経営情報システム学特論１
８．メッセージ通信

SS専攻 経営情報システム学講座 客員
石川 冬樹
f-ishikawa@nii.ac.jp

目次

- メッセージ通信
- サーバクライアントのモデル例：ランデブー
- 分散システムの特徴：
  サーバクライアントの検証例

メッセージ通信

- これまでは共有ラベルを通して複数プロセスのやりとりを表現していた
  - これが直接設計を表すと見なすと，共有変数（グローバル変数）に相当する
- 実際の設計においては，メッセージ通信を用いることが多い
  - 複数コードをより適切に分離（共有変数の自由な書き換えにより不整合につながるような，必要以上の依存関係を持たないようにする）
  - 複数コードの開発者が異なる分散システムでは，通常メッセージ通信が中心となる

同期通信

- 同期通信
  - 送信側と受信側双方が送受信を実行になるまでいずれもブロックされる
  - 資源が高々1個しか置けない場合の，消費者と生産者（第4回）のような振る舞いとなる
  - 実世界では「ネットワーク通信処理」が通常はさまるが，部分を扱った先（受信側）では，上記のような振る舞いになる
    （今回のJavaコードはその部分だけ）

同期通信

- プロセス代数では下記のように同期通信を表現することが非常に一般的
  - c!x ： チャンネルcから値xを送る
  - c?x ： チャンネルcから受け取った値をxに入れる
- LTSAでは導入していないが，
  - c というラベルを用意し，片方がsend，もう片方がrecvだと思えば，同期メッセージ通信は表現可能
  - あるいは，便宜上c.sendとc.recvとして，ラベル置換によりcに揃えて同期させる
  - ただし，送信者または受信者が複数いるとき，送信同士，受信同士が同期しないように適切にラベルを区別する必要がある（ a:CL || b:CL || {a,b}::SV ）

同期通信： LTSA

```
const N = 3
range T = 0..N

CLIENT = CLIENT[0],
CLIENT[i:T] = (
 chan.send[i] -> CLIENT[(i+1)%N]
).

SERVER = ( chan.recv[i:T] -> SERVER ).

||SYSTEM = ( CLIENT || SERVER )
                /{chan/chan.{send,recv}}.
```

ラベル置換

同期通信： Java

```
public synchronized void send(T value)... {
   chan = value;
   ready = true;
   if (open) {
     notifyAll();
   }
   while (chan != null) { wait(); }
}
```

値が準備できたため,
もし待っている（openな）recvが
あれば起こす

値が読み込まれるまで
ブロックする

同期通信： Java

```
public synchronized T recv() ... {
  open = true;
  while (!ready) { wait(); }
  open = false;

  ready = false;
  T ret = chan;
  chan = null;

  notifyAll(); // should be notify()
  return (ret);
}
```

値が準備されるまで
ブロックする

値が読み込まれるのを
待っているsendを起こ
す（今回は1体に限る）

非同期通信

- 非同期通信
  - 送信側はブロックされずにメッセージを送ることができる
  - 実際にはメッセージは有限のキューに保持され，キューがいっぱいの場合には送信がまたされるか，古いメッセージが捨てられる
- 通信に焦点を当てた実用的な言語では，チャンネルに対してキュー長を与えることができることもある（PROMELA言語など）
- LTSAでもJavaでもキューを表現すればよい

非同期通信： LTSA

```
const N = 3
range T = 0..N

PORT = ( chan.send[i:T] -> PORT[i] ),
PORT[i:T] = ( chan.recv[i] -> PORT ).

CLIENT = CLIENT[0],
CLIENT[i:T] = (
 chan.send[i] -> CLIENT[(i+1)%N]
).

SERVER = ( chan.recv[i:T] -> SERVER ).

||SYSTEM = ( CLIENT || SERVER || PORT ).
```

1個だけ値を
保持できるキュー

非同期通信： LTSA

```
PORT = ( chan.send[i:T] -> PORT[i] ),
PORT[i:T] = (
  chan.send[j:T] -> PORT[i][j]
  | chan.recv[i] -> PORT
),
PORT[i:T][j:T] = (
  chan.recv[i] -> PORT[j]
).
```

2個まで値を
保持できるキュー

非同期通信： LTSA

起きうる引数のリストの集合

```
set S = {[T], [T][T], [T][T][T]}
PORT = ( chan.send[i:T] -> PORT[i] ),
PORT[i:T] = (
  chan.send[j:T] -> PORT[i][j]
  | chan.recv[i] -> PORT
),
PORT[i:T][s:S] = (
  chan.send[j:T] -> PORT[i][s][j]
  | chan.recv[i] -> PORT[s]
).
```

引数のリストを引数にしてしまうと，
長さ2以上について汎用表現が可能

非同期通信： Java

```
public synchronized void send(T value) ...{
   queue.addLast(value);
   ready++;
   if (open) {
      notifyAll();
   }
}
```

先ほどのものからキューに変更,
readyは真偽ではなく個数カウントに変更

値が読み込まれるまで待たない

非同期通信： Java

```
public synchronized T recv() ... {
  open = true;
  while (ready == 0) { wait(); }
  open = false;
  ready--;
  return queue.removeFirst();
}
```

