経営情報システム学特論１
９．並行アーキテクチャ

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

- 並行アーキテクチャの例： Pipe and Filter
- 並行アーキテクチャの例： Tuple Space

並行アーキテクチャ

- 並行システムにおいて典型的に用いられるアーキテクチャは多くある
  - 一例を紹介，LTSA/Java版にて議論
- 実際には「分散システム」で典型的に用いられる
  - 複数のプロセッサーがあることにより，処理が並列化され，高速になる・負荷分散される
  - リモート呼び出しをラッピングする機構を設ければ「分散」にできる
    ※ ただしネットワーク障害を考えると，別の問題が生じる

Pipe and Filter

- Pipe and Filter
  - 値を条件に応じて取り除いていくFilterをつなげて機能を実現する

注：Pipeと呼ばれる概念は，計算内容をFilterに限らず，複数の計算における入出力をつなげる広く用いられる

- 例：コマンドライン
- が，コマンドラインの場合は並行動作ではなく，「断続的に」値が流れるわけではない

Pipe and Filter： モデル化

- Pipe and Filterにおける部品化
  （今までも出てきて暗黙的であったが）
  - 何個も部品をコピペで作り込むのではなく，1種類の部品をいくつも作る（インスタンス化する）方がよい（LTSAでもJavaでも）

