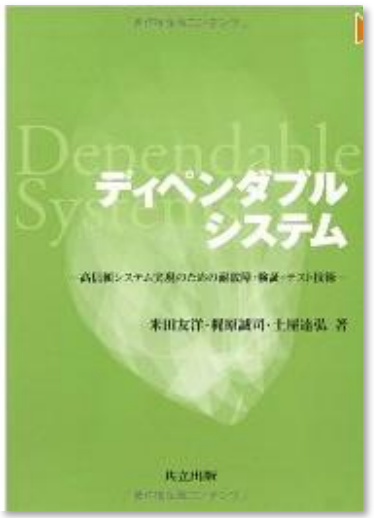
先端システム工学

土屋達弘(大阪大学) 2021.8.18

自己紹介

- 氏名：土屋達弘
- 所属：大阪大学
- 研究テーマ：ディペンダブルシステム

- 耐故障システム
- 信頼性評価
- 自動検証
- テスト
  - 組み合わせテストツール CIT-BACH



テーマ

- SATとその周辺
  - 論理式を「解く」
  - パズル
- プログラムの検証
  - 有界モデル検査
    - SATの応用
- 分散アルゴリズムの検証

SATとその周辺

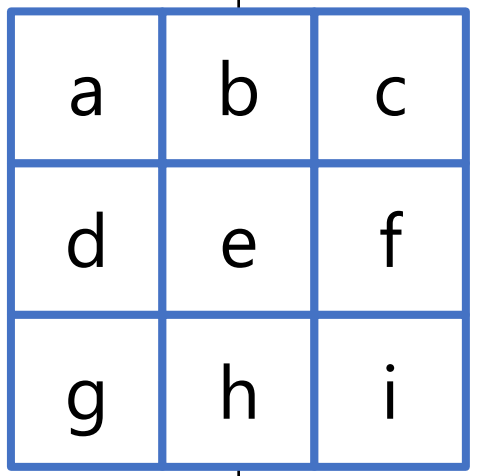
- 論理式を解くという問題
  - 論理式：値がブール値であるような式
- SAT（充足可能性問題）
  - ブール式が真にできるか
- Satisfiability Modulo Theories (SMT)
  - SATを拡張
    - 実数, 整数, 関数などを扱えるように
- 制約充足問題 (Constraint Satisfaction Problem, CSP)
  - 複数の制約条件を満たす解を求める問題
  - SMTで表せることが多い

SMTソルバ

- SMTを解くツール
  - 代表例：Z3 (Microsoft)
  - オンラインでも利用可能
    - https://jfmc.github.io/z3-play/
    - https://compsys-tools.ens-lyon.fr/z3/

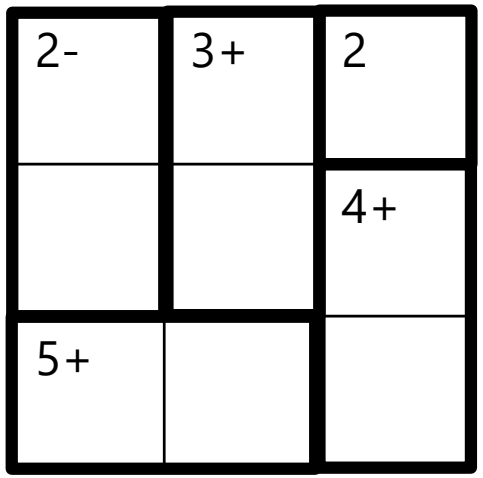
例. 魔方陣

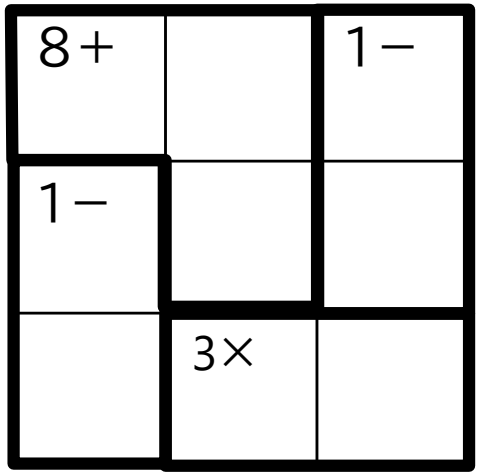
```
; ;の後はコメント
; 変数を宣言
(declare-const a Int)
...
; 1~9まで出現
(assert (or (= a 1) (= b 1) (= c 1) (= d 1) (= e 1) (= f 1) (= g 1) (= h 1) (= i 1))
...
; 1行目の値の和をsumに設定
(declare-const sum Int)

; 行例の値がすべてsumに一致
(assert (= (+ a b c) sum))
...

; 解があるかしらべる
(check-sat)
; 解を表示
(get-model)
```



