3.分散システムのフォールトトレランス

1. 並行性とトランザクションを学習する
2. 分散システムモデルを学習する
3. 合意問題を学習する
4. 多重化を学習する

1.並行性とトランザクションを学習する

- 並行性 (concurrency)
  - 複数の処理が, 時間的に重なって実行される性質
- 競合状態 (race condition)
  - 並行性によっておこるエラー
- トランザクション (transaction)
  - 不可分な連続した処理の系列
    - 典型的な例. データベース上の処理

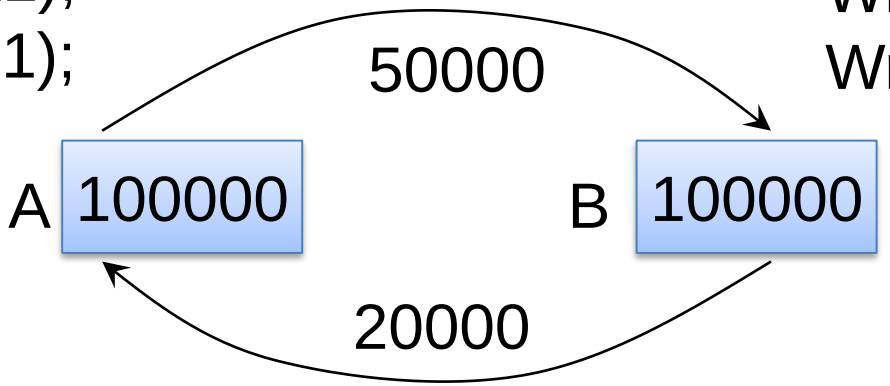
トランザクションの例

T1:

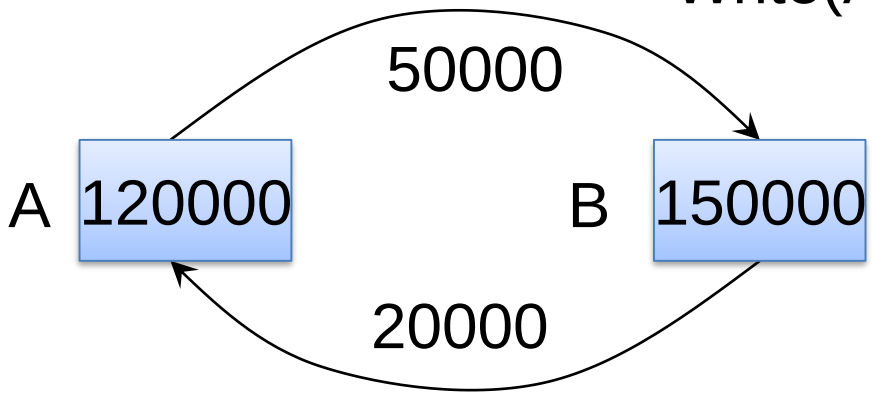
```
Read(A, X1);
Read(B, Y1);
Y1 := Y1 + 50000;
X1 := X1 – 50000;
Write(A, X1);
Write(B, Y1);
```

T2:

```
Read(B, X2);
Read(A, Y2);
Y2 := Y2 + 20000;
X2 := X2 – 20000;
Write(B, X2);
Write(A, Y2);
```



```
Read(A, X1);
Read(B, Y1);
                                    Read(B, X2);
                                    Read(A, Y2);
                                    Y2 := Y2 + 20000;
                                    X2 := X2 – 20000;
                                    Write(B, X2);
Y1 := Y1 + 50000;
X1 := X1 – 50000;
Write(A, X1);
Write(B, Y1);
                                    Write(A, Y2);
```



疑似的にトランザクションを実行するプログラム

```
public class Main {
   static Account A = new Account(100000);
   static Account B = new Account(100000);

   public static void main(String[] args) {
      new Thread1().start();
      new Thread2().start();
   }
}

class Account {
   int value ;
   Account (int x) {
      value = x;
   }
   synchronized void set (int x) {
      value = x;
   }
   synchronized int get()  {
      return value;
   }
}
```

```
class Thread1 extends Thread {
   public void run () {
      int x = Main.A.get();
      int y = Main.B.get();
      //Thread.yield();
      //try {sleep(1000);}catch(Exception e){}
      y += 50000;
      x -= 50000;
      Main.A.set(x);
      Main.B.set(y);
      System.out.println("A:"+ x +", B:" + y);
   }
}

class Thread2 extends Thread {
   public void run () {
      int x = Main.B.get();
      int y = Main.A.get();
      x -= 20000;
      y += 20000;
      Main.B.set(x);
      Main.A.set(y);
      System.out.println("A:"+ y + ", B:"+ x);
   }
}
```

問題点

- データの読込と更新との間に，他のトランザクションの読込，更新が行われる
  - ➡ 競合状態
- 読込と更新が不可分に（原子的に，アトミックに）実行できれば十分か？