先ほどのものからキューに変更,
readyは真偽ではなく個数カウントに変更

送信側はブロックしないので起こす処理はない

目次

- メッセージ通信
- サーバクライアントのモデル例：ランデブー
- 分散システムの特徴：
  サーバクライアントの検証例

ランデブー

- クライアントはサーバ側に対しcallを呼ぶ
  - クライアントのリクエストに対応する「エントリー」ができあがる
  - 「エントリー」内には，クライアントのリクエスト内容と，クライアントへの返信チャンネルが作られ保持される
- サーバは「エントリー」をacceptしてリクエスト内容を取り出し，replyする
  - acceptではリクエスト内容を取り出す
  - replyではエントリーに保持された返信チャンネルを通して返答を返す

ランデブー：LTSA

```
const N = 3
range T = 0..N
set REP = { reply1, reply2 }

PORT = ( chan.send[i:T] -> PORT[i] ),
PORT[i:T] = ( chan.recv[i] -> PORT ).
||ENTRY = entry:PORT/{call/send, acpt/recv}.

CLIENT1 = ( entry.call.reply1 -> reply1 ->
CLIENT1 ).

SERVER = ( entry.acpt[ch:REP] -> [ch] ->
SERVER ).

||SYSTEM = ( CLIENT1 || ENTRY || SERVER ).
```

返信ラベルが「引数」となる

ランデブー：LTSA

返信ラベルをパラメーター化
クオーテーションは，ラベル名を
指すときのprefix（変数名と区別）

```
...

CLIENT(CH='reply1) = ( entry.call[CH] ->
[CH] -> CLIENT ).

||SYSTEM = ( CLIENT('reply1) ||
CLIENT('reply2) || ENTRY || SERVER ).
```

ランデブー： Java

```
public P call(R req) ... {
  Channel<P> clientChan = new Channel<P>();
  cp.send(new CallMsg<R, P>
            (req, clientChan));
  return clientChan.recv();
}

public R accept() ... {
  cm = cp.recv();
  return cm.request;
}

public void reply(P res) ... {
  cm.replychan.send(res);
}
```

返信用チャンネル生成

リクエスト内容と一緒にサーバへ送信

返信用チャンネルへの返事を待つ

リクエスト内容取り出し

返信用チャンネルから送信

目次

- メッセージ通信
- サーバクライアントのモデル例：ランデブー
- 分散システムの特徴：
  サーバクライアントの検証例

基本例題： 概要

- サーバに印刷処理を依頼する（RPC）
  - プロトコル例
    - クライアントはリクエストをサーバに送信
    - サーバは処理を実行
    - サーバは確認メッセージを送り返す
  - 問題： サーバがクラッシュするとどうなる？
    - 後に復帰し，復帰した旨を伝える
    - まずは，メッセージ配信は確実に成功すると仮定する

基本例題：クラッシュの影響

- クラッシュが起きるタイミングごとに，サーバ側で起きることを考えてみる
  - クラッシュ （未印刷，未確認）
  - 印刷 → クラッシュ （未確認）
  - 印刷 → 確認メッセージ → クラッシュ
- クライアントから見ると，1個目と2個目の状況は確認メッセージが来ないという同じ状況に見える
  - 再送すると，2個目では2回印刷してしまう
  - 再送しないと，1個目では印刷されない

基本例題： 対応方針

- 確認できない場合，メッセージを再送し，再依頼
- サーバ側には，（処理が完了していても）二回以上同じメッセージが到達する可能性がある
  - メッセージID管理をし，高々1回しかメッセージを処理しないようにする
  - あるいは，何度実行されても結果が同じとなる冪等（べきとう，idempotent）処理であるように設計，確認しておく
    - 例： 送付された情報を，専有されており名前が固定の指定名のファイルに書き込む

基本例題： ネットワークの考慮

- メッセージ配信の失敗を考えると，さらにややこしくなる
  - 確認がないことが，サーバのクラッシュなのか，ネットワークの問題なのか，一般にはクライアント側からはわからない
  - 処理が終わったかどうかを知ることができないという点はもとよりそうで，対応方針は同じとなる

演習：クライアントサーバ

- 下記ファイルに記されたLTSAを確認せよ
  - client_server.lts
  - 同期通信（失敗の可能性あり）によるクライアントサーバ
  - サーバは印刷を行い，成功通知を行う（ただしこれも失敗の可能性あり）
  - クライアントはタイムアウト時に1回だけリクエストを再送
  - 「印刷成功か？」に関するクライアントの理解と，実際にサーバで起こったこととのずれはありうる？
- LTL式を読み，各LTL式の検証結果を予想するとともに，実際にLTSA Analyzerで検証してみよ

並行システムと分散システム

- 分散システムでは，ネットワーク障害のため相手のことを確実に知ることができないことを前提とするという問題が加わる
- ただし，並行システムの時点で存在する問題も十分難しい
  - 資源の待ち合いによるデッドロック
  - 値を聞いてから書き込む間に値が変わっており，ただしくカウントできない（ロック忘れ）
  - 適切なwait/notify機構と，入れ子のロックにおける適切なロック解放
  - ・・・

まとめ

- メッセージ通信
  - 分散システムの基本部品として，また並行システムでの疎結合な協調手段として広く用いられる
  - プリミティブとしては同期，非同期通信があり，これまでと似た考え方で表現できる

- 次回： 並行システムにおける有名なアーキテクチャーを議論する