

# Raft on Weekend

**Riku Mochizuki (moz)**

moz [at] sfc.keio.ac.jp

# はじめに

Delightの夏合宿用のRaftレポジトリです。  
このレポジトリではRaftの実装を行います。  
座学と実装はこのスライドに沿って行います。

## 日程と内容

日程と各種内容です。

進捗状況によって、行う内容は変更します。

各Docsは最低限の説明のみ記載しています。

わからないことがあれば適宜メンターに相談するか、自分で調べて実装を進めることにする。

# 1日目

Raftの理解/実装に必要な知識の習得

- イントロダクション(0-intro.md)
- 並列処理入門(1-parallel-processing.md)
- 一貫性モデル入門(2-consistency.md)
- Raftの説明(3-raft.md)

## 2日目以降

Pythonを用いたRaftの実装

- このレポジトリのファイル構成について(4-repo.md)
- ハンズオンの説明(5-hands-on.md)

# 問い合わせ

moz [at] sfc.sfc.ac.jp

# 並列・並行処理基礎

Riku Mochizuki

moz at sfc.keio.ac.jp

# 並行/並列処理基礎

このセクションでは主要なライブラリについて解説します。

- 並列/並行処理とは？ 何が嬉しいの？
- Pythonで並列処理 並列処理の種類と違いは？
- コルーチンとNon-blocking I/O "マルチスレッドとの違いは？
- Pythonでコルーチンの実装 並行処理とは？
- マルチスレッド処理とBlocking I/O 並列処理と並行処理の違いは？
- スレッドセーフ 並列処理を安全に実装するためには？
- アムダールの法則 並列化によってどれくらいプログラムが高速になるのか？
- コルーチン(スレッド)間での値の受け渡し

# 並行/並列処理基礎

## 並行/並列処理

並行処理(concurrency computation)とは、計算機のリソースを効率的に利用することで計算速度を速めることを指します。

一方、並列処理(parallel computation)とは、複数の計算を同時に実行することを指します。

並行処理と並列処理は似たような字面ですが、概念や実装は相当に異なります。

## 逐次処理

並行/並列処理の逆は逐次処理(serial computation)です。

逐次処理は、タスクを一つずつ順番に処理する方法です。

逐次処理では、各タスクが完了するまで次のタスクが開始されません。

そのため、タスクの実行時間はそれぞれのタスクの時間の合計となり、並列処理と比較して処理速度が低下することが多いです。(必ずしも早くなるわけではないことをあとで説明します。)

# Pythonで並行/並列処理

# Pythonで並行/並列処理

今回のhandsonでは[asyncio](#)を用います。

asyncioはPythonで並行（一部並列、並行と並列の違いはあとで説明）を実現するためのライブラリです。

asyncioはシングルスレッド内で並行処理を行っています。

並行/並列処理と聞くと、一般にマルチスレッドやマルチプロセスを想像すると思います。

しかし、**コルーチン**という仕組みを使うとシングルスレッド内で並行処理が可能です。

コルーチンについてはあとで説明します。

# Pythonで並行/並列処理

Pythonでは、ほかにも以下の方法で並列/並行処理を実現できます：

- `threading.Thread` : スレッドを使用して並列処理を行います。
- `multiprocessing.Process` : プロセスを使用して並列処理を行います。
- `concurrent.futures` : 高レベルのインターフェースを提供し、スレッドまたはプロセスプールを使用して並列処理を簡単に実装できます。
- `asyncio` : 非同期I/O操作を効率的に処理するためのライブラリで、シングルスレッド内での並行処理を実現します。

これらの方法を使用することで、Pythonで効率的な並列処理を実現することができます。

# コルーチンとNon-blocking I/O (並行処理)

## コルーチンとは？

シングルスレッド内で並行処理を実現するためにはコルーチンという概念を理解する必要があります。

コルーチンはよくマルチスレッド処理と比較されます。以下は簡単な比較例です。

## コルーチンの特徴

- シングルスレッド: コルーチンはシングルスレッド内で実行され、イベントループによって管理されます。
- 非同期I/Oに最適: コルーチンはI/Oバタスク（ネットワーク操作やファイル操作など）に最適です。
- 軽量: スレッドに比べて軽量で、コンテキストスイッチのオーバーヘッドが少ないです。
- 協調的マルチタスク: コルーチンは明示的にawaitを使用して他のタスクに明示的に制御を渡します。

## マルチスレッドの特徴

- マルチスレッド: マルチスレッドは複数のスレッドを使用して並列にタスクを実行します。
- CPUバウンドのタスクに適: マルチスレッドはCPUバウンドのタスク（計算集約型の処理など）に適しています。
- 重い: スレッドはコルーチンに比べて重く、コンテキストスイッチのオーバーヘッドが大きいです。
- プリエンプティブマルチタスク: スレッドはOSによってスケジューリングされ、明示的に制御を渡す必要はありません。

## コルーチンの状態推移

コルーチンを理解するためには、まずコルーチン（または一般的なプロセス）には3つの状態があることを理解する必要があります。

- **実行中(Running)**...処理がスレッドに割り当てられ、実行されている状態です。
- **ブロック中(Blocking)**...ファイルのreadやnetworkの待ち状態など、次の処理を待っている状態です。処理がスレッドに乗っていない状態です。
- **実行待ち(Pending)**...Block中で待っていた処理が届き、次の処理がいつでも実行可能な状態。

通常は、実行中->ブロック中->実行待ち->実行中->...とプロセスが終了するまで無限にループします。

## コルーチンの状態推移

さて、コルーチンにも同じことが言えます。例えば以下のコードを見てみます。

```
import time

def read_file(n):
    print('読み込みスタート:', n)
    time.sleep(1) # 何か重いファイルを読み込んでいると仮定
    print('読み込み終了:', n)

read_file(1)
read_file(2)
read_file(3)
```

## コルーチンの状態推移

この場合、合計で3秒かかります。

しかし、ファイル読み込みの場合、実行スレッドはSSDやハードディスクからデータが転送されるのを待っている状態がほとんどです。

したがって、実行スレッドはほぼ3秒間ブロック中(Blocking)です。このようなブロックを**I/O blocking**と言います。

## Non-blocking I/Oとコルーチン

ここでブロック中(I/O blocking)中に違う処理ができたら嬉しいですか?  
そこでコルーチンという概念があります。

コルーチンを使うとスレッドをブロックしない代わりに、違う処理（コルーチン）を実行することができます。

このような概念を\*\*並行処理（非同期処理）\*\*と言います。

そして並行処理を実現するコアな仕組みが**Non-blocking I/O**です。

言葉で説明してもわかりづらいので、実際コードを見ながら理解していきましょう。

# Pythonでコルーチンの実装

# Pythonでコルーチンの実装

コルーチンは、`asyncio` の非同期タスクの実行単位です。

コルーチンは、`async` キーワードを使用して定義され、`await` キーワードを使用して他のコルーチンにスレッドの制御権利を渡すことができます。

コルーチンの基本的な使い方は以下の通りです：

# コルーチンを定義する

```
import asyncio

async def say_hello():
    print('Hello')
    await asyncio.sleep(1)
    print('World')

# コルーチンをイベントループ（シングルスレッド）で実行
asyncio.run(say_hello())
```

この例では、say\_hello コルーチンが定義され、await asyncio.sleep(1) によって1秒間待機します。この間、スレッドは他のコルーチンを実行することができます。

つぎにコルーチンを用いて並行処理を実行してみます。



```
import asyncio

async def func1():
    print('func1() started')
    await asyncio.sleep(1)
    print('func1() finished')

async def func2():
    print('func2() started')
    await asyncio.sleep(1)
    print('func2() finished')

async def main():
    task1 = asyncio.create_task(func1())
    task2 = asyncio.create_task(func2())
    await task1
    await task2

asyncio.run(main())
```

## コルーチンを定義する `async/await`

実行してみると1秒で処理が終了します。

このようにfunc1, func2をコルーチンとして包み、並行処理を行っています。

そして `task = asyncio.create_task()` を呼び出すことで、コルーチンオブジェクトを生成し `await task` を呼び出すことで、コルーチンを実行します。

## コルーチンを定義する `async/await`

`await`の目的は、処理が終了するまで、ほかのタスク（コルーチン）に実行権利（制御権利）を移す、ということです。この場合、`await asyncio.sleep(1)` で1秒処理を止める、ということを明示的に記述しています。言い換えればこれは1秒間ほかのコルーチンにスレッドの制御を譲るということを意味します。

ファイルからデータを読み込む処理を `await readLine()` とします。

この場合、ディスクからファイルのデータを読み込むためI/O blockingが発生します。コルーチンでは、このI/O blocking中にほかのコルーチンにスレッドの制御権利を移します。