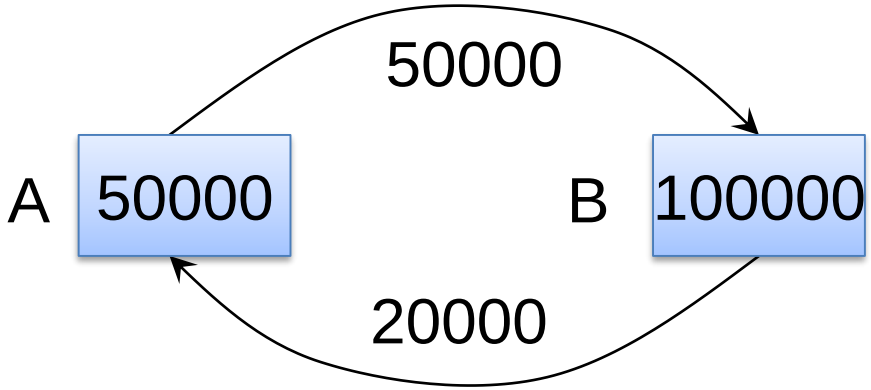
読込と更新とが原子的に実行できる場合

A + B = 200000

A := A – 50000;
B := B + 50000;

A + B = 150000
A + B = 200000

B := B – 20000;
A := A + 20000;