賢くなるパズル
Kenken

- マスに整数を割当
  - 1～行(列)の数
- 指定された演算で
  指定された値に一致
- 行と列において，
  数はすべて異なる





問題の難しさ

- この問題はどれくらい難しい？
- 難しさの議論は，判定問題を中心に行うことが多い
  - 判定問題
    - YesかNoかでこたえる問題
  - すべての問題例の中でのワーストケースについて考える
- 代表的なクラス
  - P：効率のよいアルゴリズムが存在するクラス
  - 決定可能：アルゴリズムが存在するクラス
    - 答えがでる
  - 決定不能：アルゴリズムが存在しないクラス

充足可能性問題 (satisfiability problem, SAT)

- 充足可能性問題
  - 入力：ブール式（Boolean formula)
    - ブール変数 (値は真(1, True)か偽(0, False))
    - 論理演算子：¬ (否定), ∧ (論理積), ∨ (論理和)
    - 例. $x \wedge (y \vee \neg x) \wedge (\neg y \vee \neg z)$
  - 出力：ブール式を真にすることが可能ならYes, そうでないならNo
    - 例. Yes (変数への値割当の例. $x \leftarrow$ 真, $y \leftarrow$ 真, $z \leftarrow$ 偽)

SATの問題例

- 例1. $x \wedge (y \vee \neg x) \wedge (\neg y \vee \neg z)$
  - Yes (真理値割り当ての例. $x \leftarrow 1, y \leftarrow 1, z \leftarrow 0$)
- 例2. $(x \vee y) \wedge (\neg(x \vee y) \vee (\neg z \vee \neg y))$
  - Yes
- 例3. $(x \vee \neg y) \wedge (y \vee z) \wedge (\neg x \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (x \vee y \vee \neg z)$
  - No

SATの難しさ

- NP完全
  - クック - レビンの定理
- 完全の意味→ NPでもっとも難しい問題のクラス
  - NP：Yesに対する答えがあれば, その答えの正しさは効率よく確認できる問題のクラス
- 答えを求める効率のよいアルゴリズムは存在しないと予想されている

「P≠NP問題というのは当然知っているよな」湯川が後ろから声を掛けてきた。

石神は振り返った。

「数学の問題に対し、自分で考えて答を出すのと、他人から聞いた答えが正しいかどうかを確認するのとでは、どちらが簡単か。あるいはその難しさの度合いはどの程度か――クレイ数学研究所が賞金をかけて出している問題の一つだ」

引用：東野圭吾, 容疑者Xの献身, 文藝春秋, 2005

テーマ2:プログラムの検証

- SATとその周辺
  - 論理式を「解く」
  - パズル
- プログラムの検証
  - 有界モデル検査
    - SATの応用
- 分散アルゴリズムの検証

モデル検査 (Model checking)

- モデル検査
  - コンピュータによる状態探索で，プログラムやシステムが与えられた性質を満たすかを検証する手法
- 有界モデル検査
  - 検査するシステムの要素の数や動作の長さを限定して，検証を行うモデル検査手法

コードに対する有界モデル検査

- コードを制約に変換し，充足解を探索

```
assume(x + y > 2);
if (b)
  x = x +1;
else
  x = x + 2;
y = y + x;
x = y + 1;
assert(x > 5);
```

$$
\begin{aligned}
&\wedge\ y0 + x0 > 2 \\
&\wedge \\
&\quad \vee\, b0 \wedge x1 = x0 + 1 \\
&\quad \vee\, \neg b0 \wedge x1 = x0 \\
&\wedge \\
&\quad \vee\, \neg b0 \wedge x2 = x1 + 2 \\
&\quad \vee\, b0 \wedge x2 = x1 \\
&\wedge\ y1 = y0 + x2 \\
&\wedge\ x3 = y1 + 1 \\
&\wedge \neg(x3 > 5)
\end{aligned}
$$

解があれば，assert違反の動作が存在

コードに対する有界モデル検査

- 制約の充足解の探索
  - SATソルバ, SMTソルバを利用

$$
\begin{aligned}
&\wedge\ y0 + x0 > 2 \\
&\wedge \\
&\quad \vee b0 \wedge x1 = x0 + 1 \\
&\quad \vee \neg b0 \wedge x1 = x0 \\
&\wedge \\
&\quad \vee \neg b0 \wedge x2 = x1 + 2 \\
&\quad \vee b0 \wedge x2 = x1 \\
&\wedge\ y1 = y0 + x2 \\
&\wedge\ x3 = y1 + 1 \\
&\wedge \neg(x3 > 5)
\end{aligned}
$$

