ネットワークアプリケーション

第4回 アプリケーションプロトコルの設計 (2)

石井 健太郎
(423研究室・オフィスアワー水3限)

- レポートを提出してください
  - 講義終了時でも可

スケジュールの微修正をします

- 第5回・第9回・第13回も第3演習室で行います
  - ここまで少し思ったより順調に進捗しているため実践の時間を増やします

- 来週10月20日（火）は**第3演習室**に集まってください

スケジュール

9月15日　第1回「TCP/IPプロトコルスイート」
9月29日　第2回「ネットワークアプリケーションのプログラミングモデル」
10月6日　第3回「アプリケーションプロトコルの設計（1）」
10月13日　第4回「アプリケーションプロトコルの設計（2）」
10月20日　第5回「アプリケーションプロトコルの設計（3）」　演習（第3演習室）
10月27日　第6回「アプリケーションプロトコルの設計（4）」　演習（第3演習室）
11月10日　第7回「サーバサイドウェブプログラミング（1）」
11月17日　第8回「サーバサイドウェブプログラミング（2）」

スケジュール

11月24日	第9回「サーバサイドウェブプログラミング（3）」	演習（第3演習室）
12月1日	第10回「サーバサイドウェブプログラミング（4）」	演習（第3演習室）
12月8日	第11回「クライアントサイドウェブプログラミング（1）」	
12月15日	第12回「クライアントサイドウェブプログラミング（2）」	
12月22日	第13回「クライアントサイドウェブプログラミング（3）」	演習（第3演習室）
1月12日	第14回「クライアントサイドウェブプログラミング（4）」	演習（第3演習室）
1月19日	第15回「まとめと演習」	演習（第3演習室）

どのようなデータ形式にすればよいか？

- 可能な限り，ASCII文字列で1行1コマンド/レスポンスとするのがよい
  - ASCII文字列にすることで，
    - バイトオーダーを気にしなくてよい
    - 空白・カンマといった，アルファベットや数値には出現しない文字を区切りに利用できる
      - 可変長データが表現しやすい
    - 送信/受信したデータそのままで人間が読めるのでデバグが容易
  - 1行1コマンド/レスポンスにすることで，
    - 1つの送信に対して相手からの受信を期待することで，
      アプリケーション同士が足並みをそろえやすい（デッドロックになりにくい）
      - 改行文字までを1つのデータとする送信を常に考える
    - 1つのデータ単位が読みやすいのでやはりデバグが容易

実例

- ASCII文字列で1行1コマンド/レスポンスとする
  - <id>␣<locationX>␣<locationY>␣<orientationX>␣<orientationY>¥n

  - 送信側

```
char line[1024];
sprintf(line, "%d %f %f %f %f¥n", id, locationX, locationY, orientationX, orientationY);
send(socket, line, strlen(line), 0);
```

  - 受信側

```
char line[1024];
length = recv(socket, line, sizeof(line), 0);
sscanf(line, "%d %lf %lf %lf %lf", &id, &locationX, &locationY, &orientationX, &orientationY);
```

実例

- ASCII文字列で1行1コマンド/レスポンスとする
  - <id>␣<locationX>␣<locationY>␣<orientationX>␣<orientationY>¥n

  - 送信側

```
writer = new PrintWriter(socket.getOutputStream(), true);
writer.println(id + " " + locationX + " " + locationY + " " + orientationX + " " + orientationY);
writer.flush();
```

  - 受信側

```
reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));
line = reader.readLine();
tokens = new StringTokenizer(line);
id = Integer.parseInt(tokens.nextToken());
locationX = Double.parseDouble(tokens.nextToken());
locationY = Double.parseDouble(tokens.nextToken());
orientationX = Double.parseDouble(tokens.nextToken());
orientationY = Double.parseDouble(tokens.nextToken());
```

1行1コマンド/レスポンスにできない場合

- 区切り行を設ける
  - 空行
  - ピリオドだけの行
  - 「250␣」で始まる行
  - 「FieldCards ＊」がきたら

実例

- 1行1コマンド/レスポンスにできない場合
  - 区切り行を設ける

（※赤字は入力）

```
> telnet is.is.oit.ac.jp 587
220 is.is.oit.ac.jp ESMTP Postfix (Ubuntu)
EHLO is.is.oit.ac.jp
250-is.is.oit.ac.jp
250-PIPELINING
250-SIZE 10240000
250-VRFY
250-ETRN
250-STARTTLS
250-ENHANCEDSTATUSCODES
250-8BITMIME
250 DSN
```

バイナリデータを送らなければいけないときは？

- まず考えるべきは，送るデータを固定長にできないか？
  - ただし，たいていの場合，固定長の効率はよくない
    - cf. バイナリデータの必要性

- 固定長にできない（したら意味がない）場合，以下のいずれかを考える
  - データには出現しない区切りバイト（バイト列）を使用する
  - 決まったバイト位置にデータサイズを埋め込む
  - パケットサイズでデータサイズを判断する
    - 一部しか届かなかったり（TCP）失われたり（UDP）することがあるので，
      それを許容するアプリケーションのみ
    - 十分なバッファサイズの確保が必要

実例

- 送信データを固定長にする

```
struct marker {
  long id;
  float locationX;
  float locationY;
  float orientationX;
  float orientationY;
}
```