トランザクション

- 永続的なデータに対する不可分な一連の処理
  - PostgreSQLの例

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
WHERE name = 'Alice';
UPDATE accounts SET balance = balance + 100.00
WHERE name = 'Bob';
COMMIT;
```

ACID

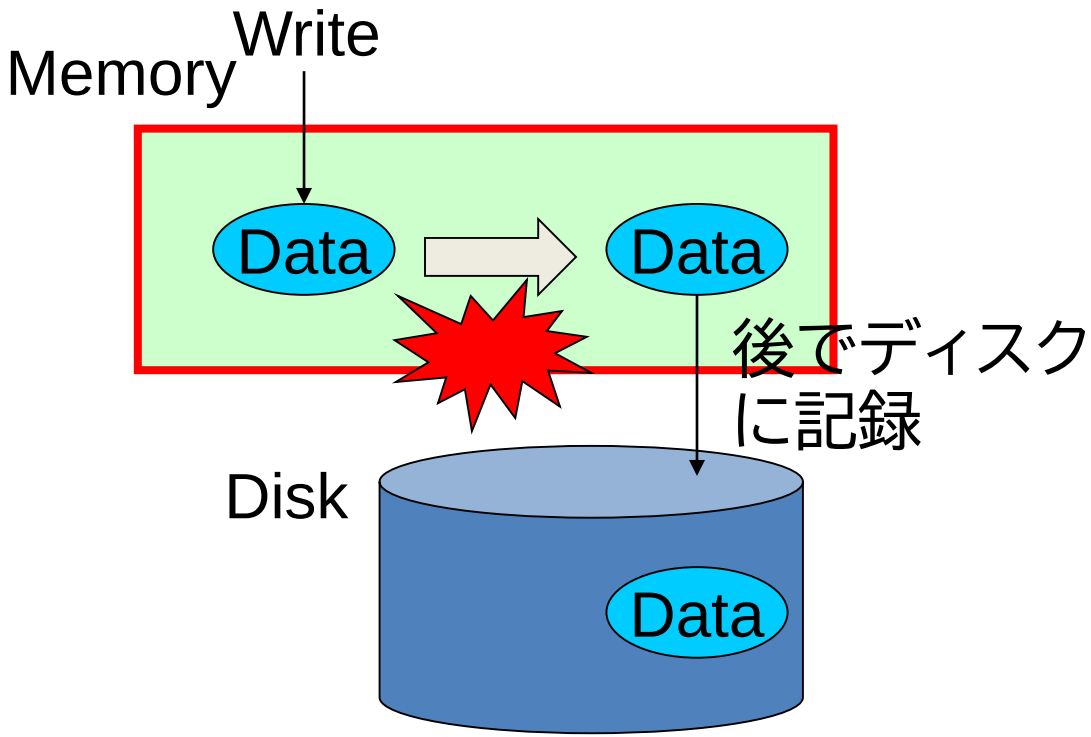
- 原子性 Atomicity
  - トランザクションの最後は, コミットかアボート
    - ◆コミット Commit: データ操作をデータベースに反映
    - ◆アボート Abort: データ操作を取り消し
- 一貫性 Consistency
- 分離性 Isolation ≒ 直列化可能性 Serializability
  - 複数のトランザクションが同時に実行された場合に, 一貫性が保たれること. 直列化可能性は, 逐次実行された場合と同じであること.
- 持続性 Durability = Fault tolerance
  - 単一ノードでの故障への対処
  - 分散システムでの故障への対処

NoSQL データベース

- 関係データベース管理システム (RDBMS) 以外のデータベース管理システム
  - MongoDB, Cassandra, Memchached,..
  - トランザクションを利用できないことが多い
