マイクロサービスにおけるトランザクション処理パターンの切り替えによる整合性担保の時間削減

秋田谷 駿一[1]　飯島 貴政[2]　串田 高幸[1]

概要：マイクロサービスアーキテクチャでは, 主なトランザクション処理として TCC パターンと Saga パターンの 2 つがある. 2 つにはそれぞれ問題点がある. TCC パターンでは, 正常時にデータを取得できるまでの速度が遅い. Saga パターンでは, ロールバック発生時にロールバック前のデータを取得できるまでの時間が遅い. 本稿では, ロールバックを必要とする処理が頻繁には起こらないと仮定して基本は TCC パターンよりも約 20 %速度が早い Saga パターンを使用する. 本稿で提案する akira は, Dawid-Skene アルゴリズムをもとにした信頼度付与アルゴリズムを使用する. このアルゴリズムをもとにトランザクションが失敗するか成功することを予測する. その予測をもとに TCC パターンと Saga パターンを切り替える. akira を使用することでどちらか一方のパターンを使用するよりも早く整合性の担保を行える. 評価として, ロールバック処理が発生時に正しいデータを取得できるまでの時間と通常時にデータを取得できるまでの時間を akira と TCC パターン, Saga パターンで比較する. 結果として, すべての処理が完了するまでの時間が Saga パターンより約 26 %, TCC パターンにおいて約 20 %削減された.

1. はじめに

1.1 背景

近年, 技術の進歩によりビジネス環境が劇的に変化している. 主にコンピュータを使ったデジタルな変化が多く業務の一部を自動化し, 人の手がかかる部分を最小化することで効率を上げている [1]. 従来のようにすべての要件を固め長い期間をかけて開発をするウォータフォール手法では, この変化に対応することが困難である. そこで, 短い期間で開発やテストを繰り返し進めるアジャイル開発でのプロジェクト進行が求められている [2]. 理由としては, サービスを追加し続ける必要があるためである.

1.2 マイクロサービスアーキテクチャとは

マイクロサービスアーキテクチャとは, サービスを小さく分割して各サービスごとを連携させたものである. 図 1 に EC サイトを例にしたマイクロサービスアーキテクチャについて示す.

それに対しモノリシックアーキテクチャは, 従来までのある目的に対して分割されていない 1 つのモジュールで構成されたアプリケーション構造である [1]. 図 2 に同様の例のモノリシックアーキテクチャについて示す. マイクロサービスにおける重要な課題として一貫性の確保が難しい, サービスの分割の仕方, 設計の難易度が上がるというものが挙げられる [3]. データベースを分けているため, 同期に時間がかかることから即時に一貫性の担保は現状不可能とされている [4]. DB が複数別れている環境には, ACID 特性と呼ばれる関連する複数の処理を, 一つの単位として管理するトランザクション処理に求められる 4 つの特性が存在する. “Atomicity”（原子性), “Consistency”（一貫性）, “Isolation”（独立性）, “Durability”（耐久性）の 4 つの特性の頭文字を取っている. マイクロサービスアーキテクチャは, ローカルトランザクションを勧める一方, 分散トランザクションは推奨していない [5]. 分散トランザクションは, 複雑で不具合の原因になるからである [6]. マイクロサービスアーキテクチャで推奨されているのは, 分散トランザクションではなく複数データベースの結果整合性を利用することである [4]. 複数リソースの結果整合性を利用する方法として TCC パターンと Saga パターンが存在する.

[1] 東京工科大学コンピュータサイエンス学部
〒 192–0982 東京都八王子市片倉町 1404-1
[2] 東京工科大学大学院 バイオ・情報メディア研究科コンピュータサイエンス専攻
〒 192–0982 東京都八王子市片倉町 1404-1