## 制御権の明示的譲渡

`await asyncio.sleep(1)` に戻りましょう。

似たような関数として `time.sleep(1)` があります。

これも一秒間処理を停止する処理ですがコルーチン用ではありません。  
したがってほかのコルーチンに制御権利を渡しません。

## 制御権の明示的譲渡

`await/async` で制御権利をコントロールすることが大切です。

このようにコルーチンの状況（コンテキスト）に応じて次に実行するコルーチンを切り替えることを並行処理と言います。（似た言葉に並列処理がありますが全然違います）

# マルチスレッド処理とBlocking I/O (並列処理)

## コルーチンの限界

コルーチンは並行処理に利用できます。ですが、所詮一つのスレッドをコルーチン間で共有して使っています。

そのため、コルーチンが制御権を譲るような設計ができるない場合は、逐次処理と結果は変わりません。

まず以下のコードを実行してみよう



```
import asyncio
import concurrent.futures
import time

def func1():
    print("func1() started")
    time.sleep(1)
    print("func1() finished")

def func2():
    print("func2() started")
    time.sleep(1)
    print("func2() finished")

async def main():
    task1 = asyncio.create_task(func1())
    task2 = asyncio.create_task(func2())
    await task1
    await task2

asyncio.run(main())
```

## コルーチンの限界

実行完了までは2秒かかると思います。

これは `time.sleep(1)` は制御を他のコルーチンに移すのではなく、スレッド全体をブロッキングしてしまうためです。

ここからコルーチンはマルチスレッド処理ではないことも理解できると思います。

## マルチスレッド処理とBlocking I/O (並列処理)

ですが、コルーチンはシングルスレッド内で並行処理を実現する機構です。

ハードウェアがメニーコア and/or 複数のスレッドが利用でいる場合は、新たなスレッドで並列処理を行うことが可能です。

asyncioではコルーチンではなくマルチスレッド処理も可能です。

スレッドプールから利用可能なスレッドを取得し、実行することができます。

まず以下のコードを実行してみよう



```
import asyncio
import concurrent.futures
import time

def func1():
    print("func1() started")
    time.sleep(1)
    print("func1() finished")

def func2():
    print("func2() started")
    time.sleep(1)
    print("func2() finished")

async def main():
    loop = asyncio.get_running_loop()
    with concurrent.futures.ThreadPoolExecutor() as pool:
        task1 = loop.run_in_executor(pool, func1)
        task2 = loop.run_in_executor(pool, func2)
        await task1
        await task2

asyncio.run(main())
```

## マルチスレッド処理とBlocking I/O (並列処理)

このようにスレッドを用いる場合は、Operating Systemが適切にスレッドを割り当て、計算をします。

したがって、制御権利を移すタイミングを明示的(`async/await`)に書かないので！これがコルーチンとの大きな違いです。これは非同期と並行処理の大きな違いでもあります。

# スレッドセーフ

## スレッドセーフ

スレッドセーフとは、あるコードを複数のスレッドが同時並行的に実行しても問題が発生しないことを意味する。

問題とは:

- 一貫性のないデータ取得や更新(Dirty read, Dirty write, Fuzzy read, Phantom read...)
- デッドロック
- (あとなんかありましたっけ?...)

## スレッドセーフ

まずはだめな例から見てみましょう。

以下の例は一貫性のないデータ更新を行う例です。

以下のコードでは10個のスレッドが値を+1する処理です。

なのでcounterの値は10になることが期待されます。



```
import asyncio
import random

class UnThreadSafeCounter:
    def __init__(self):
        self._value = 0

    async def increment(self):
        # スレッドセーフな操作を行うためのロックを使用
        current_value = self._value # 1. 現在の値を読み取る
        # 1~4秒のランダムなスリープ
        await asyncio.sleep(random.randint(1, 4) * 0.1) #割り込みを許す。
        self._value = current_value + 1 # 2. 新しい値を設定する

async def main():
    counter = UnThreadSafeCounter()

    # 非同期タスクを作成してカウンタをインクリメントする
    tasks = [counter.increment() for _ in range(10)]
    await asyncio.gather(*tasks) # すべてのタスクが完了するのを待つ

    print(f"Final counter value: {counter._value}")

# asyncio.run()を使用してコルーチンを実行
asyncio.run(main())
```

## 何がだめなのか？

このコードを実行すると、最終的なカウンタの値が期待値にはなりません（ほとんどの確率で）。これは、incrementメソッドがスレッドセーフでないため、複数のスレッドが同時に値を読み取り、同じ値でインクリメントを行うためです(dirty write)。このような競合が起こることで、期待する結果が得られない場合があります。

## スレッドセーフを導入

続いて良い例です。

複数のスレッドが同時に値を読み取り、値を更新しています問題を解決するためには排他制御をする必要があります。

今回は排他制御の一つであるロックを使います。ロックを"正しく"実装することでスレッドセーフを満たせます。

(正しいとは、、ロックをただむやみにつけるだけではダメです。デッドロックが起こる可能性があり、これではスレッドセーフではありません。)

`asyncio.Lock`は非同期コードで使用するために設計されたロックで、同時に複数のコルーチンがリソースにアクセスしないように制御することができます。



```
import asyncio
import random

class ThreadSafeCounter:
    def __init__(self):
        self._value = 0
        self._lock = asyncio.Lock() # asyncio用のロックを初期化

    async def increment(self):
        # スレッドセーフな操作を行うためのロックを使用
        async with self._lock:
            current_value = self._value # 1. 現在の値を読み取る
            # 1~4秒のランダムなスリープ
            await asyncio.sleep(random.randint(1, 4) * 0.1)
            self._value = current_value + 1 # 2. 新しい値を設定する

    async def main():
        counter = ThreadSafeCounter()

        # 非同期タスクを作成してカウンタをインクリメントする
        tasks = [counter.increment() for _ in range(10)]
        await asyncio.gather(*tasks) # すべてのタスクが完了するのを待つ

        print(f"Final counter value: {counter._value}")

# asyncio.run()を使用してコルーチンを実行
asyncio.run(main())
```

## スレッドセーフ

このコードを実行すると、最終的なカウンタの値は期待どおりの値（10）になります。これは、`asyncio.Lock`を使用して、カウンタのインクリメント操作がスレッドセーフに行われるようとしたためです。

## スレッドセーフと並列不可能な部分

ただし、実行時間がスレッドセーフではないプログラムと比較して、上がってしまったと思います。

これは `sleep` 関数が`lock`によって排他制御されているためです。

あるコルーチンが`lock`を取得すると、ほかのコルーチンは`lock`を取得できるまで`blocking`状態になります。したがってこの場合は逐次処理をした結果と実行速度に変化はありません。

## アムダールの法則

このように並列不可能な部分があるとき、 $N$ 個のプロセス（コルーチン）を使うことで逐次的に処理するアルゴリズムと比べてどれくらい高速化できるかを論理的に示す式があります。それがアムダールの法則です。