- BASE (Basically Available, Soft State, Eventually Consistent)
  - ACIDより緩い性質
    - ◆アベイラビリティ優先
    - ◆結果整合性

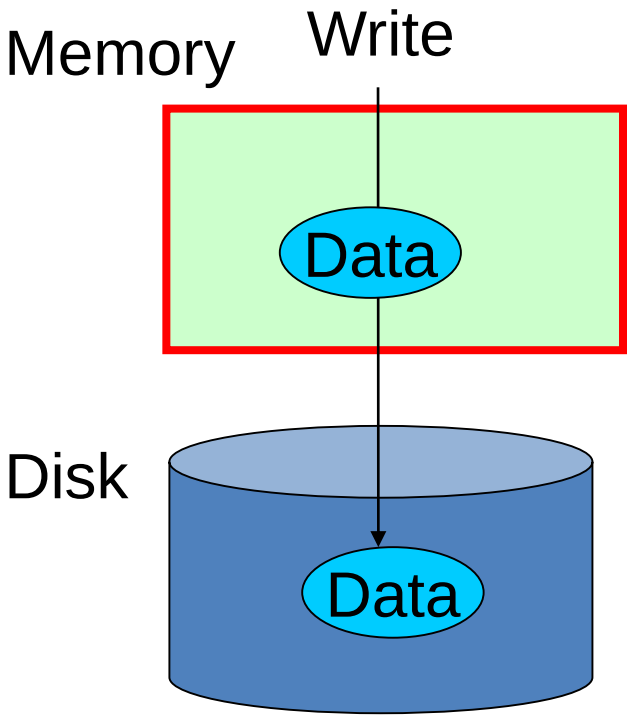
単一ノードでの故障への対処

Write-back
キャッシュを後で記録



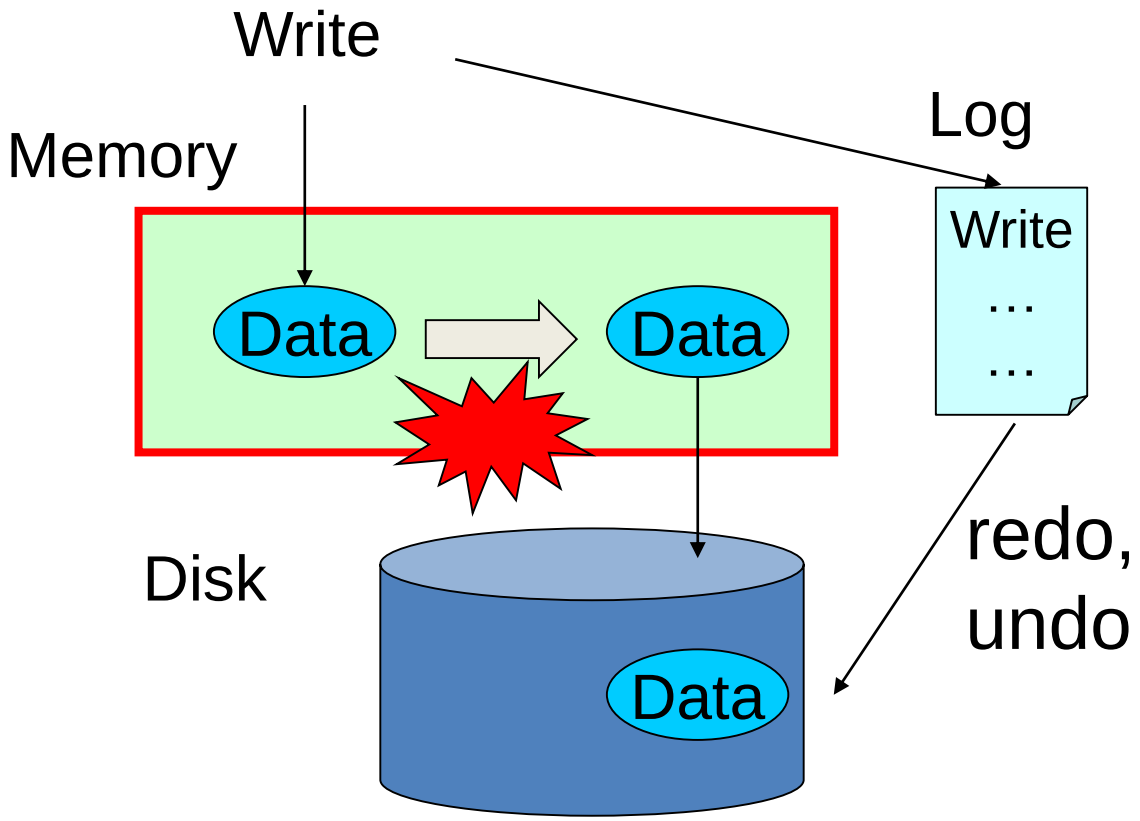
Fault → 不整合の可能性

Write-through
キャッシュとディスクに書込



低性能

ログの利用



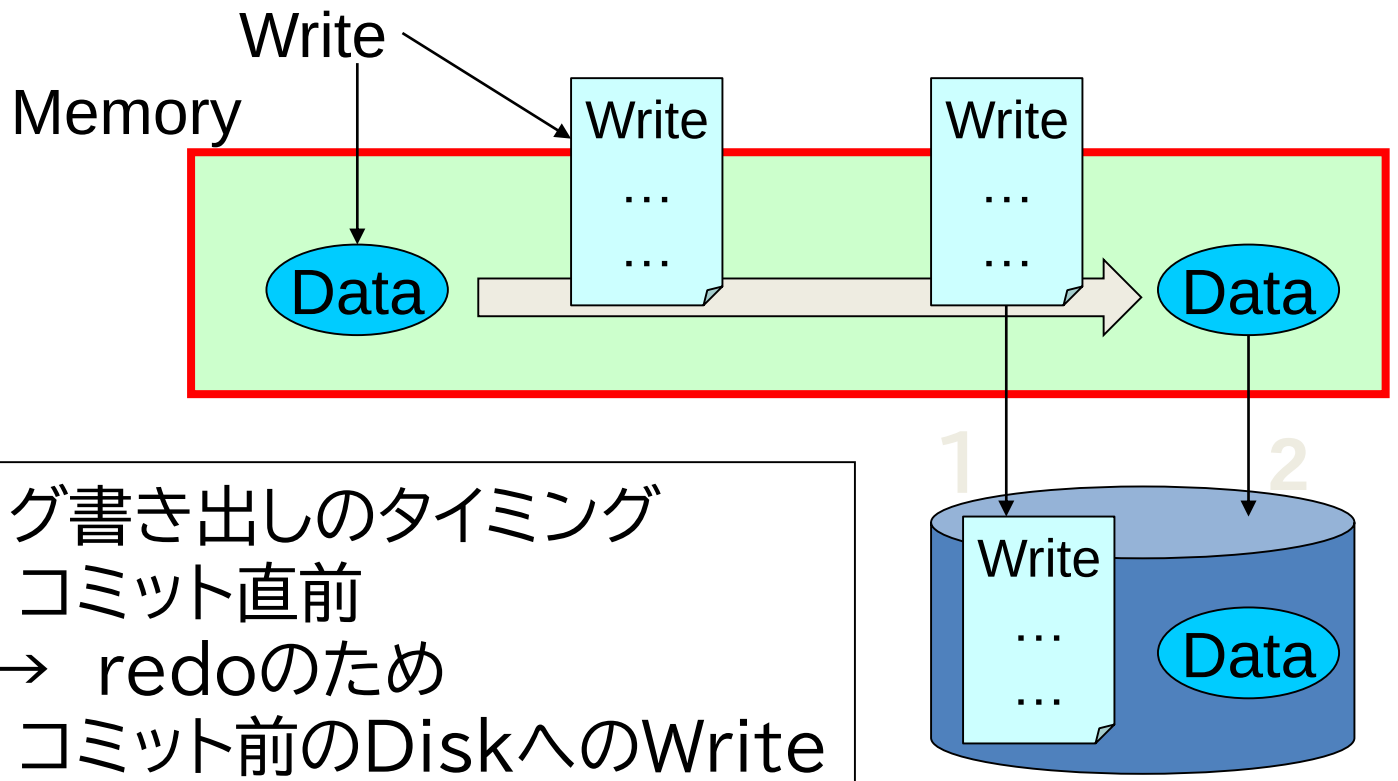
故障時の対応

- トランザクションがコミットされていたとき
  →リドゥ（redo）＝ 更新（write）をディスクに固定化
- コミットされていなかったとき
  →アンドゥ（undo）＝ 無効化

問題
ログが主記憶に保存された場合，ログが消えてしまう

ログ先行書き込み (WAL; Write Ahead Logging)

- ログをディスクに書いてから，ディスク上のファイルを更新



ログ書き出しのタイミング
- コミット直前
  → redoのため
- コミット前のDiskへのWrite
  → undoのため

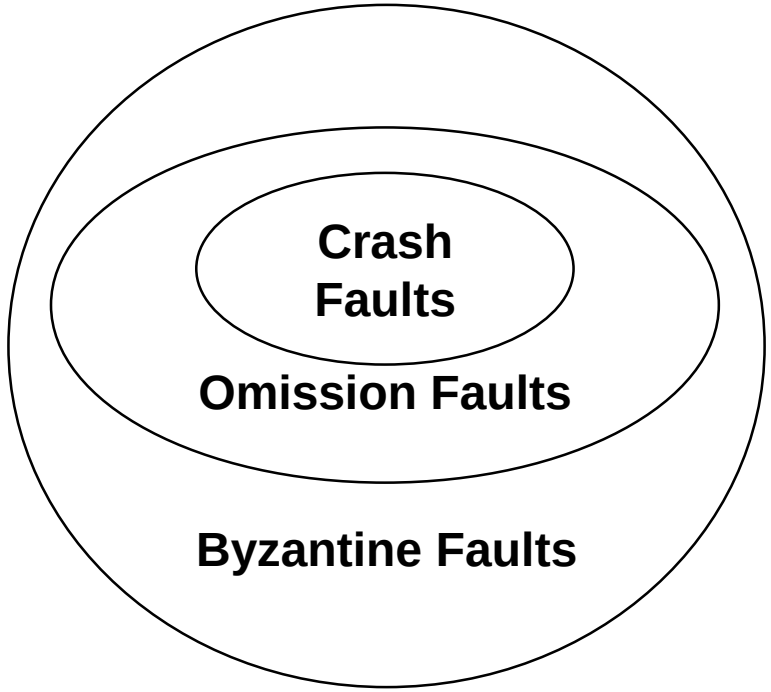
ファイルシステムでは，
ジャーナリング
（journaling）という

2. 分散システムモデルを学習する

- 分散システム (distributed system)
