# Concurrent processes and their description

並行プロセスの挙動とその記述

慶應義塾大学名誉教授
徳田英幸



# ちょっと復習。。。

# 基本的な概念：自律分散協調

- 自律性
  - 個の確立
  - 主体的行動
- 分散性
  - 多数の個
  - 空間的・ネット的に分散
- 協調性
  - 個と個の協調プロトコル
  - 協調により全体の機能を維持・形成する
  - 構成論的手法　vs. 自己組織論的手法
- システムとしての評価
  - 評価の軸
  - 良いシステム vs. 悪いシステム



# プロトコルの記述
## ～N-way Protocols～

慶應義塾大学名誉教授
徳田英幸

# プロトコルとは？

～協調動作の記述～



# プロトコルとは？

- 外交上の儀礼
- 通信の送信側と受信側で取り決めた約束事
- 通信手段

pro･to･col

1 U(外交上の)儀礼, 典礼.

2 C 条約原案; 議定書, プロトコル.

3 C (国家間の)協定.→＃

4 C《米》(実験･治療の)実施要綱[計画].

5 C【電算】プロトコル《データ通信の手順》.

[株式会社研究社　新英和･和英中辞典]

## プロトコルの記述

- プロトコルの記述は、通信上の約束事すべてを定義する。
- 通信メッセージのフォーマット（書式）の詳細 (format) -> (syntax)
- メッセージを交換する際の手順 (procedure) -> (grammer)
- 正しいメッセージが表わしている意味、語彙 (correctness) -> (semantics)
- Completeな記述 vs. Incompleteな記述



## 何のためにプロトコルが必要か？

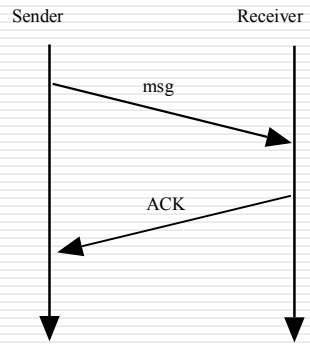
- データ転送に必要な初期化や終了
- 送信者と受信者との同期（条件）
- 転送エラーの検出や訂正
- データの書式や符合化を行う

## Protocolの記述

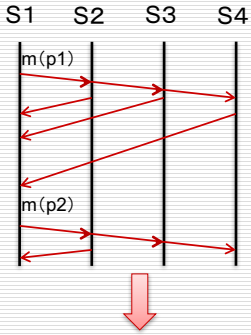
- 自然言語
- Time-space chart
- 状態遷移図
- 擬似Prog. Languages



## Time-Space Chart



# 入札プロトコル（1）



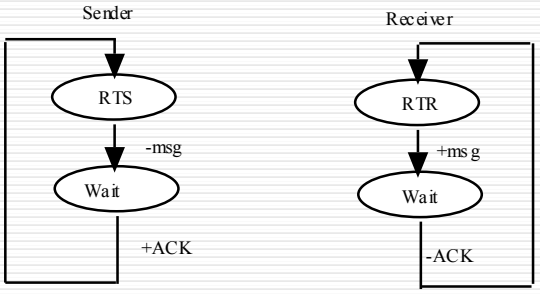

Q: 終了条件は？
　winnerへの通知は？

Q：タイムアウトは？
　全員がいつもタイムリーに返信？

Note: S1からのグループキャストは、ユニキャストと区別するために各受信者のところで矢印マークを入れる



# State Transition Diagram状態遷移図



# Concurrent Programming Language

```
Process Sender( )
{
   while(TRUE) {
      prepare_message(buffer);
      frame.data = buffer;
      send_to_network(frame);
}
```

```
Process Receiver( )
{
   while(TRUE) {
      wait_for_frame(buffer);
      receive_from_network( frame);
      buffer = frame.data;
}
```



# Stop-and-Wait Protocol

```
Process Sender( )
{
   while(TRUE) {
      prepare_message(buffer);
      frame.data = buffer;
      send_to_network(frame);
      wait_for_frame(ack);
      . . .
}
```

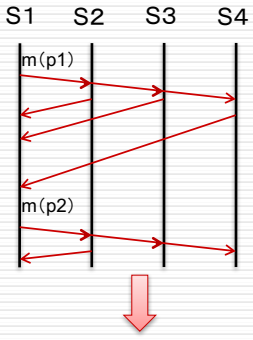
```
Process Receiver( )
{
   while(TRUE) {
      wait_for_frame(buffer);
      receive_from_network( frame);
      buffer = frame.data;
      prepare_message(ack);
      frame.data = ack;
      send_to_network(frame);
}
```