$$S = \frac{1}{(1 - P) + \frac{P}{N}}$$

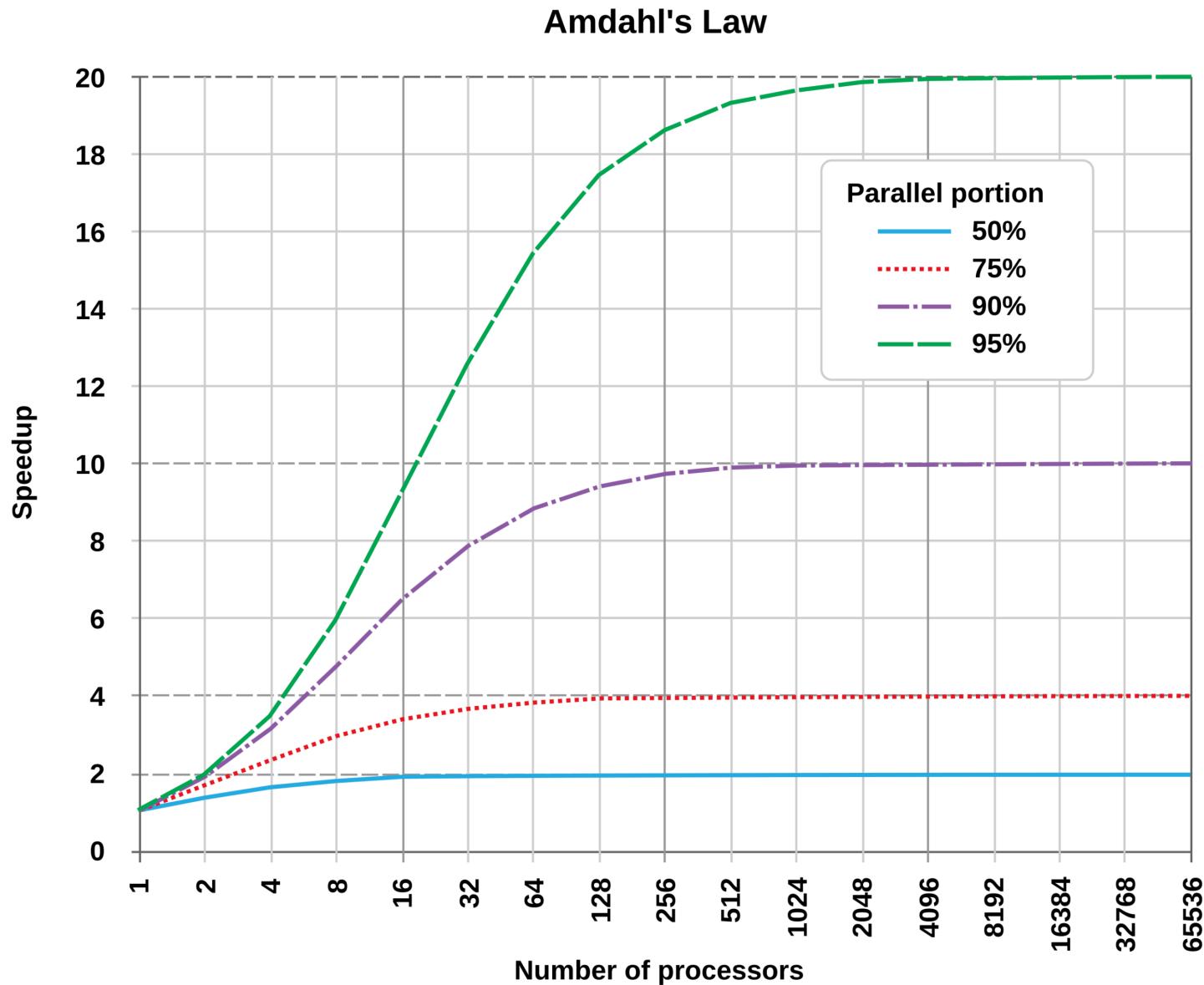
ここで、

- $S$  は全体の速度向上率 (Speedup)
- $P$  は並列/並行化可能な部分の割合
- $N$  は使用するプロセッサ (またはスレッド) の数

## アムダールの法則

下の図は並列化可能な部分を変えたときのグラフです。

並列化可能な部分が小さいほど、性能向上の頭打ちが早くなるだけではなく、プロセスを増やしたときの性能向上率も下がります。



## スレッドセーフとアムダールの法則

先ほどの `ThreadSafeCounter` では `increment` でいきなり lock を用いて排他制御をしてしまっているので  $P$ (並列化可能な部分の割合) は限りなく 0 です。

したがって、逐次処理と結果は変わりありません。(だからといって Lock が不必要なわけではありません。Lock は並列可能な割合を下げてしまう要因ですが、スレッドセーフのために欠かせません。)

# asyncioの便利な機能

## コルーチン間での値の受け渡し

コルーチン(またはスレッド)間でデータの受け渡しを行うためにはFutureオブジェクトを利用します。

他のコルーチンの処理結果を待ち、値を取得することができます。



```
import asyncio

async def get_http_response(future):
    print("Get http response")
    await asyncio.sleep(2) # Simulate an asynchronous operation
    future.set_result("Hello http!") # Set the result of the future

async def main():
    # Futureオブジェクトを作成
    future = asyncio.Future()

    # サーバからテキストを取得するコルーチンを起動
    asyncio.create_task(get_http_response(future))

    print("Waiting for the future result...")

    # future.set_result("Hello http!")が呼び出されるまで待つ
    result = await future
    print("result:", result)

asyncio.run(main())
```

# 参考文献

- 分散システム 原理とパラダイム
- <https://docs.python.org/ja/3/library/asyncio-eventloop.html>
- <https://docs.python.org/3/library/asyncio.html>
- <https://qiita.com/everylittle/items/57da997d9e0507050085>
- <https://ja.wikipedia.org/wiki/スレッドセーフ>
- <https://chocottopro.com/?p=764>
- <https://tech.connehito.com/entry/2023/09/25/130605>
- [https://en.wikipedia.org/wiki/Amdahl's\\_law](https://en.wikipedia.org/wiki/Amdahl's_law)

# 分散システムにおける一貫性

Riku Mochizuki

moz at sfc.keio.ac.jp

## 分散システムにおける一貫性

この章では分散システムにおける一貫性について説明します。

分散システムにおいて重要な機能は、データの複製（レプリケーション）です。

また次のセクションでは一貫性の議論と共にデータの複製プロトコルであるRaftを取り上げます。

## データーの複製

情報システムでは故障(fault)が発生します。

それはソフトウェア的な要因もあればハードウェア的な要因に分けられます。さらにその原因是自然災害、ヒューマン、経年劣化などさまざまです。

したがってそのような障害が起こらないシステムを作ることは不可能です。

## 情報システムに対する信頼

情報システムの可用性（ほしいときにはほしいデータを手に入れられる）を担保するためには、そのような障害が起きる前提を置き、システムを設計しなければなりません。

その設計のテクニックとしてよく使われる方法が**データの複製**です。

## データの複製

可用性を高めるために、一般に異なるコンピュータ（ノード）にデータを複製します。このような冗長構成にすることで、何か一つのノードが障害でダウンしてしまっても、ほかのノードが利用可能できます。

したがってシステム全体で見たときに可用性が担保できるわけです。

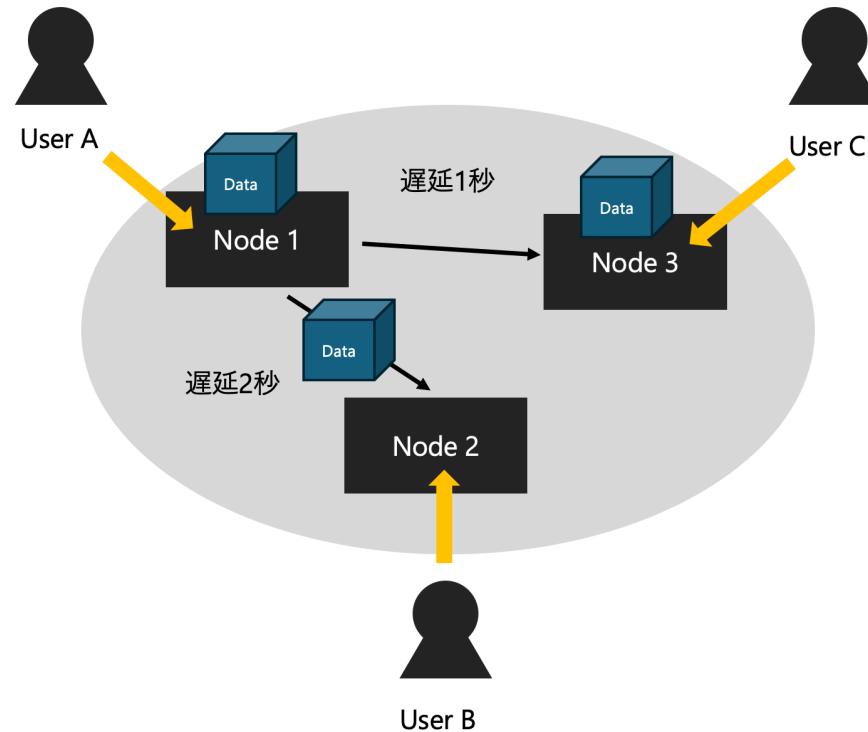
## データの複製

データを複製するためにはノード間の通信が必要です。

このような通信には遅延が起こります。さらにthe Internetなどのベストエフォート型ネットワークを用いる場合、データが届かなかったり、データが到着する順番が異なったりと予期せぬイベントが発生します。

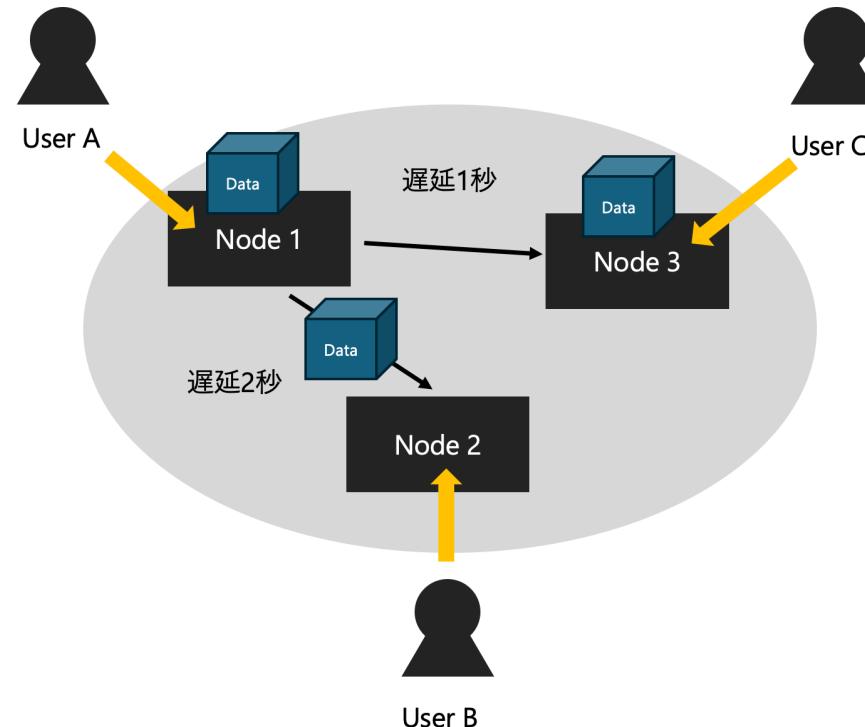
# データの複製

Node1からNode{2,3}に複製する場合、それぞれの経路はスペックが異なります。



# データの複製

Node 1 が Data を Node{2,3} に送ったタイミング（時間）を  $t$  (単位 : 秒) とします。  
 $t + 1$  秒では Node3 にはデータが届いている（ユーザーから見える）のに Node2 にはまだデータが届いていないためユーザーから見れません。