  - ネットワークを通してメッセージパッシング (message passing)により通信する複数のプロセスから構成
- プロセス (process)
  - 独立して実行されている計算の実態
  - 以下とほぼ同義
    - プロセッサ (processor)
    - ノード (node)
    - サイト (site)

分散システムにおける故障モデル

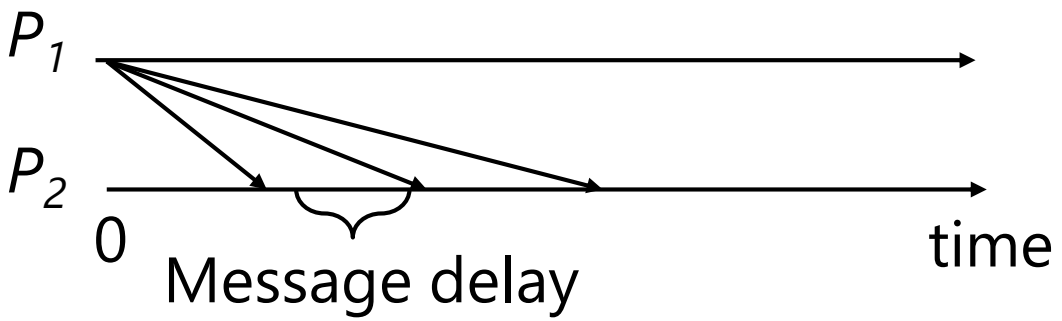
- クラッシュ故障 (Crash Fault)
  - 故障したプロセスは停止
- オミッション故障 (Omission Fault)
  - メッセージの送信, 受信の失敗
    ＋クラッシュ故障
- ビザンチン故障 (Byzantine Fault)
  - 任意の動作



- これらは通信の故障をプロセスの故障としてモデル化
- プロセスと通信と, 別々に故障モデルを考える場合もある

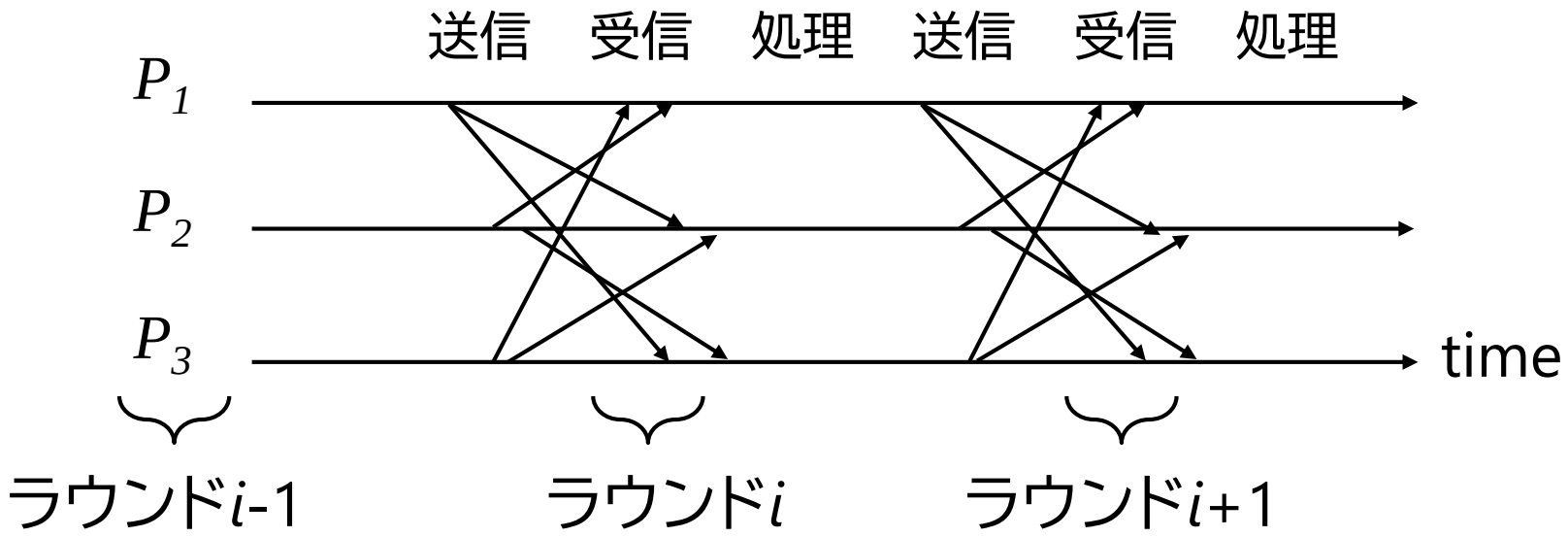
同期性 (synchrony)

- 同期性
  - 異なるメッセージの時間的な性質が均質であること
- 同期システム (synchronous system)
  - メッセージ遅延に上限が存在し, 既知
- 非同期システム (asynchronous system)
  - メッセージ遅延に上限が存在しない
    ➡遅いメッセージと故障を区別できない



同期ラウンドモデル (Synchronous round model)

- プロセスが同期してステップを実行するモデル