# いくつかの例題

- 入札プロトコル
- 回覧板プロトコル
- 仮想リングプロトコル
- 逆オークションプロトコル



# 入札プロトコル（1）

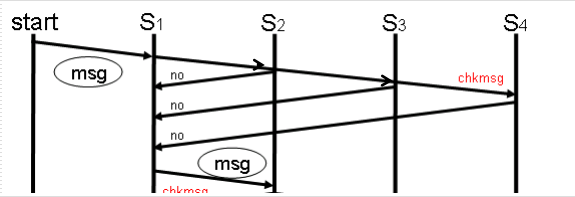



Q: 終了条件は？
　winnerへの通知は？

Q:タイムアウトは？
　全員がいつもタイムリーに返信？

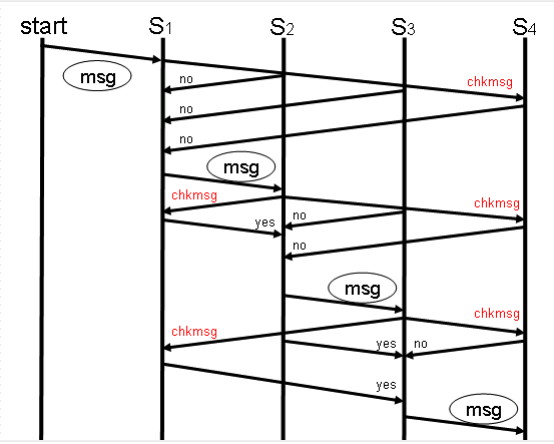
Note: S1からのグループキャストは、ユニキャストと区別するために各受信者のところで矢印マークを入れる

# 回覧板プロトコル（1）






# 回覧板プロトコル(2)



# 自律分散協調システム
# 分散アルゴリズム(1)

慶應義塾大学名誉教授
徳田英幸



## 演習-1: Dist. Partitioned Sort

- 1からNまでの整数の分散ソート
  - Distributed Sort for the numbers between 1 and N
  - Initial value, 100 cards, N=1,000
  - Processes: P1- P10
  - 各プロセスは、P1から順に横一列に並んでおり、自分の隣接しているプロセスとのみ交信出来るものと仮定する。
- 問題
  - P1からP10までの各プロセスにランダムに選ばれた10個の整数が初期の値として与えられた時、どのようにしてソートすることができるか？
  - 各プロセスは、どのような条件で、ソートが終了したことを判定できるか？

## 分散プログラムにおける仮定 （1）

- それぞれのプロセスの動作スピードに関して仮定してはならない。
  - e.g. ×：同一である
  - e.g. ×： P1はP2の10倍のスピード
- 各プロセスのアドレス空間は、閉じており、ローカルな変数しかアクセスできない。
- 各プロセスのクロックは、同期していない
- メッセージの伝送スピードに対しても仮定してはならない。



## 分散プログラムにおける仮定 (2)

- 仮定しないと難しいもの
  - プロセスは、故障しない。
    - プロセスは、任意の時点で故障し、終了したりする。
    - いつかのケース：fail-stop, fail-safe, fail-arbitrary failures
  - 送信側が送ったメッセージは、その順番に受信側に到着する
    - メッセージは、届くことが100%保証できない

# 分散分割ソート：Process Pi

```
Process Pi(my_pid,  pred_id, succ_id, c[1..10]) {
   while(1) {
     local_sort (c[1..10]); 昇順
     Nsend (succ_id, c[10]);
     pid = Brecany()
     if ( pid == pred(my_pid)) { /* 左側から*/
          do_exchange (pid, c[1]);
    }else if (pid == succ(my_pid)) { /* 右側から  */
         do_exchange (pid, c[10]);
     } /* 終了判定 */

?
   }
}
```



# python3による演習: thread

```
#
# A thread creationg by hxt
#
import time
import threading

def proc(id):
    for i in range(2):
        time.sleep(1)
        print("thread",id,"count",i)
#
# main
#
if __name__ == '__main__':
    threadlist = list()
    for i in range(2):
        t = threading.Thread(target=proc, args=(i, ))
        threadlist.append(t)
        t.start()

    print(threadlist)
    for thread in threadlist:
        thread.join()

    print("All thread is ended.")
```

fork

# Vring: thread + message queue

```
#main
if __name__ == '__main__':
    # 3つのqueueを作成
    mq = [queue.Queue()]*3
    threadlist = list()
    for i in range(3):
        t = threading.Thread(target=proc, args=(i, mq[i], mq[(i+1)%3],))
        threadlist.append(t)
        t.start()

    for thread in threadlist:
        thread.join()
    print("main is done.")
```