## データの複製

したがってある実時間において、各ノードの状態は異なります。

また、同じ時間でもクライアントによってはシステム（クラスタ内の任意のノード）から得た値がほかのクライアントと異なる場合もあります。

## 分散システムにおける一貫性

データがどのように複製されるか、それらの複製をクライアントはどのように観測するかを**一貫性**といいます。

さらにどのように複製/観測されるべきかをモデル化することができ、これを**一貫性モデル**といいます。

# 一貫性モデルの種類

一貫性には二つの観点があります。

## 1. データ中心一貫性モデル (Data Centric Consistency Model)

どのクライアントも一意の操作が観測できることを保証します。

しかし、広域に計算機が分散し、それらを協調するようなシステムでは、達成が困難な場合もあります。

## 2. クライアント中心一貫性モデル (Client Centric Consistency Model)

クライアント視点で一意の操作が観測できることを保証します。

システム全体では一貫性が保てていない時間もありますが、特定のクライアントからは一貫性があるように見えます。

# 一貫性モデルの具体例

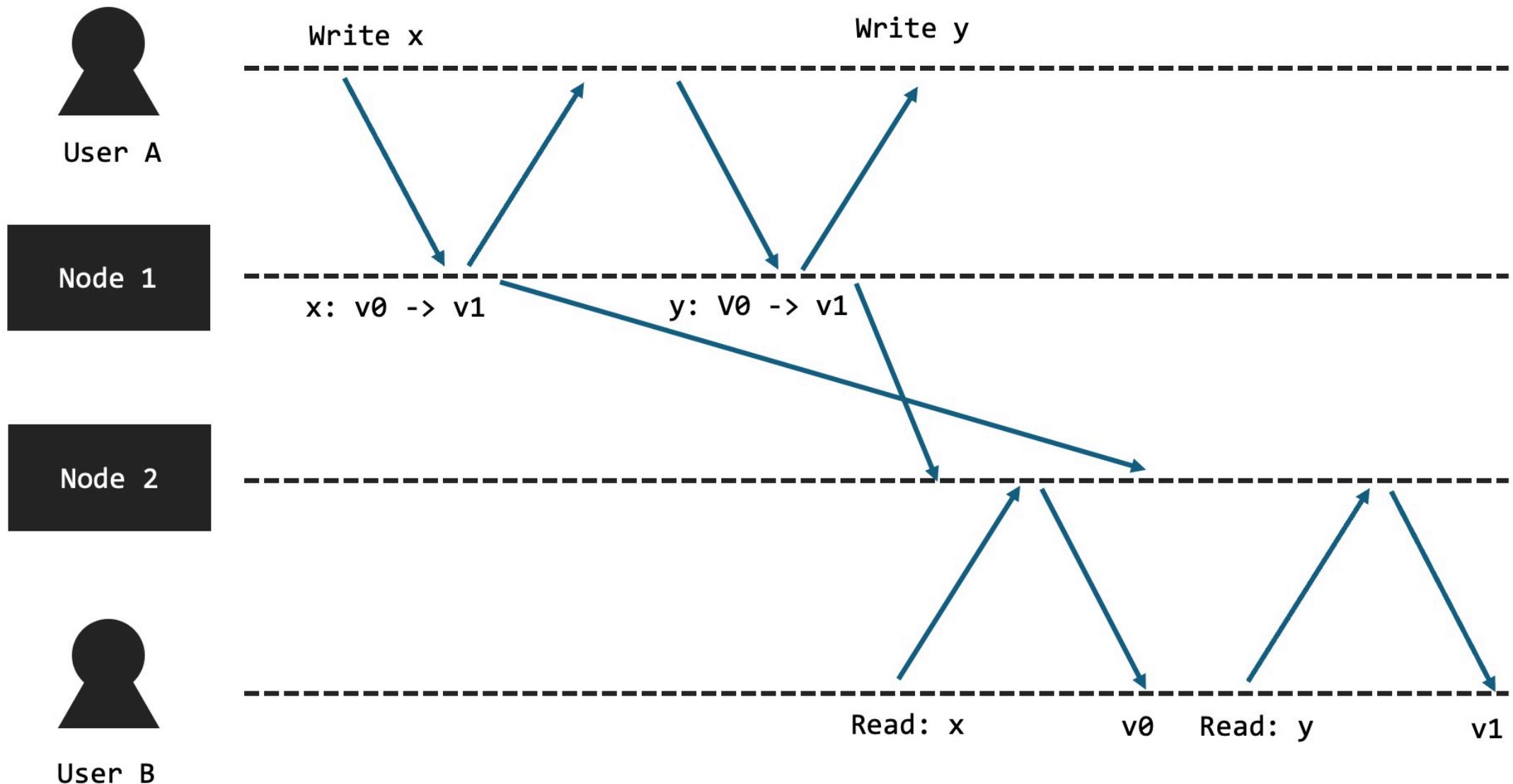
- クライアント中心一貫性モデル (Client Centric Consistency Model)
  - 結果整合性 (Eventual Consistency)
  - 因果一貫性 (Causal Consistency)
- データ中心一貫性モデル (Data Centric Consistency Model)
  - 逐次一貫性 (Sequential Consistency)
  - 線形化可能性 (Linearizability)