  - 同期システムで実現可能
  - アルゴリズムの設計・証明が容易



3.合意問題 (Agreement Problems) を学習する

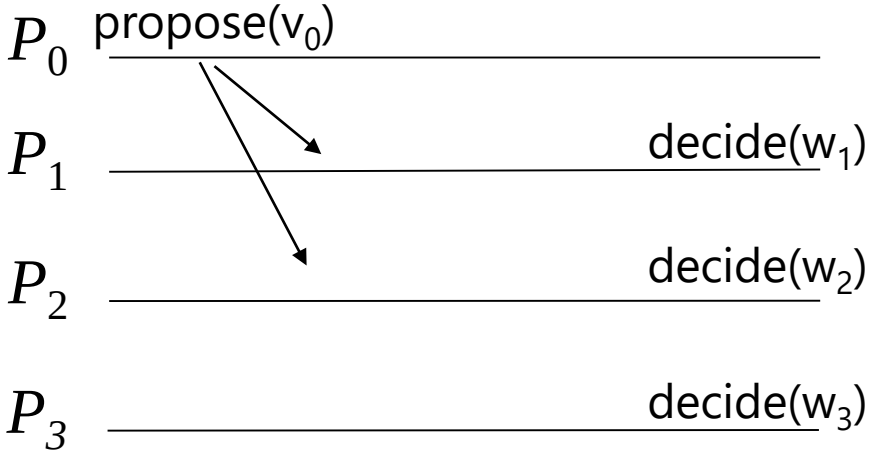
- 代表的な問題
  - ビザンチン将軍 (Byzantine generals)
  - ブロードキャスト (broadcast)
  - コンセンサス (consensus)
- 応用
  - リプリケーション (replication)
  - トランザクション (transaction)

ビザンチン将軍問題 (Byzantine generals problem)

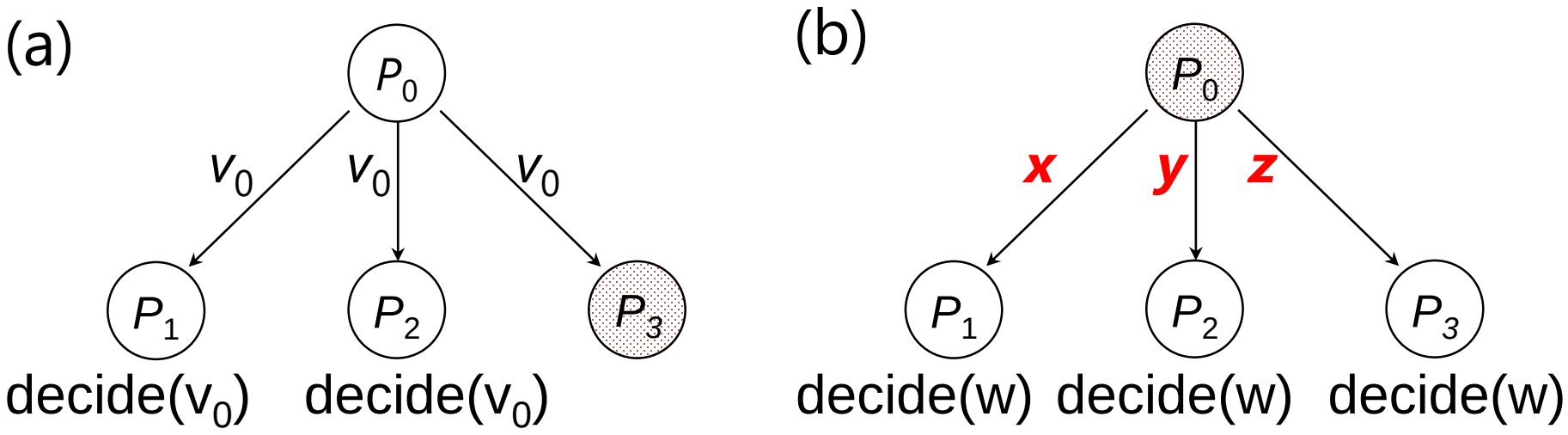
- システムモデル
  - 同期システム＋同期ラウンドモデル
  - ビザンチン故障
- プロセス
  - P_0: メッセージを送るプロセス
  - $P_1, P_2, \ldots, P_{n-1}$: メッセージを受け取るプロセス

Conditions

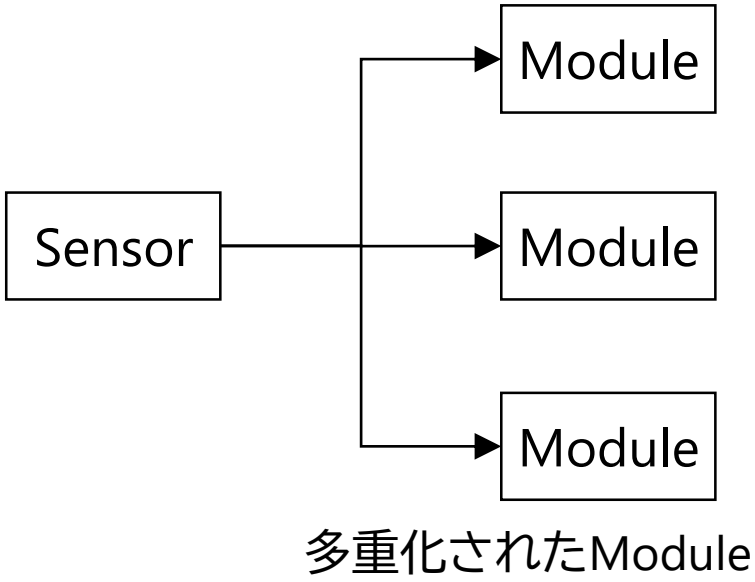
- 停止性 termination
  - P_i (i =1,..,n-1)は, 正常なら, いずれdecideする
- 合意 agreement
  - 正常なP_i, P_j (i, j =1,.., n-1)がw_i, w_jをdecideしたとき, $w_i = w_j$
- 妥当性 validity