➡

Z3への入力例

```
(declare-const b0 Bool)
(declare-const x0 Int)
(declare-const x1 Int)
(declare-const x2 Int)
(declare-const x3 Int)
(declare-const y0 Int)
(declare-const y1 Int)

(assert (> (+ y0 x0) 2))
(assert (or
  (and b0 (= x1 (+ x0 1)))
  (and (not b0) (= x1 x0))
))
(assert (or
  (and (not b0) (= x2 (+ x1 2)))
  (and b0 (= x2 x1))
))
(assert (= y1 (+ y0 x2)))
(assert (= x3 (+ y1 1)))
(assert (not (> x3 5)))
(check-sat)
(get-model)
```

ソルバの実行結果

```
sat
(model
  (define-fun b0 () Bool
    true)
  (define-fun y0 () Int
    0)
  (define-fun x2 () Int
    4)
  (define-fun x1 () Int
    4)
  (define-fun x0 () Int
    3)
  (define-fun x3 () Int
    5)
  (define-fun y1 () Int
    4)
)
```

```
assume(x + y > 2);
if (b)
  x = x +1;
else
  x = x + 2;
y = y + x;
x = y + 1;
assert(x > 5);
```

CBMC

- 有界モデル検査ツール
  - C言語を検証
  - https://www.cprover.org/cbmc/

- 使い方
  - cbmc ファイル名 [-trace]

前の例

```c
#include <stdio.h>
#include <assert.h>

int main (void) {
    int x, y, b;
    scanf("%d %d %d", &x, &y, &b);

//     __CPROVER_assume((x + y > 2));
    if (x + y > 2) {
        if (b)
            x = x +1;
        else
            x = x + 2;
        y = y + x;
        x = y + 1;
        assert(x > 5);
    }
}
```

パズル

- パズルを解くこともできる
- 例. 魔方陣

```c
#include <stdio.h>
#include <assert.h>
#include <stdbool.h>

#define SIZE 3

int main(void) {
    int entry[SIZE][SIZE];
    bool allValues = true;
    bool validColumns = true;
    bool validRows = true;

    ...

    assert(!(allValues && validColumns && validRows));
}
```

別の例

```c
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

int main () {
    int x, y;
    scanf("%d %d", &x, &y);
    if (x > y) {
        int tmp;
        tmp = x - y;
        y = x;
        x = tmp;
    }
    printf("x:%d y:%d¥n", x, y);
    assert(x <= y);
}
```

思わぬバグ

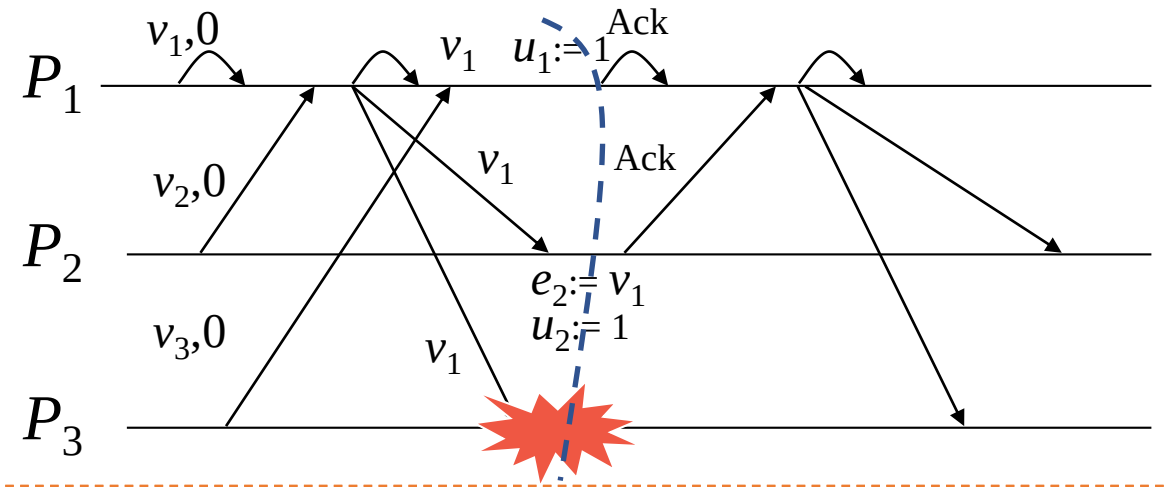
- **404 Blog Not Found**
- https://dankogai.livedoor.blog/archives/50522708.html

テーマ3:分散アルゴリズムの検証

- SATとその周辺
  - 論理式を「解く」
  - パズル
- プログラムの検証
  - 有界モデル検査
    - SATの応用
- 分散アルゴリズムの検証

分散アルゴリズム

- 分散アルゴリズム
  - ネットワーク上の複数のプロセスを協調させる
  - 特徴
    - 非同期メッセージ通信
    - プロセス故障, メッセージ消失への対応