## 結果整合性 (Eventual Consistency)

結果整合性モデルでは、全てのノードが最終的に同じデータの状態に到達することを保証します。

操作の順序が一致しない場合もありますが、十分な時間が経過すれば全てのノードが一貫した状態になります。

どのクライアントも同じ順序で操作を観測できないが、最終的には状態が収束する。

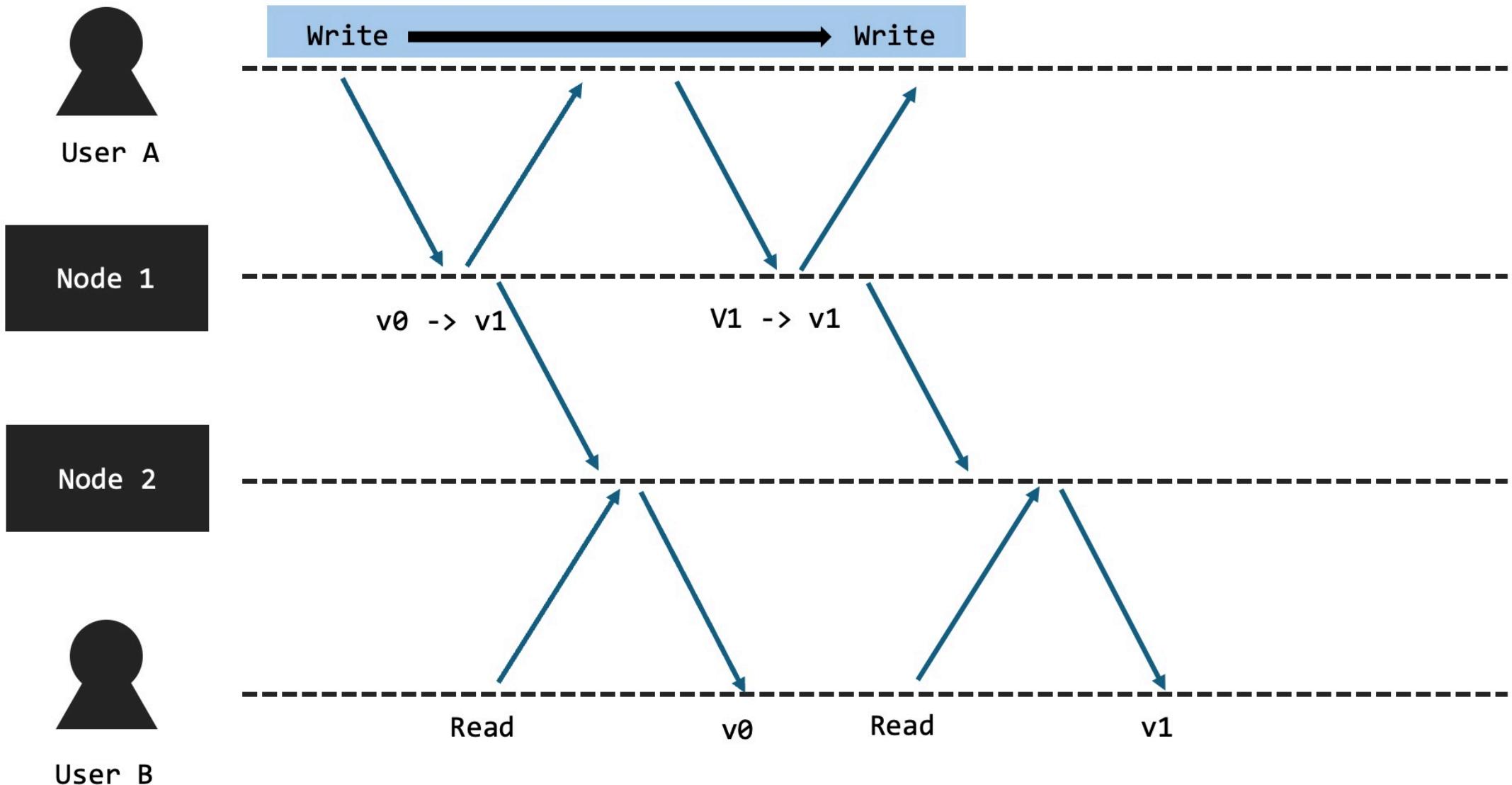


## 因果一貫性 (Causal Consistency)

因果一貫性は、因果関係を持つ操作が一貫した順序で実行されることを保証するモデルです。

因果関係のある書き込みは同じ順序で観測されますが、同時に行われる操作の順序は保証されません。

どのクライアントも同じ順序で操作を観測できないが、最終的には状態が収束する。



# 因果一貫性の詳細

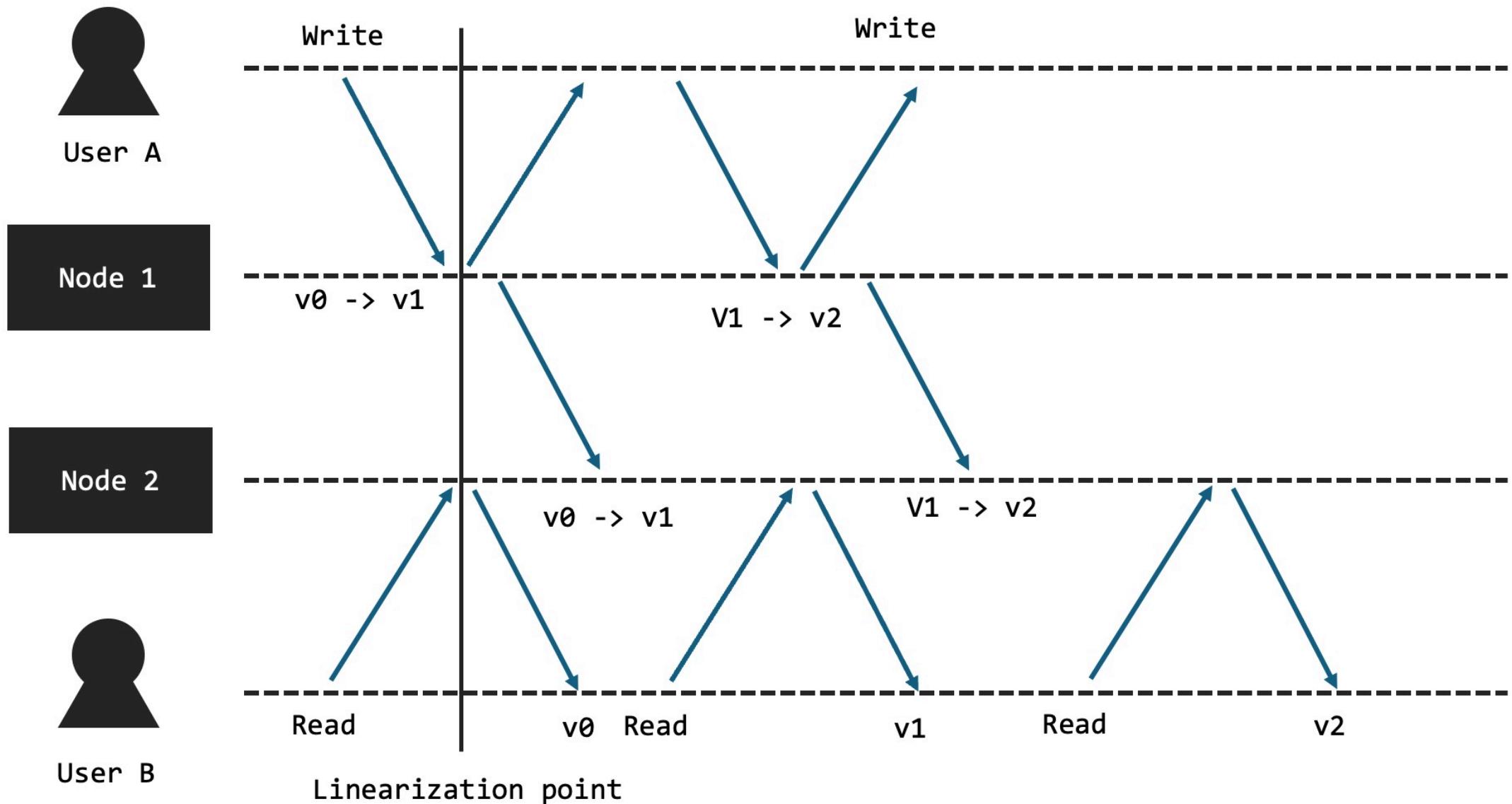
- **モノトニック読み取り一貫性 (Monotonic-Read Consistency):**  
クライアントがノードから値を読み込むと、以降の読み込みでは同じ値か新しい値が読み込まれる。
- **モノトニック書き込み一貫性 (Monotonic-Write Consistency):**  
クライアントの書き込み操作は、後続の書き込み操作よりも前に完了している。
- **書き込み後読み取り一貫性 (Read-Your-Write Consistency):**  
クライアントの書き込み操作の結果は、後続の読み取り操作で必ず観測される。
- **読み取り後続書き込み (Write-Follow-Read Consistency):**  
読み取り操作に後続する書き込み操作は、常に前回の読み取り操作時と同じか、新しい値に対して行われる。

## 逐次一貫性 (Sequential Consistency)

逐次一貫性は、全ての操作がシステム全体で一貫した順序で実行されることを保証します。

全てのプロセスは同じ順序で操作を観測しますが、操作がいつ行われるかはノードによって異なる場合があります。

すべての操作に対して、同一のクライアントは同じ順序で観測する。

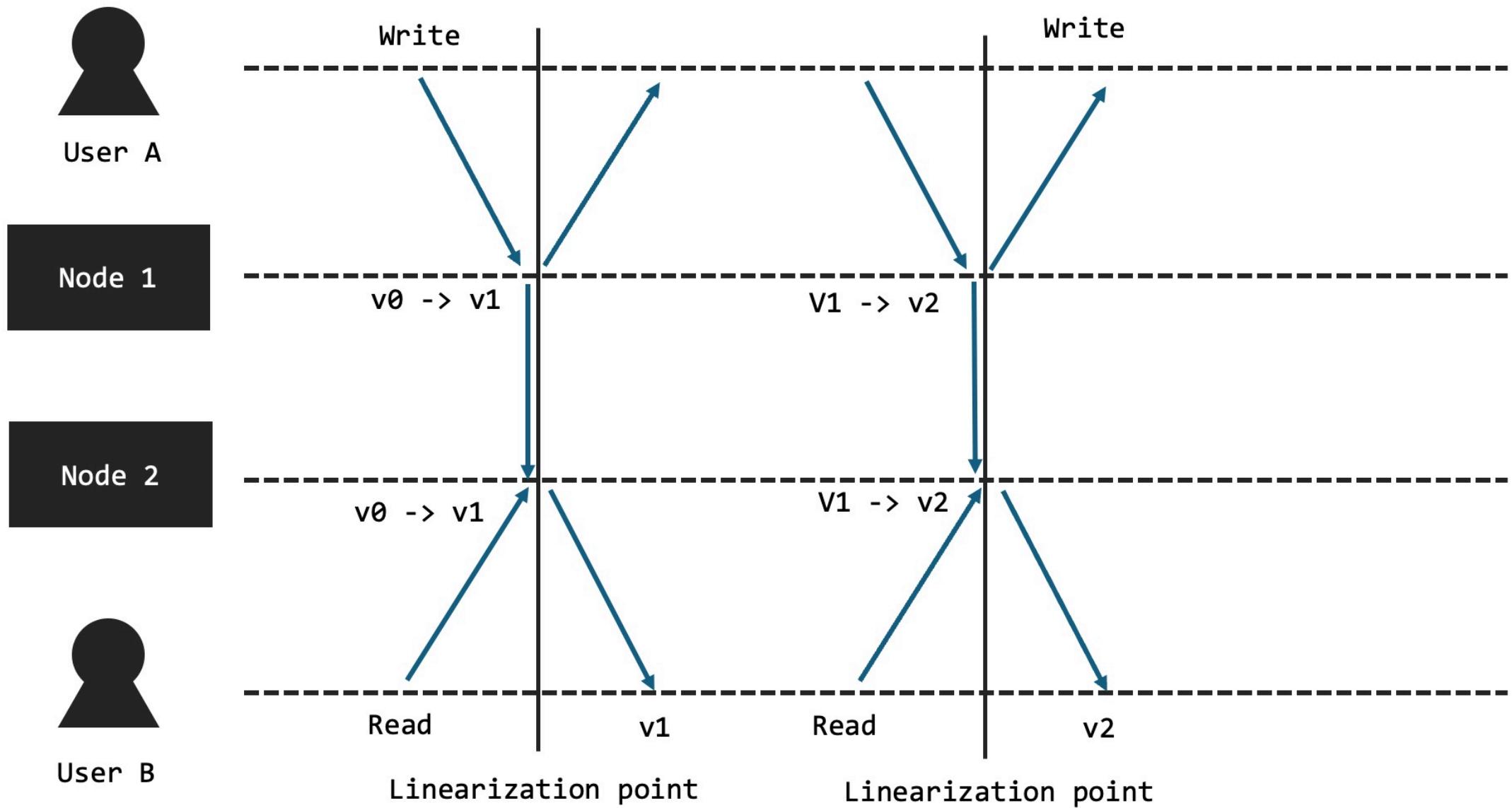


## 線形化可能性 (Linearizability)

線形化可能性は、全ての操作が実際に起こった瞬間に一意の順序で実行されたかのように見えることを保証します。

すべてのプロセスは実時間に基づいて同じ順序で書き込みが実行されたことを観測します。

すべてのクライアントは同じ時間に同じ状態を観測する。



## 線形化可能性の難しさ

一般に即座に（遅延が0秒）で更新データを他のノードに反映させることは不可能です。

したがって、プライマリーノードとセカンダリーノード分けて、すべての操作はプライマリーノードに送り、読み込みもプライマリーノードから行うようにすれば線形化可能性は達成されます。Raftは線形化可能性を保証します。