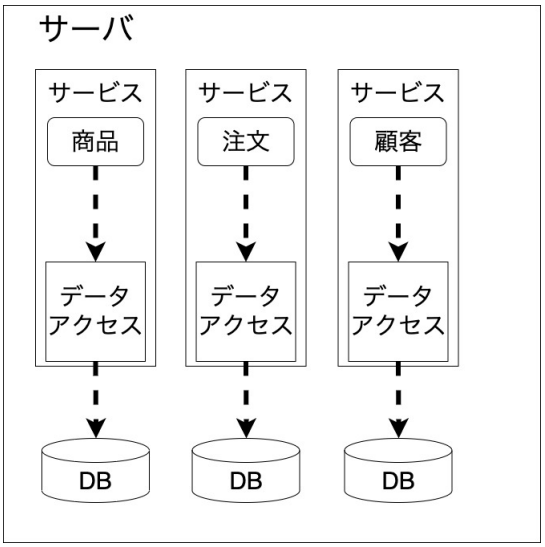

図 1: マイクロサービスアーキテクチャ

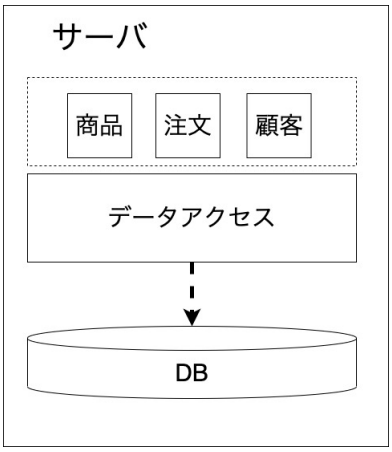

図 2: モノリシックアーキテクチャ

1.3 Saga パターン

マイクロサービス間でデータを同期する代表的な手法がイベントソーシングである [7]. サービス同士を疎結合にしながらデータの変更を非同期にやり取りする. イベントソーシングの活用により小さなサービスに閉じたローカルトランザクションを非同期につないでワークフローを構成する手法は Saga パターンと呼んでいる [8].

Saga パターンでは, ACID 特性の I(Isolation) が欠如している. そのため, Saga が並行して実行されている場合他のサービスが実行中の結果に影響を及ぼす [9].

1.4 TCC パターン

TCC パターンは, Try operations, Confirmation, and Cancellation の略称である [10]. TCC パターンは, Try フェーズと Confirm or Cancel フェーズに分かれている. Try フェーズでは, 各サービスに仮状態を登録する. また, タイムアウトに代表される障害が発生し仮登録ができなかった場合はキャンセル要請を送る. 一時的なエラーで送れていない可能性もあるため再送処理を行った後バッチ処理を行う. Confirm or Cancel フェーズでは, 仮登録成功した場合は, Confrim 要求を行なう. できない場合は, Cancel 要求を行なう. 各サービスは, 要求を受けとった後登録を行なう.

課題

本稿で上げる課題としてトランザクション中に TCC パターンと Saga パターンどちらか一方だと一貫性を確保することが困難という課題を挙げる [4]. この課題を証明するため, 両方のパターンで成功時, 失敗時に正しいデータの取得にかかる時間を計測した. この時間が早いということは, 一貫性を素早く確保できることにつながる. かかった時間を表 1 に表す. 平均時間は, 2000 件のリクエストを送ったときの 1 リクエストの平均を求めている. CPI[*1]の出すレポートでは, 写真の販売をするサイトでアクセスが瞬間アクセスの急増が発生した瞬間に 2000 件のリクエストが発生しているからである.

表 1: 正しいデータを取れるまでにかかった時間

	Saga	TCC
合計 (成功時)[s]	118	146
合計 (失敗時)[s]	130	113
すべての合計 [s]	249	259
1 リクエストの平均 [ms]	130	120

Saga パターンと TCC パターンの成功時の実行時間頻度を図 3 に表す. これは, 同一のトランザクション 1000 回送ったときの時間頻度である. 赤が Saga パターン, 青が TCC パターンを示す. 横軸が応答時間, 縦軸が発生回数を示している. 左側の発生回数が多いほど時間は早いことがわかる. 0.09〜0.1 の分布では, Saga パターンしか存在しない. 表 1 を見ると成功時の時間の合計は, Saga パターンが早いことがわかる.

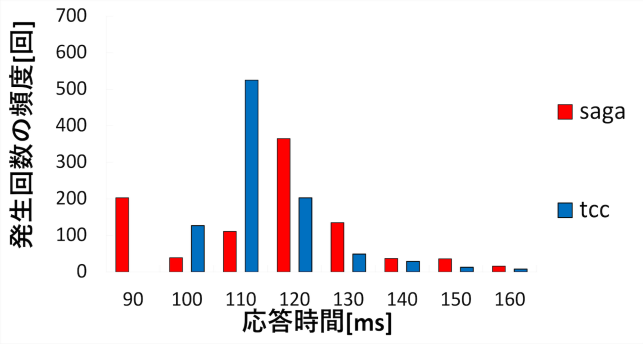

図 3: Saga と TCC の成功時の時間分布

Saga パターンと TCC パターンの失敗時の実行時間頻度を図 4 に表す. これは, 同一のトランザクション 1000 回送ったときの時間頻度である. 赤が Saga パターン, 青が

[*1] https://www.cpi.ad.jp/use/pakutaso/

TCC パターンを示す. 横軸が応答時間, 縦軸が発生回数を示している. この頻度図からわかることは, TCC パターンは Saga パターンに比べ短い時間の発生回数が多いことである. 表 1 を見ると失敗時の合計時間は, TCC パターンのほうが早いことがわかる.