  - 正常なP_0がv_0をproposeしたとき, 正常なP_i (i=1,..)はv_0をdecideする



- 正しい動作例



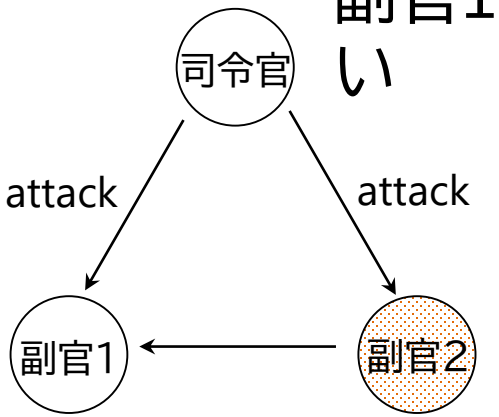
- 応用例
  センサの故障に
  耐えられる多重系



Impossibility result

- $n < 3k+1$の場合, アルゴリズムは存在しない
  - k: 故障したプロセス数の上限
- 例. $n = 3, k = 1$
  - P_0: 司令官 (commander)
  - P_i $(i = 1, 2)$: 副官 (lieutenant)
  - 提案値v_0: 攻撃(attack)か退却(retreat)

副官1には区別できない

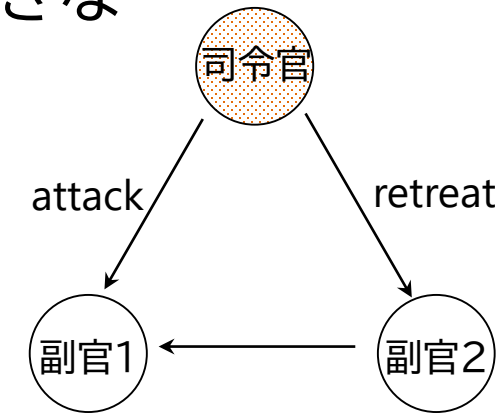


decide(attack)

妥当性の条件より

He said `retreat.'

⇔

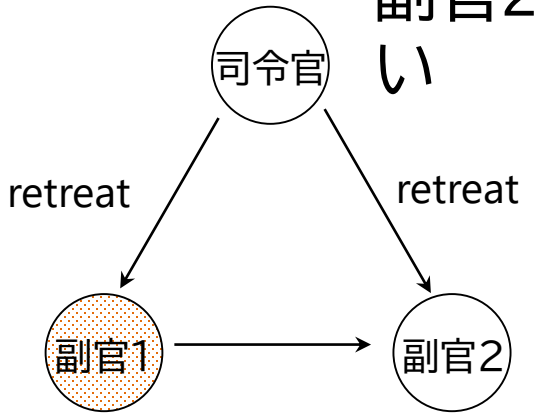


decide(attack)

He said `retreat.'

同じ状況

副官2には区別できない

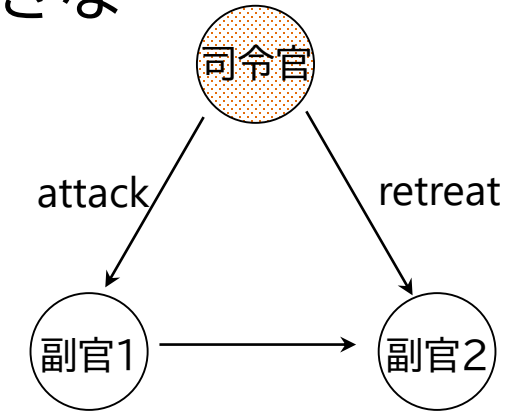


He said `attack.'

decide(retreat)

妥当性の条件より

⇔

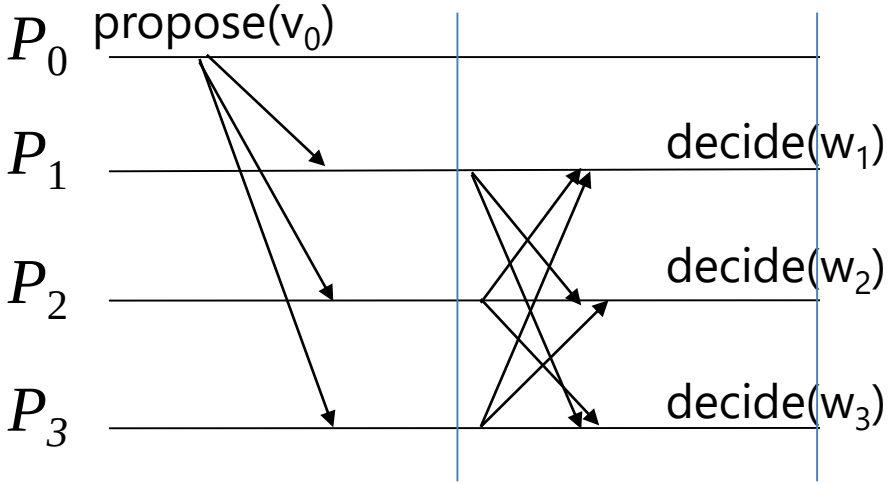


He said `attack.'

decide(retreat)