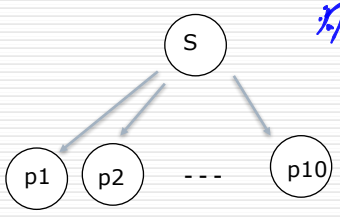
```
def proc(id, pred_q, succ_q):
    logging.debug('start')
        # 最初のthreadでは、メッージをput
    if (id== 0):
        succ_q.put(100) #put(100)
        logging.debug('put')
    time.sleep(1)
    # itemをget
    item = pred_q.get()
    logging.debug('get')
    logging.debug(item)

    # itemをput
    succ_q.put(item)
    logging.debug('put')
    logging.debug('endloop')
```



# 分散サーチ(1)

- Distributed Search for the numbers between 1 and N
- Initial value, 100 cards, N=1,000
- Processes: P1- P10
- Given integer k (1..1000), is it in 100 cards or not?




カードを分割→統合
(join と fork
みたいなの)

# Dsearch: Distributed Search (1)

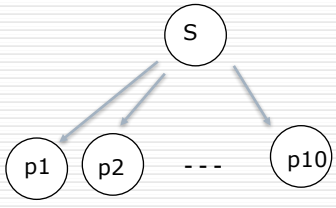
```
#
# main()
#
if __name__ == '__main__':
    #
    # 100枚のcardを作成し、任意の整数(0..1000)を記入
    #
    card = [0 for i in range(100)]
    print("card all 0 clear:",card)
    for i in range (100):
        card[i] = random.randint(1,1000)
    print("card=",card)


    #
    # 検索用の整数を入力
    #
    print("Give me a number (1..1000)?")
    num = int(input())
    print("looking for", num)
    #
    # 10個のスレッドと通信用Queueを生成、起動
    #
    threadlist = list()
resultq = queue.Queue()
for i in range(10):
    t = threading.Thread(target=proc, args=(i, card, num, re
    threadlist.append(t)
    t.start()
 #
#  スレッドの終了を待ち、結果を印刷
#
print(threadlist)
for thread in threadlist:
    thread.join()
    if (resultq.get() == 1):
        print("num=",num,"was found")
        break
print("All thread is ended.")
```



# 分散サーチ(2)

- Distributed Search for the numbers between 1 and N
- Initial value, 100 cards, N=1,000
- Processes: P1- P10
- Given integer k (1..1000), how many cards are with integer k?



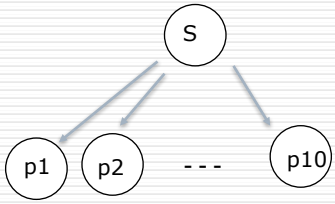
## 課題-3: due 5/27/2019

- python3で書かれたdsearch.pyを拡張し、分散サーチの結果が0 or 1でなく、何件発見できたかを出力するプログラムにせよ。



## 並列処理の限界は？

- 処理スピード P1..P10
  - Fork and Join型
  - Divide and Conquer型
- 一番遅いプロセスがボトルネック



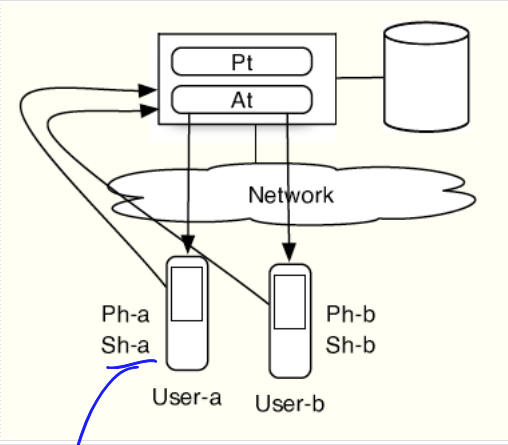
# 演習3:集中型から分散型へ

待ち合わせ支援システム:
A Rendezvous System using Smart Phone



# 設定画面(Setting Imange)

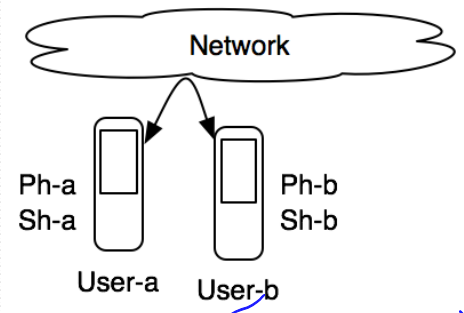
# HOT-SPA Model (Centralized type)






Sensing here.

