでサーバにあった exec() を持たせる。
- サーバは今度はこの Data オブジェクトを取り出したり戻したりするようなインタフェースに直した。

□ [2] サーバ側コード

- Data インタフェースを implements するクラス MyData を用意。このオブジェクトがクライアント側に転送され、そこで動作する。

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.io.*;

public class Sample14Server
    extends UnicastRemoteObject implements Sample14IF {
  Data data;
  public Sample14Server(Data d)
    throws RemoteException { data = d; }
  public Data get() { return data; }
  public void put(Data d) { data = d; }
  public String toString() { return data.toString(); }
  static class MyData implements Data {
    String str = "?";
    public String exec(String s) {
      str = str + s + "!"; return str;
    }
    public String toString() { return str; }
  }
  public static void main(String[] args)
      throws Exception {
    BufferedReader in = new BufferedReader(
      new InputStreamReader(System.in));
    Sample14Server srv = new Sample14Server(new MyData());
    Registry reg = LocateRegistry.createRegistry(4918);
    reg.bind("str1", srv);
    System.out.print("command> ");
    while(!in.readLine().equals("bye")) {
      System.out.println(srv.toString());
      System.out.print("command> ");
    }
    System.exit(0);
  }
}
```

□ 解説:

-
- サーバは最初 MyData オブジェクトを与えて初期化する。
- get()、put() はその Date オブジェクトをやりとり。
- toString() では Data オブジェクトの toString() を返す。
- クラス MyData はこれまでのサーバと類似しているが、通信機能はなく、単にデータを保持し計算。

□ [3] クライアント側コード

- フィールド、ボタンとも 2 つに増えている。
- 最初のフィールドには「rmi://sma:4918/str1」などの URI を与えて Conn ボタンを押すとサーバから Data オブジェクトを取得。その状態で再度 Conn ボタンを押すと取得してあった Data オブジェクトを書き戻す。
- 2 番目のフィールド/ボタンはこれまでの機能に類似。ただしサーバが処理を行うのではなく、手元に持って来た Data オブジェクトが処理を実行する。

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.rmi.*;

public class Sample14 extends JFrame {
  Sample14IF srv;
  Sample14IF.Data data;
  JTextField f0 = new JTextField();
  JButton b0 = new JButton("Conn");
  JTextField f1 = new JTextField();
  JButton b1 = new JButton("Exec");
  JLabel l1 = new JLabel("start...");
  public Sample14() throws Exception {
    setSize(400, 400);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
```

```
    Container c = getContentPane(); c.setLayout(null);
    c.add(f0); f0.setBounds(40, 40, 180, 30);
    c.add(b0); b0.setBounds(240, 40, 80, 30);
    c.add(f1); f1.setBounds(40, 80, 180, 30);
    c.add(b1); b1.setBounds(240, 80, 80, 30);
    c.add(l1); l1.setBounds(20, 360, 360, 30);
    b0.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          if(srv == null) { // connect & get
            srv = (Sample14IF)Naming.lookup(f0.getText());
            if(srv == null) l1.setText("Connect failed.");
            else { l1.setText("OK."); data = srv.get(); }
          } else { // save
            srv.put(data); srv = null; l1.setText("save.");
          }
        } catch(Exception ex) { l1.setText("!" + ex); }
      }
    });
    b1.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        try {
          if(data != null)
            f1.setText(data.exec(f1.getText()));
        } catch(Exception ex) { l1.setText("!" + ex); }
      }
    });
  }
  public static void main(String[] args)
      throws Exception {
    new Sample14().setVisible(true);
  }
}
```

□ 解説:

- 部品の配置などはこれまでと同様。
- ボタン b0 の動作では、srv が null であれば f0 に指定された文字列を使って Naming.lookup() を呼びサーバを取得。さらにサーバから Data オブジェクト取得。
- srv が null でないならそこに Data オブジェクトを書き込んで srv を null に。
- ボタン b1 の方はこれまでと同様ただし data に対して実行。

□ 演習: この例題を動かせ。文字列を加工してもサーバ側には反映されていなくて、最後に書き戻すと反映されることを確認すること。

□ 演習: クラス MyData の exec() の中身を変更して自分独自のものにしてみよ。続いて自分のサーバを動かし、動作を確認。OK なら、その rmi: URI をホワイトボードに書き、他人に使ってみてもらう (また自分も他人のものを動かしてみる)。