分散アルゴリズム開発の問題点

- 実装が非常に手間
  - プログラムを書いてテストすることはまれ
- 正しい設計が困難
  - 実装・運用している場合でも, バグがあることが多い
    - Chord
      - 分散ハッシュテーブル
    - Zyzzvya, FaB
      - Byzantineコンセンサス

研究の目的と提案アプローチ

- 目的
  - アルゴリズムを容易に検証・テストできる手法の開発

- アプローチ

1. 分散アルゴリズムを通常の逐次プログラムとして記述
   - 具体的には, Cプログラム
2. 逐次プログラムに対する既存の方法でテスト・検証
   - SATを用いた方法. 具体的には, 有界モデル検査
     - ツール CBMC

逐次プログラムによる分散アルゴリズムの表現

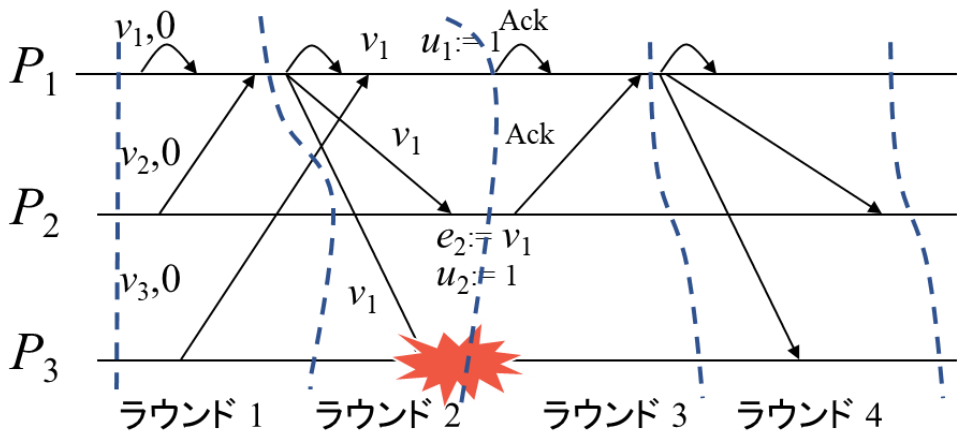
- どうやって？
- プログラムが表現すべき分散アルゴリズムの特徴
  - プロセスの非同期性
  - メッセージの喪失
  - メッセージの遅延
  - プロセスの故障

- アルゴリズムにCommunication-closednessを仮定

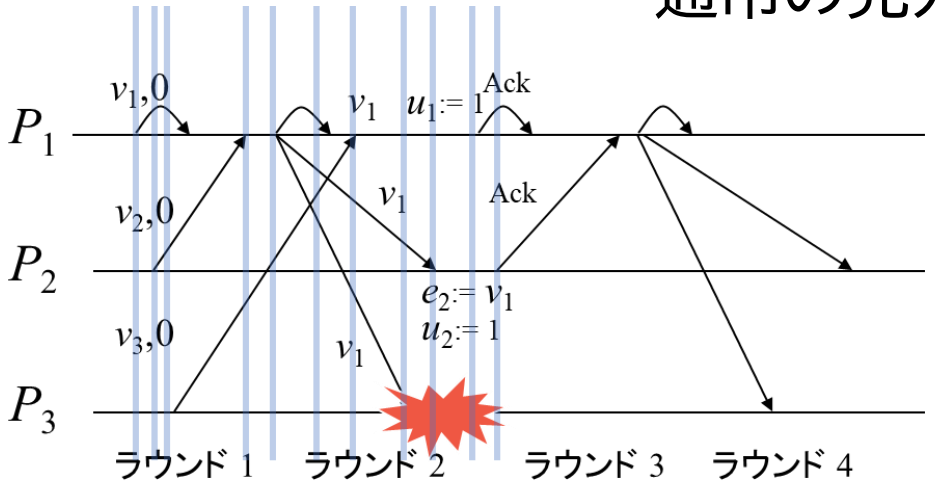
Communication-closedness (1/2)

- あるラウンドで送信されたメッセージは, そのラウンドでしか受信(delivery)されないという性質
- 動作をラウンド毎にとらえることが可能
  ↔ 通常の見方：プロセスの状態遷移が, イベント毎に発生