## 一貫性の分類

一貫性モデルは以下のように分類できる。

	実時間の一貫性あり	実時間の一貫性なし
操作の順序の一貫性あり	線形化可能性	逐次一貫性, 因果一貫性 (一部の操作群)
操作の順序の一貫性なし		結果整合性

## どの一貫性モデルが良いのか？

一概に良いというと、今の文脈では線形化可能性が一番良さそうに見えます。

ですが、システム全体で考えると必ずしも線形化可能性が良い選択になるとは限りません。

## どの一貫性モデルが良いのか？

シーケンシャルダイアグラムの通り、線形化可能性を実現するためには、ネットワークの遅延が0の場合、プライマリーセカンダリー構成にする場合、など極端な前提が必要なのです。

これらの前提を達成するためには膨大なコストが必要なほか、前提を達成することが不可能な場合がほとんどです。したがって、設計するシステムに応じて適切な一貫性モデルを選択することが大切です。

例えばAmazonの[DynamoDB](#)は結果整合性を保証します。ですが、冗長性を担保できたり、スループットを増加させたり、レイテンシを低くすることができます。

# 参考文献

- 分散システム 原理とパラダイム
- <https://qiita.com/kumagi/items/3867862c6be65328f89c>
- <https://techblog.yahoo.co.jp/architecture/2015-04-distributed-consistency/>
- <https://www.alexdubrie.com/posts/dynamodb-eventual-consistency/>
- <https://www.sraoss.co.jp/tech-blog/db-special-lecture/masunaga-db-special-lecture-11/>
- <https://www-higashi.ist.osaka-u.ac.jp/~nakata/mobile-cp/chap-06j-1.pdf>
- <https://christina04.hatenablog.com/entry/causal-consistency>

# Raft入門

**Riku Mochizuki**

moz at sfc.keio.ac.jp

# Raft

Raftはノード間でステートマシンの一貫性を担保するためのアルゴリズムです。

Raftは線形化可能性を保証します。

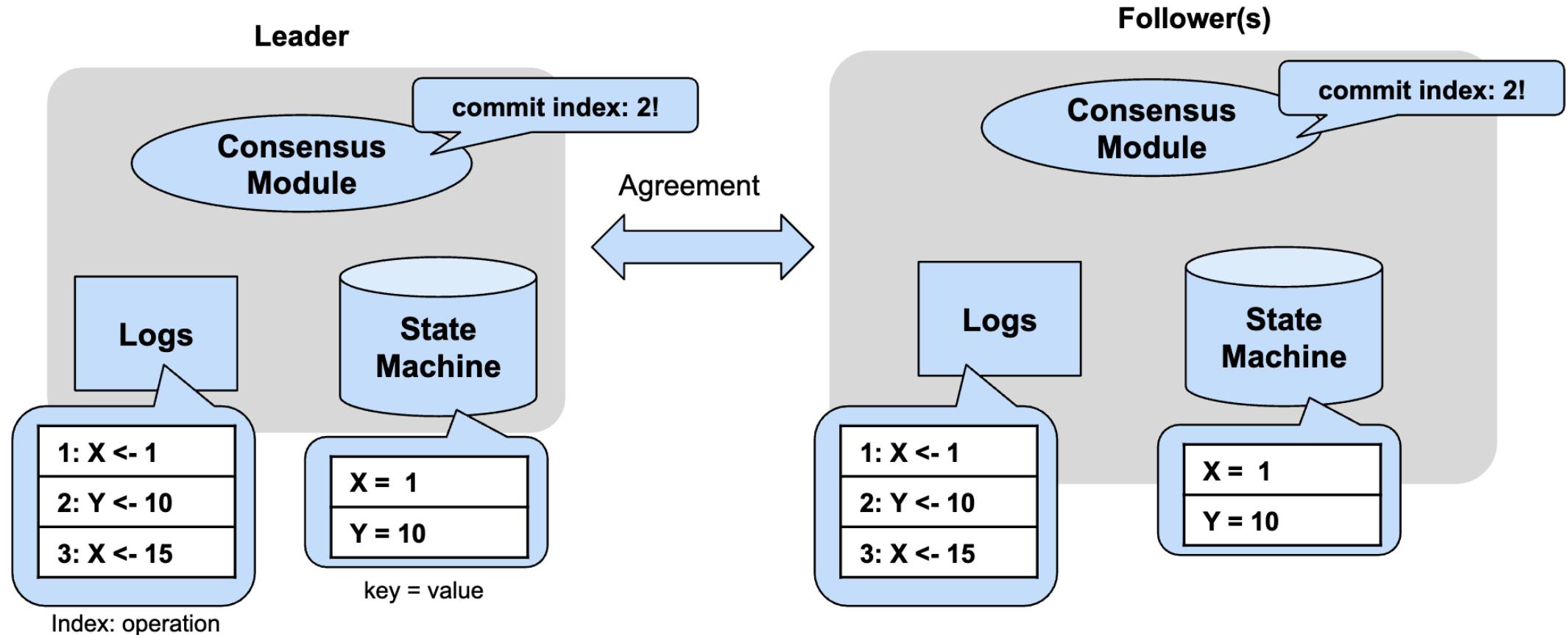
このアルゴリズムは分散型データベースやブロックチェーン、コントロールプレーンなど、様々なデータストアで利活用されています。

元論文: [In Search of an Understandable Consensus Algorithm - USENIX'14](#)

# Raftの構造

Raftは主に3つのパートから成り立っています。

- 1. コンセンサスモジュール:** 他のノードと通信を行い、合意アルゴリズムを実行する。
- 2. ログ:** 合意を得た/得られるログエントリを保存する。
- 3. ステートマシン:** 合意を得たログエントリを実行したキーバリューストア。ステートマシンの状態はすべてのノードで同じになる(ただし、ノードがアクティブな場合に限る)。



# ノードの状態

## 1. フォロワー (Follower)

クラスター内のデフォルトの状態で、リーダーからの指示を待ち、リーダーや候補者になるまで受動的な役割を果たします。

## 2. 候補者 (Candidate)

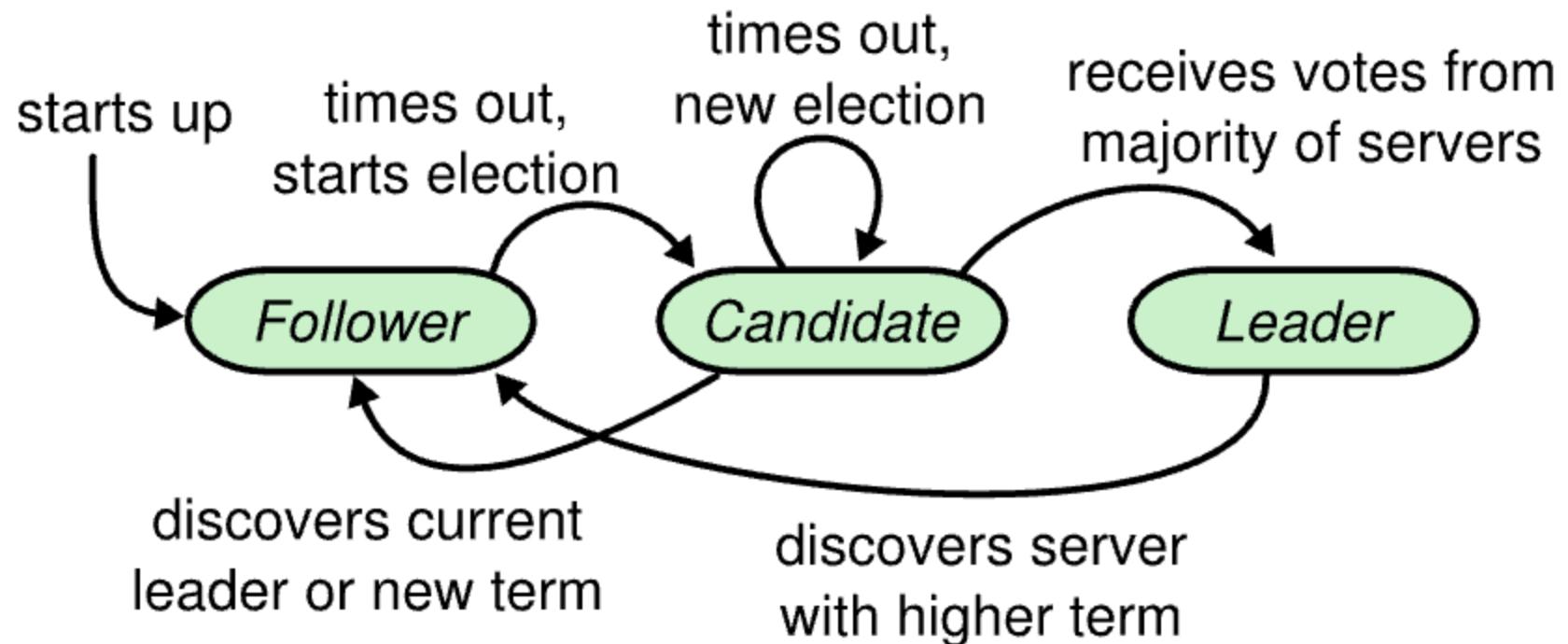
リーダー選挙に参加するためにフォロワーから昇格した状態で、リーダーになることを目指して他のノードに投票を依頼します。