- 送信側

```
network_id = htonl(marker.id);
network_locationX = htonl(marker.locationX);
network_locationY = htonl(marker.locationY);
network_orientationX = htonl(marker.orientationX);
network_orientationY = htonl(marker.orientationY);
unsigned char* data[20];
memcpy(data, &network_id, 4);
memcpy(data + 4, &network_locationX, 4);
memcpy(data + 8, &network_locationY, 4);
memcpy(data + 12, &network_orientationX, 4);
memcpy(data + 16, &network_orientationY, 4);
send(socket, data, sizeof(data), 0);
```

- 受信側

```
unsigned char data[20];
size = recv(socket, data, sizeof(data), 0);
marker.id = ntohl(*((long*)data));
marker.locationX = ntohl(*((float*)(data + 4)));
marker.locationY = ntohl(*((float*)(data + 8)));
marker.orientationX = ntohl(*((float*)(data + 12)));
marker.orientationY = ntohl(*((float*)(data + 16)));
```

実例

- 送信データを固定長にする

```
struct marker  {
  long id;
  float locationX;
  float locationY;
  float orientationX;
  float orientationY;
}
```

- 送信側

```
send (socket, &marker, sizeof (marker) , 0) ;
```

- 受信側

```
size = recv (socket, &marker, sizeof (marker) , 0) ;
```

実例

```
class Marker  {
  long id;
  float locationX;
  float locationY;
  float orientationX;
  float orientationY;
}
```

- 送信データを固定長にする
  - 送信側

```
output = new DataOutputStream(socket.getOutputStream());
output.writeLong(marker.id);
output.writeFloat(marker.locationX);
output.writeFloat(marker.locationY);
output.writeFloat(marker.orientationX);
output.writeFloat(marker.orientationY);
output.flush();
```

  - 受信側

```
input = new DataInputStream(socket.getInputStream());
id = input.readLong();
locationX = input.readDouble();
locationY = input.readDouble();
orientationX = input.readDouble();
orientationY = input.readDouble();
```

実例

- データには存在しない区切りバイト(バイト列)を使用する
  - 一枚の画像中のみつかったマーカのIDを列挙する
    - 0のIDを持つマーカはないという前提とすると
    - 35, 2, 43, 119, 15, 0
      というようなバイト列は5つのIDのマーカが見つかったことを意味する

実例

- 決まったバイト位置にデータサイズを埋め込む
  - <id>, <version>, <linkCount>, <link>, <link>, <link>, ...

  - 送信側

```
unsigned char data[1024];
data[0] = id;
data[1] = version;
data[2] = linkCount;
for (int i = 0; i < linkCount; i++)
  data[i + 3] = links[i];
send(socket, data, 3 + linkCount, 0);
```

  - 受信側

```
unsigned char data[1024];
size = recv(socket, data, 3, 0);
id = data[0];
version = data[1];
linkCount = data[2];
size = recv(socket, data + 3, linkCount, 0);
for (int i = 0; i < linkCount; i++)
  links[i] = data[i+3];
```

実例

- パケットサイズでデータサイズを判断する
  - 送信側

```
unsigned char data[1024];
<データの準備>
size = <データのサイズ>;
sendto(socket, data, size, 0, &address, sizeof(address));
```

  - 受信側

```
unsigned char data[1024];
size = recvfrom(socket, data, sizeof(data), 0, &address, sizeof(address));
```

実例

- パケットサイズでデータサイズを判断する
  - 送信側

```
data = new byte [1024] ;
<データの準備>
size = <データのサイズ>;
packet = new DatagramPacket (data, size, address) ;
socket.send (packet) ;
```

  - 受信側

```
byte []  buffer = new byte [1024] ;
packet = new DatagramPacket (buffer, buffer.length) ;
socket.receive (packet) ;
size = packet.getLength () ;
data = packet.getData () ;
```

やりとりの手順を考える

Server		Client
（カードを振り分ける）		
	<--	（connect）
PlayerCards 0 *	-->	
PlayerCards 1 *	-->	
FieldCards *	-->	
（ほかのクライアントを待つ）		
（クライアントが人数分接続したら）		
Start	-->	
Turn 0 0	-->	

	<--	Hand <id> <card>
Hand <id> <card>	-->	
Turn 1 1	-->	
	<--	Hand <id> <card>
Hand <id> <card>	-->	
Turn 0 2	-->	
	<--	Claim <id>
Claim <id>	-->	（ダウトの吹き出し・カードを開く）
（ダウトの判定）		
（3秒待つ）		
PlayerCards 0 *	-->	
PlayerCards 1 *	-->	
FieldCards *	-->	
Turn 0 2	-->	

思ったとおりのデータがこなかったとき

- データがフォーマットにそっていない
  - 無視する
  - 再送を要求する

  - TCPではまず起こらない(プログラムが間違っている場合を除く)が
    UDPやシリアル通信ではありがちなので，対応を考えておく必要がある

- 届くはずのデータが届かない
  - もう1度データを要求する
  - あきらめる

  - タイムアウトにより次の処理を行う
  - プロトコルの設計(やりとりの手順)をもう1度見直す
    - デッドロックになっていないかを確認する

- 本日宿題はありません
- 来週10月20日（火）は**第3演習室**に集まってください