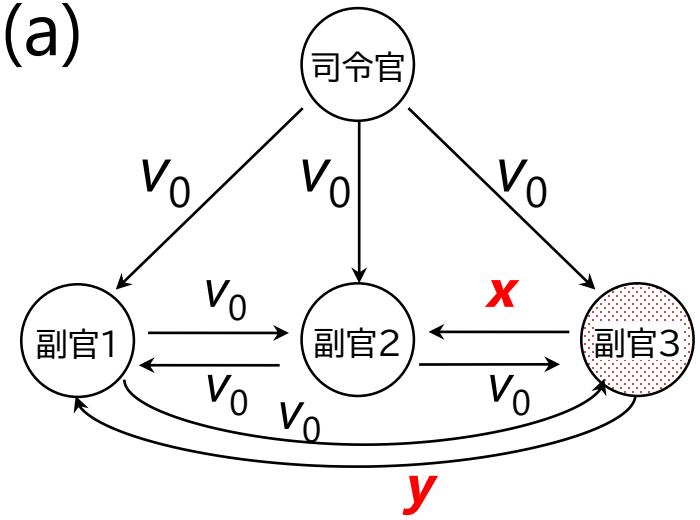
Algorithm OM (n = 4, k =1の場合）

1. 副官は司令官からうけとった値を, 他の2名の副官に送信
2. 受け取った3値のうち, 過半数をしめるものを選択
   そのような値がなかった場合は, デフォルト値⊥を選択

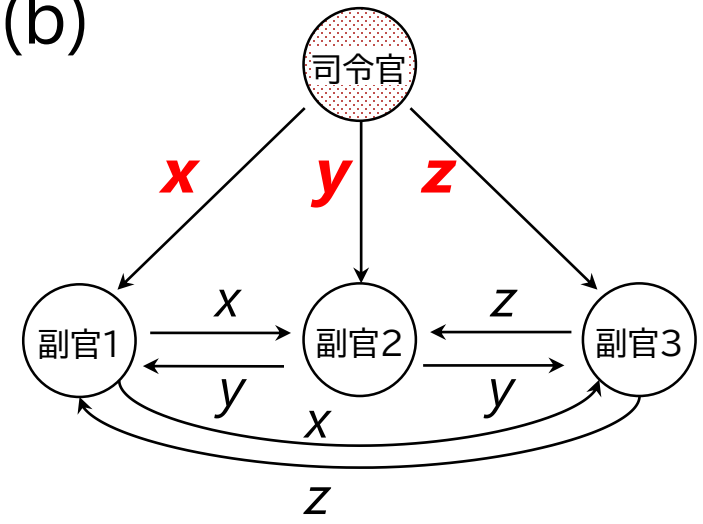


例. n=4, k=1の場合

(a)



(b)



	副官1	副官2	副官3
受信値集合	v_0, v_0, y	(ア)	(イ)
決定値	v_0	v_0	(ウ)

副官1	副官2	副官3
x, y, z	x, y, z	x, y, z
⊥	(エ)	(オ)

ブロードキャスト(broadcast)

- ブロードキャスト
  - 全プロセスに同じメッセージを送ること
- 高信頼ブロードキャスト (reliable broadcast)
  - クラッシュ故障の場合
    - ◆正常なプロセスpがmをブロードキャスト
      → pはmをdeliver (受容)
    - ◆プロセスqがmをdeliver
      →すべての正常なプロセスがmをdeliver

ビザンチン将軍問題は，高信頼ブロードキャスト問題の1種

アルゴリズムの例: Eager reliable broadcast

- ブロードキャストの実行
  1. mをすべてのノードに送信
  2. mをdeliver
- mを受信
  - はじめて受信した場合
    1. mをすべてのノードに送信
    2. mをdeliver