動作をラウンド毎に把握

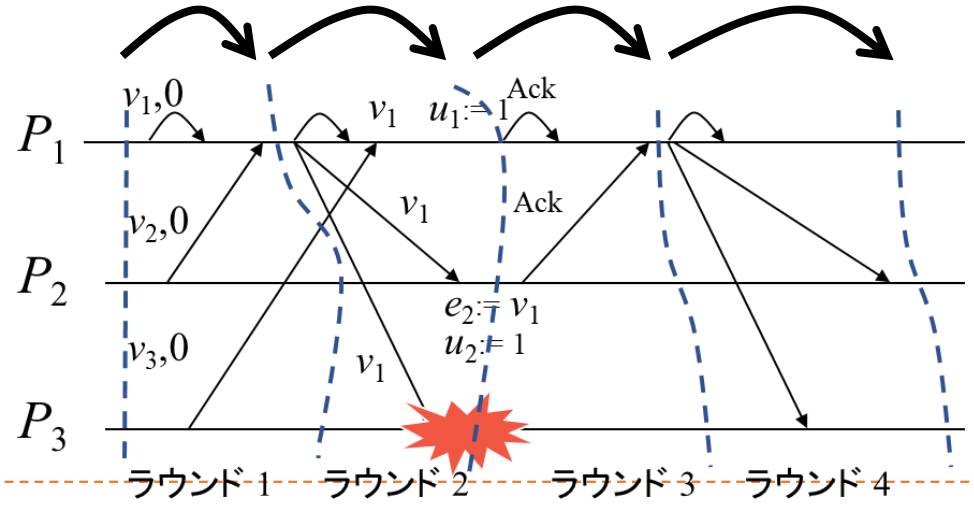


通常の見方



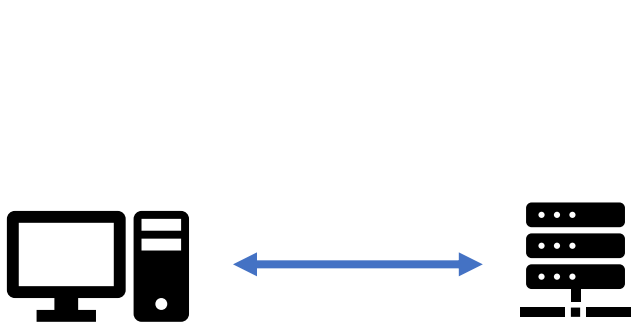
Communication-closedness (2/2)

- 動作をラウンド毎にとらえる → 逐次的な動作として表現可能
  - プロセスの非同期性 → ラウンド毎の同期的な状態遷移
  - メッセージの喪失　→ ランダムに喪失を表現
  - メッセージの遅延　→ 喪失と同一視
  - プロセスの故障　　→ メッセージ喪失で表現