## 3. リーダー (Leader)

クラスター内の単一のノードで、クライアントからのリクエストを処理し、フォロワーにログエントリを複製してクラスター全体の操作を調整します。

## ノードの状態

一般にフォロワー -> 候補者 -> リーダーという順に推移します。



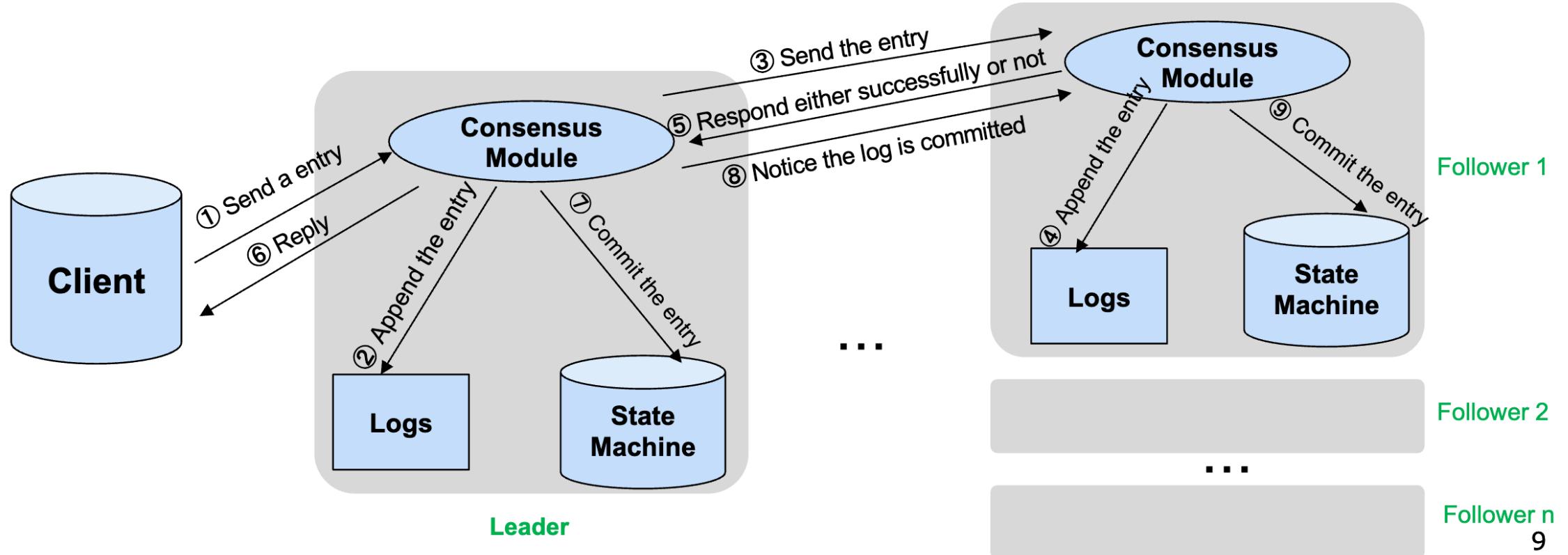
# Raftの処理

Raftは主に2つの処理から構成されます。

- **ログエントリーの追加:** ログエントリーの合意をノード間で取得し、ステートマシンにログエントリを適用する一連のプロセス。
- **リーダー選挙:** Raftでは合意にリーダーというノードが必要となる。そのノードをクラスター（ノードの集合）から決める一連のプロセス。

# ログエントリーの追加

ログエントリーの合意をノード間で取得し、ステートマシンにログエントリを適用する一連のプロセスを説明します。



## ログエントリーの追加 - 手順 1-3

### 1. クライアントからリーダーにエントリを送信する

クライアントは、リーダーとなっているノードにエントリを送信します。

### 2. リーダーのログにエントリを追加する

リーダーは受け取ったエントリを自身のログに追加します。

### 3. エントリをフォロワーに送信する

リーダーは、ログに追加したエントリをすべてのフォロワーに送信します。

## ログエントリーの追加 - 手順 4-6

### 4. フォロワーのログにエントリを追加する

フォロワーは、リーダーから受け取ったエントリを自身のログに追加します。

### 5. リーダーに対して成功または失敗を応答する

フォロワーは、エントリが正常にログに追加されたかどうかをリーダーに通知します。

### 6. クライアントに応答する

リーダーは、フォロワーからの応答を受け取り、エントリがコミットされたかどうかをクライアントに返答します。

## ログエントリーの追加 - 手順 7-9

### 7. リーダーの状態機械にエントリをコミットする

リーダーは、ログエントリが確定したと判断すると、それを自身の状態機械にコミットします。

### 8. フォロワーにログがコミットされたことを通知する

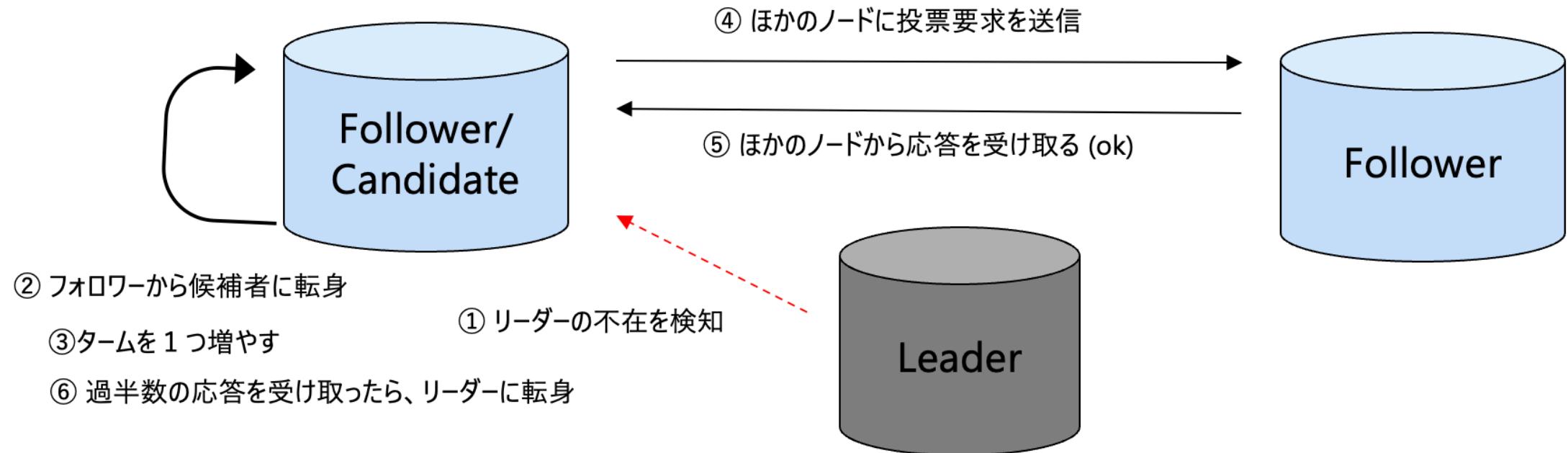
リーダーは、ログエントリがコミットされたことをフォロワーに通知します。

### 9. フォロワーの状態機械にエントリをコミットする

フォロワーは、リーダーからの通知を受け取り、自身の状態機械にエントリをコミットします。

## リーダー選挙

リーダーノードをクラスター（ノードの集合）から決める一連のプロセスを説明します。



## リーダー選挙 - 手順 1-3

### 1. リーダーの不在を検知する

フォロワーノードは、一定期間リーダーからのハートビートメッセージを受信しない場合、リーダーがダウンしたかネットワーク障害が発生したと判断します。

### 2. フォロワーから候補者に昇格する

ハートビートの欠如を検知したフォロワーは、自身を候補者に昇格させます。候補者はリーダーになることを試みます。

### 3. 選挙タイムアウトのリセットと任期の増加

候補者に昇格すると、選挙タイムアウトをランダムな期間にリセットし、自身のタームを1つ増やします。

# リーダー選挙 - 手順 4-6

## 4. 他のノードに投票要求を送信する

候補者は、他のすべてのノードに対して投票要求を送信します。この要求には、候補者の任期と自身のログ情報が含まれます。

## 5. 他のノードの応答を受け取る

他のノードは、投票要求を受け取ると、自身の状況に応じて投票するか否かを決定します。投票の条件は以下の通りです：

- そのノードがまだ投票しておらず
- 候補者のログがそのノードのログと同じかそれより新しい場合

## 6. 過半数の票を獲得し、リーダーに転身

候補者が過半数の票を獲得した場合、その候補者はリーダーに選出されます。

## リーダー選挙 - 上手くいかないケース

- タイムアウトにより新たな選挙が発生する可能性

選挙が終了せず、候補者が過半数の票を得られない場合は、新しい選挙タイムアウトが発生し、新たな選挙が開始される可能性があります。これにより、選挙がリーダーを選出するまで繰り返されます。