Filter[1]のoutとPipe[1]のinをつなげ，

Filter[2]のinとPipe[1]のoutを，
Filter[2]のoutとPipe[2]のinをつなげ，

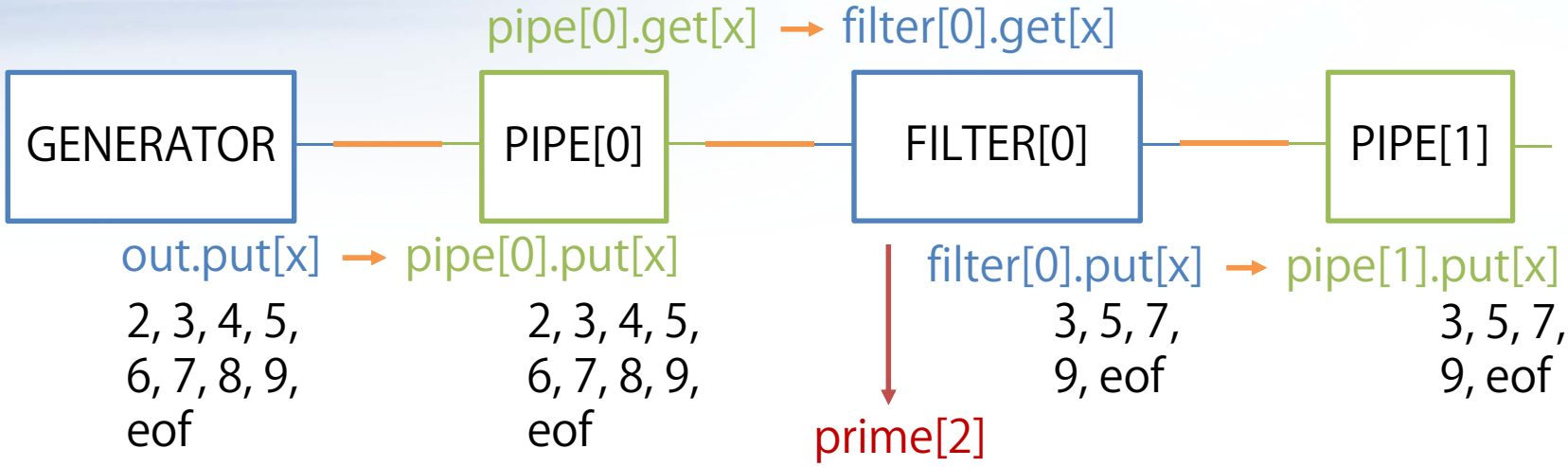
．．．

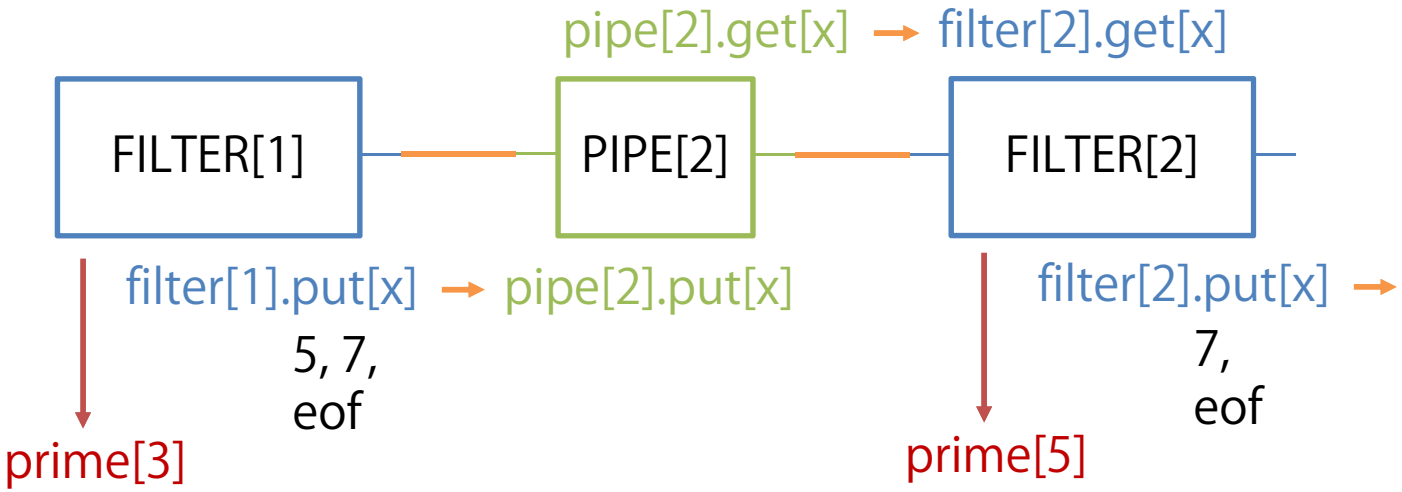
Pipe and Filter： 例題

- エラトステネスのふるい
  - 素数を求めるアルゴリズム
  - 2は素数，以降は2の倍数を除く
  - 残ったもので次に大きい3は素数，以降は3の倍数を除く
  - （4は2の倍数として除かれている）
  - 残ったもので次に大きい5は素数，以降は5の倍数を除く
  - （6は2の倍数として除かれている）

これの並列版：各Filterは，自分が最初に受け取った値の倍数はすべて除いて次に送るようにする

Pipe and Filter： 例題





Pipe and Filter： LTSA版

```
const MAX = 9
range NUM = 2..MAX
set S = {[NUM],eos}

GENERATOR = GENERATOR[2],
GENERATOR[x:NUM] = (
  out.put[x] ->
  if (x<MAX) then GENERATOR[x+1]
  else (out.put.eos->end->GENERATOR)
).
```

扱う値は2以上

やりとりされるのは
値か終了通知EOS

入力値を送り出す
プロセス

2からMAXまで
送り最後はEOS

Pipe and Filter： LTSA版

初めて受け取った値

素数として出力

```
FILTER = (
  in.get[p:NUM] -> prime[p] -> FILTER[p]
  | in.get.eos -> FILTEREND
),
FILTER[p:NUM] = (
  in.get[x:NUM] ->
    if x%p!=0 then (out.put[x] -> FILTER[p])
    else FILTER[p]
  | in.get.eos -> FILTEREND
),
FILTEREND = ( out.put.eos -> end -> FILTER ).
```

その値の倍数をフィルターするプロセスに移行

割り切れないときだけ次に送る

Pipe and Filter： LTSA版

前述の前後の紐付けのほか，
全Filterの出力をGeneratorの
endに同期させている

```
PIPE = ( put[x:S] -> get[x] -> PIPE ).

||PRIMES(N=4) = (gen:GENERATOR
                 || pipe[0..N-1]:PIPE
                 || filter[0..N-1]:FILTER)
                /{ pipe[0]/gen.out,
                   pipe[i:0..N-1]/filter[i].in,
                   pipe[i:1..N-1]/filter[i-1].out,
                   end/{filter[0..N-1].end,gen.end}
                 }
                 @{filter[0..N-1].prime,end}.
```

外部から観察可能なラベルは結果に関するものだけ
（get/putは内部動作τだと見なす）

Pipe and Filter： LTSA版

Pipe（バッファ）なしの
バージョン（比較用）

```
||PRIMESUNBUF(N=4) = (gen:GENERATOR
                      || filter[0..N-1]:FILTER)
                   /{ pipe[0]/gen.out.put,
                      pipe[i:0..N-1]/filter[i].in.get,
                      pipe[i:1..N-1]/filter[i-1].out.put,
                      end/{filter[0..N-1].end,gen.end}
                   }@{filter[0..N-1].prime,end}.
```

前のFILTERのout.putと
後ろのFILTERのin.getを
同じ名前（pipe）に揃えて直接同期

Pipe and Filter： LTSA版

- 結局重要なこととしては，並行動作で実装するのは一つの案として，結局外部から見てどういう動きをするのか
  - Pipe（バッファの役割を持ち，計算中であってもブロックされるのを防ぐ）がある場合とない場合