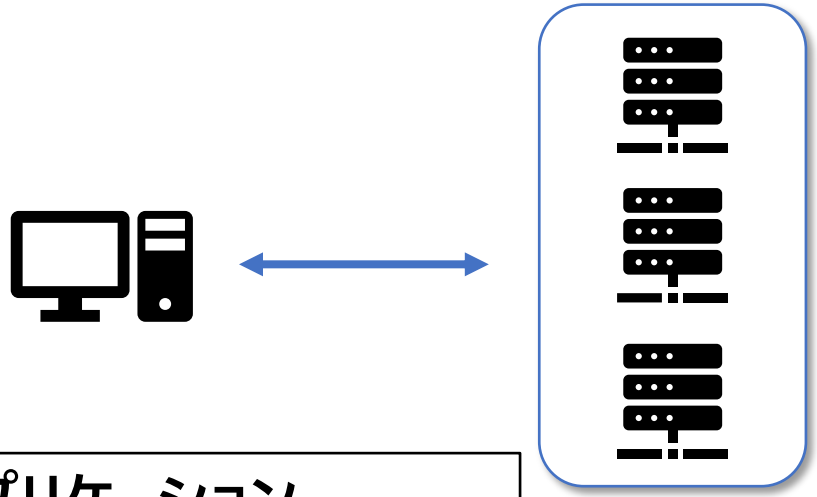


検証対象：
レプリケーションのためのコンセンサスアルゴリズム

- レプリケーション
  - 複数のサーバを同期させることで故障耐性を実現
  - ディペンダブルな分散システムを実現するコア技術



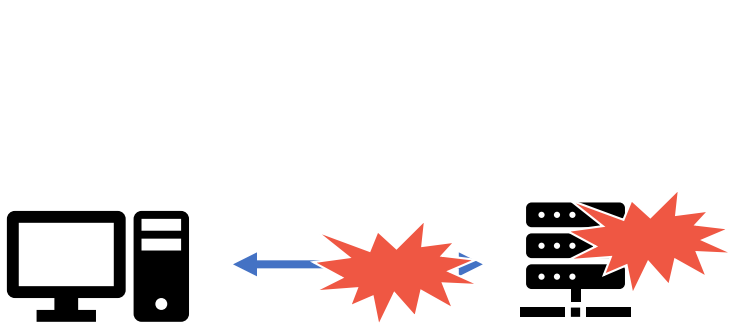
レプリケーションなし
→ サーバが単一障害点



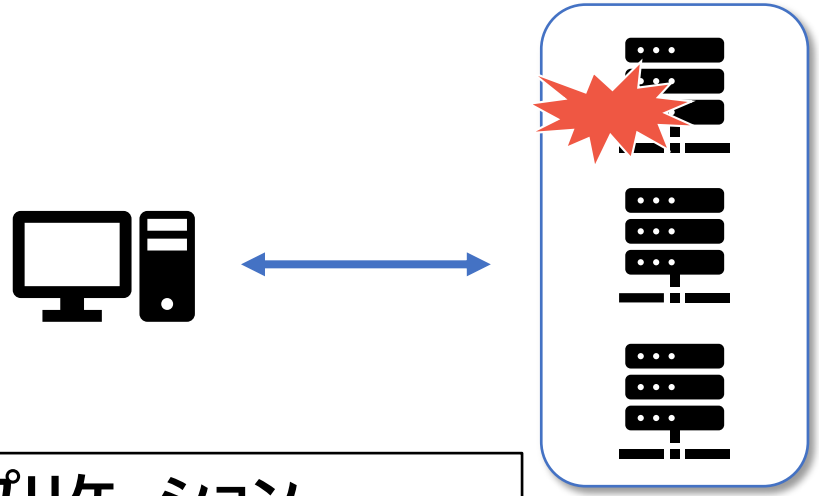
レプリケーション
→ 障害に対する耐性

分散システムにおけるレプリケーション

- レプリケーション
  - 複数のサーバを同期させることで故障耐性を実現
  - ディペンダブルな分散システムを実現するコア技術



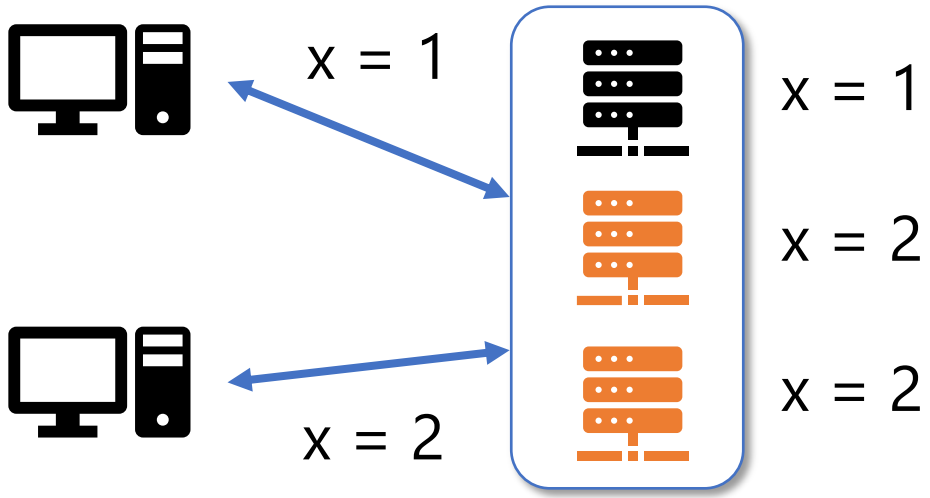
レプリケーションなし
→ サーバが単一障害点



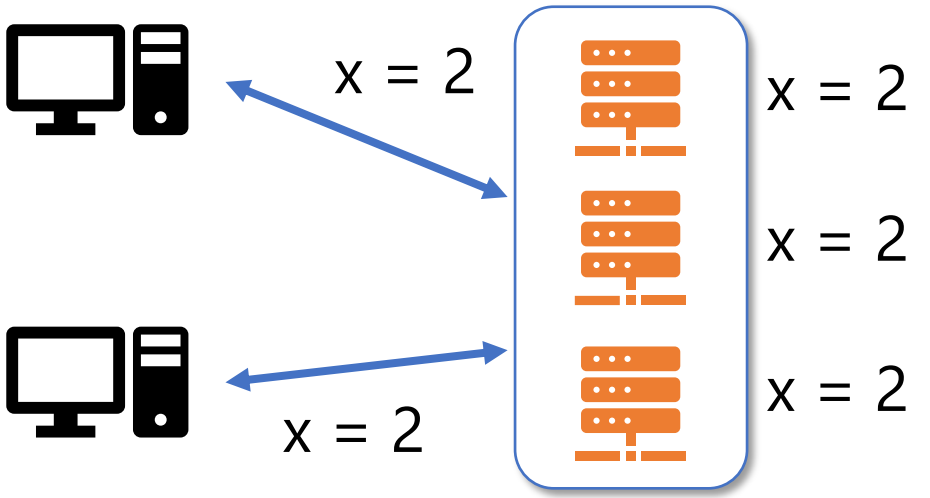
レプリケーション
→ 障害に対する耐性

コンセンサス

- レプリケーションの基礎
- 複数のサーバの状態を一致させる問題



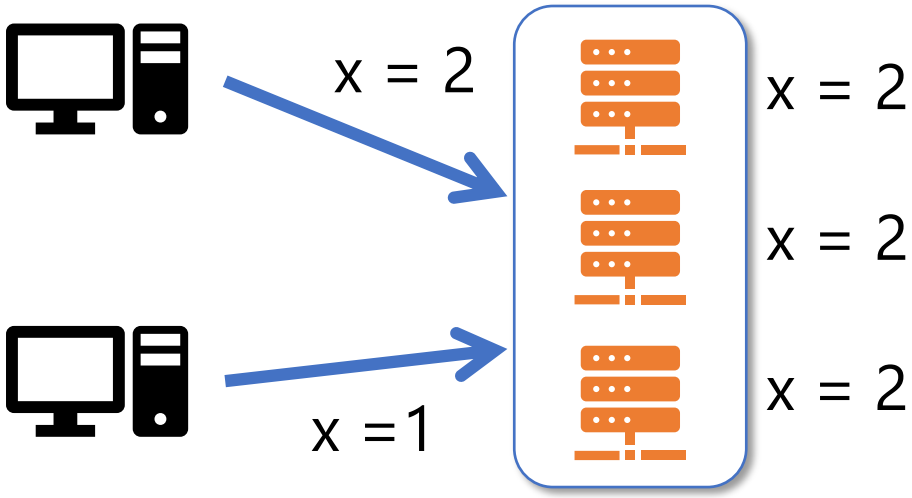
状態が不一致
→ データの不整合



状態が一致
→ 1つのサーバのように動作

コンセンサス

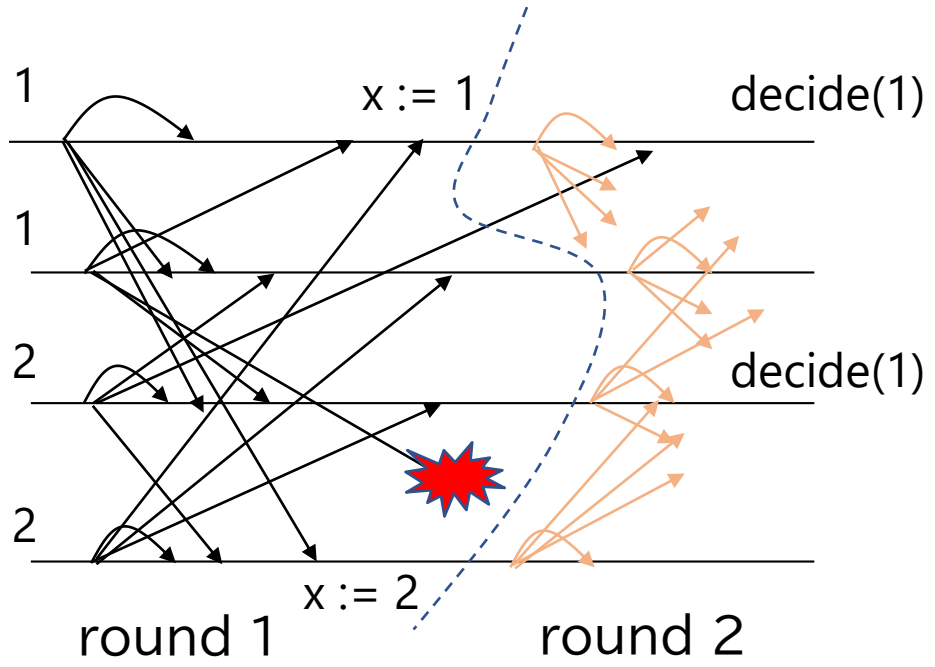
- すべてのプロセスが値を提案
- すべてのプロセスが値を決定
- 満たすべき性質:合意性
  - 決定した値はすべて等しい



簡単なアルゴリズム: One-third-rule

Initialization
　x := 提案値

Round r
　xを全プロセスに送信
　if 受信したメッセージ数 > 2n/3
　　x := 最も受信数の多い値で
　　　最も小さいもの
　if 受信した値の >2n/3 が一致
　　xで決定



逐次プログラムとしての表現 (1/3)

Initialization
　x := 提案値

Round r
　xを全プロセスに送信
　if 受信したメッセージ数 > 2n/3
　　x := 最も受信数の多い値で
　　　最も小さいもの
　if 受信した値の >2n/3 が一致
　　xで決定

```
int main(int argc, char **argv)
{
    int x[NUM_PROC];

    // Set proposed values randomly
    for (int proc = 0; proc < NUM_PROC; proc++)
    {
        x[proc] = _randint(1, NUM_PROC);
    }

    // Iterate rounds MAX_ROUND times
    for (int round = 1; round <= MAX_ROUND; round++)
    {
        ...
```