# Raftの安全性

Raftは以下の安全性を満たします。

プロパティ	説明
選挙の安全性	ある任期中に選ばれるリーダーは1つだけです。
リーダーの追記専用	リーダーはログエントリを上書きしたり削除したりせず、新しいログエントリのみを追加します。
ログの一貫性	2つのログエントリが同じindexとtermを含んでいる場合、そのログエントリは指定されたインデックスまでのすべてのエントリで一致します。
リーダーの完全性	ある任期中にコミットされたログエントリは、より高い任期のすべてのリーダーのログにも存在します。
ステートマシンの安全性	サーバーが特定のログインデックスでログエントリをステートマシンに適用した場合、他のサーバーが同じインデックスで異なるログエントリを適用することはありません。

# Raftの安全性



## 選挙の安全性

リーダー選挙の手続きにより、あるTermではLeaderノードは一つしか選ばれない。  
(ネットワークの分断によってクラスタ内にリーダーが二人以上生まれる場合はある)

## リーダーの追記専用

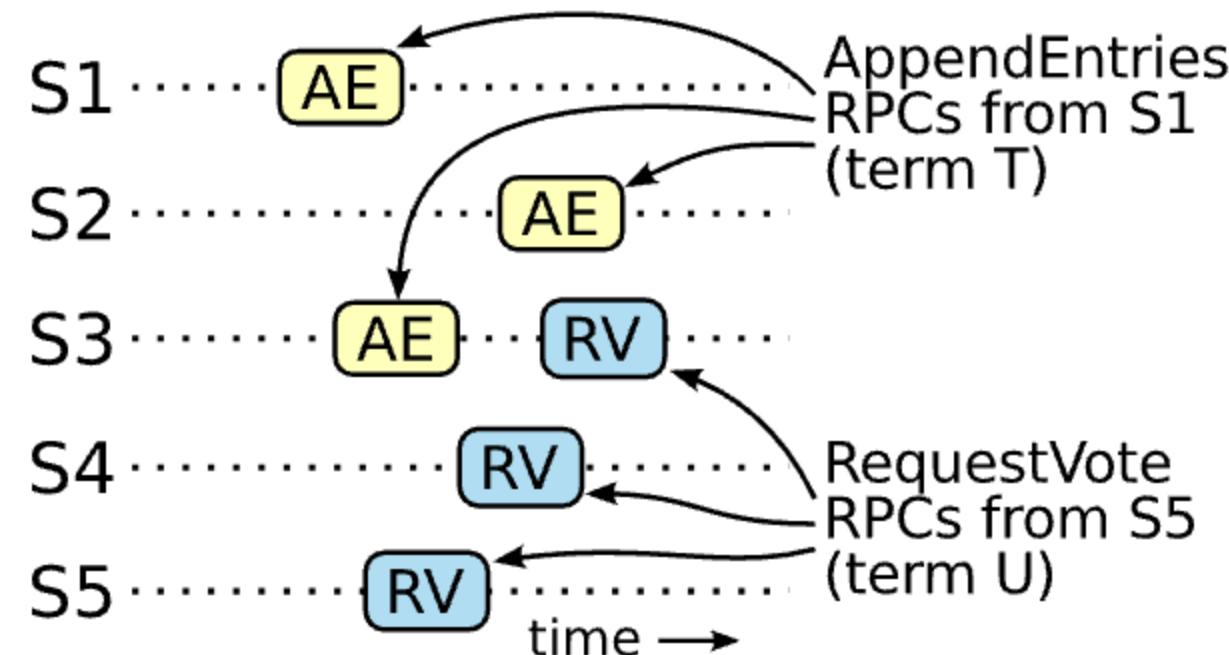
リーダーはフォロワーとの一貫性に矛盾が生じても、フォロワーのログエントリを上書きするため、追記のみである。

## ログの一貫

AppendEntriesでログの一貫性チェックを行い、一致しない場合はリーダーがフォロワーのログエントリを自身のログエントリに置き換える

# Raftの安全性 - リーダーの完全性

ログエントリが過半数のノードに伝搬されていれば、そのログは失われることはありません。かならず最新のログエントリをもつノードが一つ以上存在する。(リーダーの安全性の証明)



# Raftの安全性

## 選挙の安全性

リーダーの完全性によって必ず最新のログをコミットしたリーダーが選ばれる。したがってコミットされたログが一貫性を担保するために変更されることはない。

## Raftが保証する一貫性モデル

リーダーはログエントリにindexを付与し、フォロワー合意が得られたログエントリをindex順にステートマシンに適応する。

リーダーは適応したことをフォロワーに伝えることで、フォロワーはそのログをindex順にステートマシンに適応する。

したがってRaftはフォロワーから読み込む（フォロワーリード）ことを考えると逐次一貫性を満たす。

リーダーから読み込む場合は線形化可能性を満たす。

### フォロワーリードの問題について

## シミュレーター

Raftの動作を視覚的に確認できるサイト:

- [raft-consensus-simulator](#)
- [raft.github.io](#)

## おすすめ文献

- Raft Presentation

# レポジトリの説明

Riku Mochizuki

moz at sfc.keio.ac.jp

# ディレクトリ構成

- `docs/`: プロジェクトのドキュメントを含むディレクトリです。プロジェクトの概要や使用方法などが記述されています。
- `raft/`: プロジェクトのソースコードを含むディレクトリです。ここには、メインの実行ファイルやユーティリティ関数などが含まれます。

# 主要なファイル構成

- `run_node.py` : ノードを起動するためのスクリプトファイル
- `Dockerfile` : Dockerイメージをビルドするための設定ファイル。
- `docker-compose.yml` : 複数のDockerコンテナを定義し、ノード間のネットワークを設定するためのファイル。
- `generate_docker_compose.py` : 指定されたノード数に基づいて `docker-compose.yml` を動的に生成するPythonスクリプト。
- `raft/*.py` : Raftを実装するためのコードです。基本的に `raft/state.py` を編集することになります。

ひな形は用意してありますが、それぞれどのような実装になっているのか目を通してみよう!

# 必要な環境

- Docker
- Docker Compose
- Python 3.x (with pyyaml)

# セットアップ手順

## 1. リポジトリをクローン

```
git clone https://github.com/mzhkz/raft-on-weekends.git  
cd raft-on-weekends
```

# セットアップ手順 (続き)

## 2. 必要なライブラリをインストール

```
pip install pyyaml
```

## 3. docker-composeファイル (Raftクラスターの構成ファイル) の作成

```
python generate_docker_compose.py <ノードの数>
```

## セットアップ手順 (続き)

### 4. docker-composeを用いてクラスター（ノード群）を作成&起動

```
docker compose up -d --build
```

### 5. ある特定のノードのログ（開発ログ等）を見る場合

```
docker logs node{1~ノードの数} -f
```

## セットアップ手順 (続き)

### 6. クラスターの停止(削除含む)

```
docker compose down
```

# Hands-on

Riku Mochizuki

moz at sfc.keio.ac.jp

# 流れ



1. リーダー選挙を実装する
2. ログエントリの追加を実装する
  - Key-Value データベースを実装する (例: [RocksDB](#)やLevelDBなど)
  - ログストアを実装する
3. 5個のノードを立ち上げ、以下のケースで上手くいか試してみる
  - 定期的に `AppendEntries` を送信し、ログの合意が上手くいっていることを確認する
  - リーダーをダウンさせたあと、新たなリーダー選挙が行われるかを確認する
  - 3つのノードをダウンさせ、以下の状態を確認する：
    - ログエントリのコミットがいっさい進まない
    - リーダー選挙が終わらない

## 実装の詳細

実装の詳細は直接説明しつつ行っています。

わからないことがあればメンターに質問するか、論文、インターネットの記事（おすすめ記事は [2-raft.md#おすすめ記事](#)）に記載しています。

# 実装のポイント

コードを書いてみるのも大事ですが、以下の事項を明確にすることでスムーズなコーディングができます：

- 必要な変数はなにか？
- どのようなエッジケースがあるか？
- コルーチン化する部分はどこか？（どこでブロッキングが発生するか？）

# 大事なこと

## 分散システムを楽しもう！