- LTSAツールの機能： Minimize
  - @は「外に見える」ラベルを指定する（それ以外のラベルは内部動作τと見なす）が，τが消去される

⇨ 今回の2つのバージョンをともにMinimizeしたときに，外部から見て同一の動作になっているか？？？

Pipe and Filter： 補足

- 今回の場合，与えられた範囲ですべての素数を求められるかは，Filterの個数に依存する
  - LTSAだとプロセス数は有限なので，値に応じて設定する必要がある
  - Javaだと動的にFilterを足すことはできなくもないが複雑（配布コードではやっていない）

Pipe and Filter： Java版

- 配付資料参照
  - Pipeが上限無しのUnboundedBuffer版と，1個しか値を同時に持てないUnbuffer版あり
  - Filterは単に自身のことだけ把握して，与えられた情報を基に受け取ったデータに対し，仕事をし，データを送り出すだけ
    （Pipeも同じくデータを貯めるだけ）
  - Filter，Pipeを作成し呼び出す側（Primesクラス）で接続の指示をしている

目次

- 並行アーキテクチャの例： Pipe and Filter
- 並行アーキテクチャの例： Tuple Space

Tuple Space

- Tuple Space
  - 分散・並列データアクセスのためのモデルである Linda [Carriero and Gelernter 1989] において実装された，常に有名な並列アーキテクチャ
  - 共有空間に対して下記の操作を行うことができる

    out： 値一式を，「タグ」に対し紐付けて書き込む

    in： 「タグ」に紐付けられた値一式を読み出す（読み出されたものは消える）

    rd： 「タグ」に紐付けられた値一式を読み出す（読み出されたものは消えない）

    in/rdはブロッキング，ノンブロッキングバージョンのinp/rdpを考える場合もある

Tuple Space

- （タグに対するルールを決めれば）様々なタスク分担アーキテクチャを実現できる
- 例： Supervisor – Worker
  - Supervisorはタスクを書き込み，その結果を読み出していく
  - Workerはタスクを読み出して，その結果を書き出していく

  （多対多でもよい）
- 例： 様々な種類のWorkerがいて，Workerも自身のサブタスクをどんどん他のWorkerに依頼していく

Tuple Space： LTSA版

```
const False = 0
const True  = 1
range Bool = False..True
const N            = 2
set   Tuples       = {tag1, tag2}

TUPLE(T='tag1) = TUPLE[0],
TUPLE[i:0..N]
  = (out[T]                          -> TUPLE[i+1]
    |when (i>0) in[T]                -> TUPLE[i-1]
    |when (i>0) inp[True][T]         -> TUPLE[i-1]
    |when (i==0)inp[False][T]        -> TUPLE[i]
    |when (i>0) rd[T]                -> TUPLE[i]
    |rdp[i>0][T]                     -> TUPLE[i]
    ).
||TUPLESPACE = forall [t:Tuples] TUPLE(t).
```

保持する値は
表現していない

Tuple Space： Java版

```
public interface TupleSpace {

  public void out (String tag, Object data);

  public Object in (String tag)
    throws InterruptedException;

  public Object rd (String tag)
    throws InterruptedException;

  public Object inp (String tag);

  public Object rdp (String tag);

}
```

Tuple Space： Java版

```
public class TupleSpaceImpl
    implements TupleSpace {

 private HashMap<String,ArrayList> tuples =
    new HashMap<String, ArrayList>();

  public synchronized void out
    (String tag, Object data) {
    ArrayList v = tuples.get(tag);
    if (v==null) {
        v=new ArrayList();
        tuples.put(tag,v);
    }
    v.add(data);
    notifyAll();
}
```

Tuple Space： Java版

```
public synchronized Object in (String tag)
  throws InterruptedException {
    Object o;
    while ((o=get(tag,true))==null){ wait(); }
    return o;
}

public synchronized Object rd (String tag)
  throws InterruptedException {
    Object o;
    while ((o=get(tag,false))==null){ wait(); }
    return o;
}
```

Tuple Space： Java版

```
public synchronized Object inp (String tag) {
    return get(tag,true);
}

public synchronized Object rdp (String tag) {
    return get(tag,false);
}
```

まとめ

- 並行システム（分散システム）におけるアーキテクチャ
  - 汎用性の高い典型的なアーキテクチャを用意する・知ることにより，アプリケーション固有の部分の実装に集中することができるようになる
  - 特にwait/notifyなどの具体的な仕組みは隠蔽した形で多くのライブラリー，フレームワークが利用可能である
  - 分散システムの場合はネットワーク障害に関する考慮が必要となる
- 次回：時間の概念を扱う