提案値をランダムに決定

_randint(a, b):aからbまでの乱数を発生

逐次プログラムとしての表現 (2/3)

Initialization
　x := 提案値

Round r
　xを全プロセスに送信
　if 受信したメッセージ数 > 2n/3
　　x := 最も受信数の多い値で
　　　最も小さいもの
　if 受信した値の >2n/3 が一致
　　xで決定

_randint(a, b):aからbまでの乱数を発生

```
for (int round = 1; round <= MAX_ROUND; round++)
{
    // channels
    int chnl[NUM_PROC][NUM_PROC];
    // send
    for (int proc = 0; proc < NUM_PROC; proc++)
    {
        for (int peer = 0; peer < NUM_PROC; peer++)
        {
            chnl[peer][proc] = x[proc];
        }
    }
    // delay or lost
    for (int proc = 0; proc < NUM_PROC; proc++)
    {
        int *ch = chnl[proc];
        for (int i = 0; i < NUM_PROC; i++)
        {
            if (_randint(0, 1) == 0)
            {
                ch[i] = -1;
            }
        }
    }
        ...
```

メッセージをランダムに消失

逐次プログラムとしての表現 (3/3)

Initialization
　x := 提案値

Round r
　xを全プロセスに送信
　if 受信したメッセージ数 > 2n/3
　　x := 最も受信数の多い値で
　　　最も小さいもの
　if 受信した値の >2n/3 が一致
　　xで決定

```
int decision[NUM_PROC];
int firstdecision = 0;

void decide(int proc, int value)
{
    decision[proc] = value;
    if (firstdecision == 0)
    {
        firstdecision = value;
    }
    else
    {
        assert(firstdecision == value);
    }
}
```

assertで合意性を検証

- 満たすべき性質：合意性
  - 決定した値はすべて等しい

乱数のあつかい

- プログラム作成時
  - 通常の疑似乱数
  - シミュレーションによりプログラムをデバッグ
- アルゴリズム検証時
  - SATの対象となるブール式上で，変数として記号的に扱う
  - 任意の値での動作を検証

```
int _randint(int first, int last)
{
    int tmp;
#if defined(CBMC)
    __CPROVER_assume((first <= tmp) && (tmp <= last));
#else
    tmp = rand() % (last + 1 - first) + first;
#endif
    return tmp;
}
```

実験

- 3種のアルゴリズムに適用
  1. LastVoting[2]
  2. One-third-rule[2]
  3. Hybrid-1[1]
- パラメータ
  - ラウンド 6まで
  - n: プロセス数

アルゴリズム	n	時間 (秒)	変数	節
#1	3	0.6	62090	134663
#1	4	10.9	105687	239655
#1	5	30.5	162626	382387
#1	6	175.9	233739	566903
#1	7	T.O.	320534	799323
#2	3	0.8	52347	130997
#2	4	3.1	94117	258220
#2	5	15.3	151511	443769
#2	6	45.0	226496	693855
#2	7	93.7	320965	1025337
#2	8	267.7	436938	1445066
#2	9	T.O.	576622	1964361
#3	3	1.5	74288	195122
#3	4	9.7	131283	372779
#3	5	2.3 (SAT)	208560	628468

[1] Bernadette Charron-Bost, André Schiper: PRDC 2006.
[2] Bernadette Charron-Bost, André Schiper: Distributed Computing, 2009.

検証結果

- Hybrid-1のバグを検出
  - プロセス数n=5のときに式が充足
  - assertion違反
    - 合意性が満たされない

アルゴリズム	n	時間 (秒)	変数	節
#1	3	0.6	62090	134663
#1	4	10.9	105687	239655
#1	5	30.5	162626	382387
#1	6	175.9	233739	566903
#1	7	T.O.	320534	799323
#2	3	0.8	52347	130997
#2	4	3.1	94117	258220
#2	5	15.3	151511	443769
#2	6	45.0	226496	693855
#2	7	93.7	320965	1025337
#2	8	267.7	436938	1445066
#2	9	T.O.	576622	1964361
#3	3	1.5	74288	195122
#3	4	9.7	131283	372779
#3	5	2.3 (SAT)	208560	628468

研究のまとめ

- 分散アルゴリズムのテスト・検証手法を提案
  1. アルゴリズムを通常のCプログラムとして記述
     - 状態遷移をラウンド毎に考える
  2. 有界モデル検査で検証
     - 一定のラウンドの動作をブール式に変換
     - SATでassertion違反を検出
- いくつかのコンセンサスアルゴリズムに適用
  - アルゴリズムのバグを検出
- 今後の課題
  - コンセンサス以外のアルゴリズムへの適用

まとめ

- SATとその周辺
  - 論理式を「解く」
  - パズル
- プログラムの検証
  - 有界モデル検査
    - SATの応用
- 分散アルゴリズムの検証