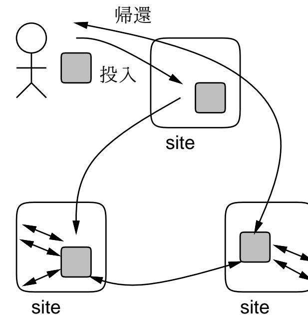
4 その他のパラダイム

□ 「ソケットによる通信」「RPC」(と RMI) がネットワーク通信の 2 大パラダイムだが、それ以外のものも→2 つほど紹介

4.1 「モバイルエージェント」

□ モバイルエージェント == 自律的に動く「コード+データ」

- 「持ち主」がエージェントを生成しネットワークに放つ
- エージェントはネットワーク上をあちこち移動しながら仕事をこなす (情報の収集や伝達など)
- 最後に「持ち主」のところに戻って来て結果を報告



- 「はじめてのおつかい」

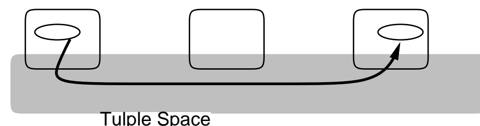
□ 予め行き先とか決めてあるのなら別に面白くない (最初からそこと通信して情報を取ればいい)

- エージェントが「自律的に」動くところがいいので、学習したりして「有用な行動を取る」ように適応していく
- このあたりは倉橋先生の専門に近いので興味があれば。

4.2 Linda(タプルスペース)

□ 1980 年代半ばころに David Gelernter が提唱した通信モデル

- すべてのノードにまたがった「場」(タプルスペース、TS) がある (ものとする)
- データはタプルと呼ばれるもので表す。「名前 (値, 値, …)」
- あるノードで投入したタプルを他のノードで取り出すことで通信



□ TS に対する操作は次の 3 つ

- out 名前 (値, 値, …) --- タプルを TS に投入する。投入したタプルは TS 中にずっと残っている

- in 名前 (引数, 引数, …) --- タプルを TS から取り出す。引数は「値」または「?変数名」。TS 中にあるタプルで、名前と値の指定された引数が一致したものを取り出す。そのとき「?変数名」のところには変数に対応するタプルの値が取り出される取り出されたタプルは TS に存在しなくなる

  out count(1) → in count(?x) (x が 1 に)

- rd 名前 (引数, 引数, …) --- in と同様だが、「読み出す」だけなのであてはまったタプルが TS 中から取り除かれない

□ アイデア: TS は全ノードにまたがっているので、どこで投入されたものでもどこででも取り出せる→位置の分離

□ アイデア: TS に入れたものはずっと残っているので受信したい人は後で受け取ることができる→時点の分離

□ アイデア: TS から取り出したタプルはなくなるので、「同じものが複数ある」ことを避けられる→整合性

□ 例: 時刻サーバ

- すべてのノードが正しく時間を知るのは簡単ではない
- TS による時刻サーバ

```
forever do
  out 時刻 (時刻の値)
  (次の時刻まで待つ)
  in 時刻 (?x)
end
```

- 時刻を知りたい人は「rd 時刻 (?t)」でいつでも時刻を知ることができる

□ 例: カウンタ

- あちこちのノードでカウントを増やす→最終的に正しいカウント値が得られるようにするのは簡単ではない (RPC とか?)
- TS によるカウンタ

```
countup:
  in カウンタ (?x)
  x = x + 1
  out カウンタ (x)
```

□ 例: RPC

- RPC も TS を使って簡単に実装することができる

```
サーバ側:
  forever do
    in rpc(?proc, ?arg, ?ret)
    手続き proc の処理を arg を用いて実行
    out reply(ret, 返値)
  end

クライアント側:
  id = 自分固有の番号
  rpc("手続き名", 引数, id)
  reply(id, ?ret)
```

□ 大変スマートでかっこいいので、色々な人が実装したりしている。

- Sun でも JavaSpaces とかそれをもっと実用的にした Jini などのフレームワークを開発
- しかしどれもいまいち実用にならなかった。いろいろな問題
- タプルをどうやって投入箇所から読み出し箇所へ運ぶか?
- バグありプログラムが山のようにゴミを投入したら?
- 悪意あるプログラムがタプルを盗み見たら?

□ でもまあ、知っておいて損はない面白いおはなし。興味ある人は「Generative communication in Linda」という論文を読むとよい (ネットで検索すると取れる)

5 まとめ

□ 今回は前回の補足としてステートフルなサーバの実装 (スレッドの使用) についてとりあげた。その後、分散オブジェクトモデル、ORB、Java RMI、モビリティ、エージェント、Linda などの話題をひととおり取り上げた。次回から Web アプリの方に進む予定。