# HOT-SPA Model (Decentralized type)



## 分散アルゴリズムの評価

どちらのアルゴリズムが優れているか？



## 分散システムの評価指標は？

- 伝統的には、
  - 計算時間（CPU）、メモリ容量(main memory, HD)
  - 消費電力
- 交換された総メッセージ数
  - バッテリー消費量
- リアルタイム vs. ノン・リアルタイム
  - デッドラインを満たすことができるかいなか？

# 自律分散協調システム
# 分散アルゴリズム(1)

NICT/慶應義塾大学名誉教授
徳田英幸



# 分散アルゴリズム
# Distributed Algorithm

## 分散アルゴリズム(1)

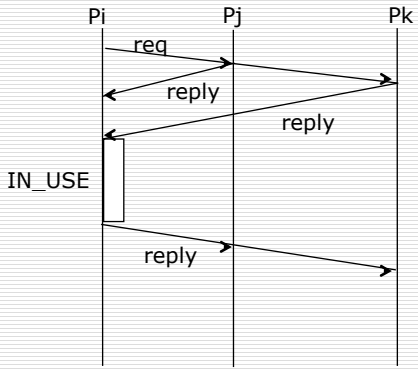
- Distributed Mutual Exclusion／分散相互排除
- Election Algorithm／リーダ選出
- Distributed Deadlock Dection／分散デッドロック検出
- Clock Synchronization／クロック同期
- Consensus Problem／合意
- Byzantine consensus problem／ビザンチ合意
- Distributed Search／分散検索
- Distributed Hash Table／分散ハッシュ
- Distributed Transaction／分散トランザクション
- ・・・



## 分散相互排除アルゴリズム

- Distributed Mutual Exclusion Algorithm
- E.g.
  - ネットワーク上の設置された逐次利用可能なリソースを相互排除して使用する場合
    - Resource Managerがいる場合：リクエストイベントの到着順で順序制御
    - Resource Managerがいない場合：リクセストイベントの出発順で順序制御

# 直感的な例






# Logical Clock

- L. Lamport (1978)
- 分散システムでの事象(event)の順序関係
- happens-before:  a -> b
- If a and b are events in the same process and  a occurs before b then a->b is true.
- If a is "send-event"  of a message in Pi and b is "receive-event" of the message in Pj, then a->b is true.
- For any two events in different processes, x->y is not true and y->x is not true:  x and y are concurrent events

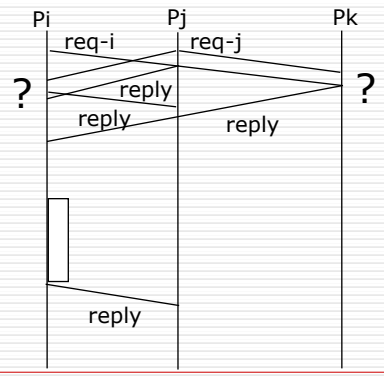
# Lamport's Logical Clock

- Ci：Logical Clock（局所論理時計）
- 規則1：受信事象以外の事象eがPiで起こった場合、その直後にCiに1を加える．事象eは、更新された時刻Ciに生起したとする．
- 規則2：
  - Piがメッセージを送信するときには、時刻印ts=Ciをメッセージに付加して送信する．
  - Piが時刻印tsを持つメッセージを受信したときには、Ciをmax{Ci, ts}+1まで進める．この受信事象は更新された時刻Ciに生起したとする．



# Lamport's Algorithm

- 基本的なアイデア



## Ricart-Agrawala's Algorithm

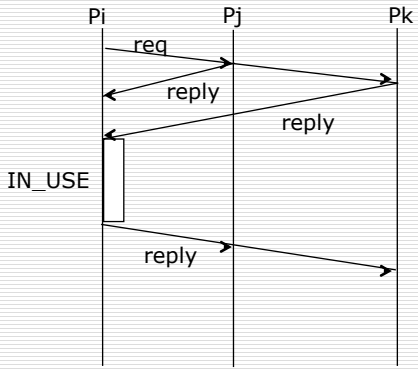
- 1)資源を使用したいプロセスPiはreq(TSi, Pi)を他のすべてのプロセスに送信する.
- 2)プロセスPiからreqメッセージを受信したプロセスはPjは以下のように行動する.
  - Pjが今資源を要請していないならば、Pjの現時点での局所時刻印を持つreplyメッセージをPiに返す.
  - もし要請中であれば、(TSi、Pi)をPjのreqの(TSj、Pj)と比較する.
    - if (TSi, Pi) < (TSj, Pj)then replyメッセージをPiに返す.
    - else req(TSi, Pi)へのreplyをPjが資源を使用し終わるまで保留する.



## Ricart-Agrawala's Algorithm (cont.)

- 3)他のすべてのプロセスから自分のreqメッセージに対するreplyメッセージを受信したときに限り、Piは、資源の使用を許可される.
- 4)Piが資源を解放したときは、replyを保留しているすべてのreqメッセージ(を送信したプロセス)に対して、replyメッセージを返す.

# 直感的な例





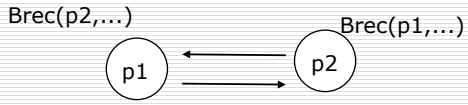

# 分散アルゴリズム
## ～分散デッドロック問題～

並行プロセスの挙動

## デッドロック(1)

```
Process P1
{
  …
  brec(p2, …);
  …
}
```

```
Process P2
{
  …
  brec(p1, …);
  …
}
```

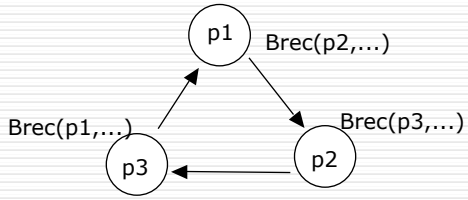





並行プロセスの挙動

## デッドロックとは？

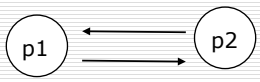
- システムを構成しているすべてのプロセスが起きるはずのない事象を待ち続ける状態



並行プロセスの挙動

## デッドロック（2）

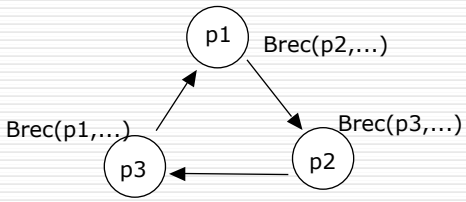
```
Process P1                    Process P2
{                             {
  …                             …
  request(printer,…)            request(tape, …)
  request(tape, …)              request(printer,…)
  …                             …
}                             }
```



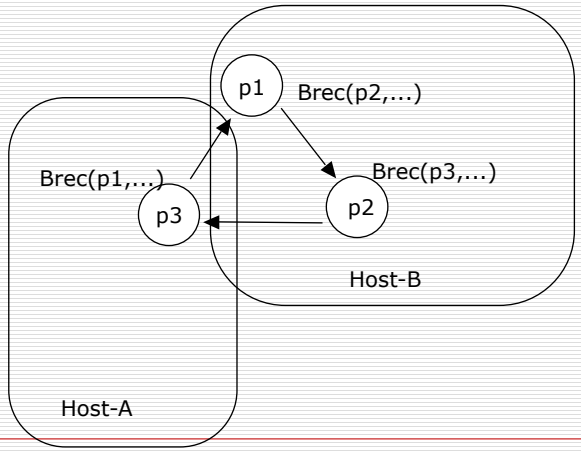



## 分散デッドロック

- システムを構成しているすべてのプロセスが起きるはずのない事象を待ち続ける状態



# WFG(Wait-for-Gragh)






# 分散アルゴリズム
## ～分散トランザクション問題～

## 銀行口座間のお金の移動

- Begin transaction
  - Step1: 普通口座 A = A -1000; （at Ha)
  - Step2: 普通口座 B = B + 1000; (at Hb)
- End transaction
- Question:
  - Step1とStep2の間でHaがダウンするとどうなるか？
  - 100%done or nothing done?
  - どのようにAtomic Propertyを担保するか？



## Transactionの性質

- Transactionが提供する特徴
  - Atomicity（原子性）
  - Consistency (一貫性）
  - Isolation(分離性）
  - Durability(永続性）

## Compensatable Atomic Transaction

- Nested Transaction Model
  - 航空券予約
  - ホテル予約
  - 両方OKの時のみ、旅行を計画する！
- Compensation
- Compensatable Transaction Model



## Nested Transaction

- 入れ子構造のTransaction
- BT
  - // Airlineの予約
  - BT
    - 行きの便の予約
    - 帰りの便の予約
  - ET
  - // Hotelの予約
  - ホテルの予約
- ET