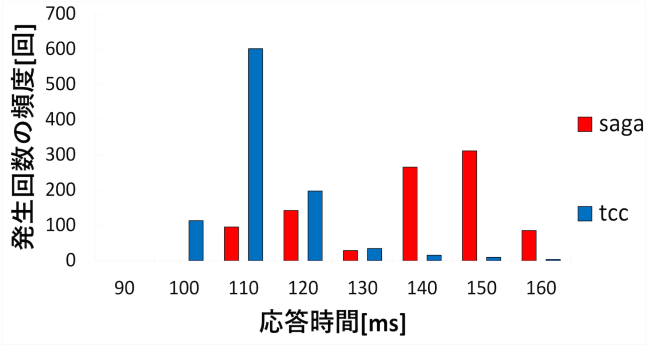

図 4: Saga と TCC の失敗時の時間分布

データの一貫性の確保ができていないとトランザクション中に DB にアクセスが来て虚偽のデータを取得するダーティリード, ファントムリード, ファジーリードが発生する [11]. DB には, 分離レベルが存在する. これはダーティリード, ファントムリード, ファジーリードの現象が発生することを防げるかどうかを示すレベルになっている [11]. READ UNCOMMITTED は, データベースの変更するときにロックがかかりトランザクションの終わりまでロックされる. なお, 読み取りにはロックがかからない. READCOMMITTED は, データベースの読み取り時と変更時にロックがかかる. 読み取り後にロックは解除される. 変更された部分のロックは, トランザクションの終わりまで継続される. REPEATABLE READ は, データベースの読み取りと変更時にロックがかかる. 変更された部分すべてのロックは, トランザクションの終わりまでロックされる. 読み取りも同様にトランザクションの終わりまでロックされる. ハッシュ構造などの変更不可能なアクセス構造のロックは読み取り後に解除される. SERIALIZABLE は, データセットの影響を受ける行に, トランザクションの終了までロックがかかる. 変更されたものすべてがトランザクションの終了までロックされる. 表 2 にトランザクションの分離レベルについて示す. 表の〇は問題が発生する. X は発生しない.

表 2: トランザクションの分離レベル

	ダーティリード	ノンリピータブルリード	ファントムリード
READUNCOMMITED	〇	〇	〇
READCOMMITTED	X	〇	〇
REPEATABLEREAD	X	X	〇
SERIALIZABLE	X	X	X

以上のことから, トランザクション中にどちらか一方の手法を使用した場合に一貫性を確保することが困難という課題をあげる.

2. 関連研究

Butler らの研究では, 調整者と参加者の 2 つに分類 [12]. 調整者は, 参加者に向かってコミットできるか聞く. 可能であれば行う. 成功した場合は, 成功, 失敗した場合は失敗と返答する. 調整者は, 全参加者から応答が来るまで待つ. 参加者の中に, 失敗が存在する場合調整者は, ロールバックせよと命令を送る. Butler らは Two-Phase Commit(以下 2PC) を提案している. 2PC は, 調整者に障害が発生した場合に問題がある. 参加者からの返答がないと次の動作ができてない. また, マイクロサービスではトランザクションの衝突が多いため向いていない. Pan Fan らの研究では, 2PC を使い一貫性の高いトランザクションを実現している [13]. 既存の 2PC ではトランザクションの衝突の多いマイクロサービスでは向いていない. Pan Fan らは, 2PC*を提案している. 2PC*では, トランザクション中にデータベースロックを保持する. データベースロックを保持することで, オーバーヘッドを大幅に削減することに成功している. 2PC*は最大でスループット 3.3 倍、レイテンシ 67 %を改善している. しかし, ロックを行なうことでデッドロックが起きる可能性がある. Wilhelm Hasselbring の研究では, マイクロサービスでの障害検知を行っている [14]. Wilhelm Hasselbring は, 小さいサービスが故障しても他のサービスに影響しないシステムの提案. 故障を素早く検知, 復旧させている. 復旧は自動で行えるようになっている. しかしこの提案では, 一貫性の確保はできていない.

Furda らの研究では, データの一貫性を読み取り専用の操作と書き込み専用の操作に分ける [15]. トランザクション中は, 書き込みロックをかけ虚偽のデータを書き込まないようにしている. しかし, この手法では読み取りは可能. また, ロックを掛けることから整合性が取れるまでに時間がかかる.

3. 提案方式

本稿では, トランザクションが成功するか失敗するかを予測することで, Saga パターンと TCC パターンを切り替える. トランザクションが来てから切り替えを行うと切り替えに時間がかかるため, 事前に予測しておくことで切り替えまでに時間を減らすことができる.

提案方式

まずワーカーを用意する. 信頼度をもとに akira は切り替えを行っている. 信頼度は, 0 から 100 の値になっている. 最小の値は, 0 である. 最大の値は, 100 である. ワーカーはそれぞれ信頼度とランクを持っている. 信頼度の初期値は, すべて 0 である. ここでいう信頼度は, ワーカーのもつ 0 から 100 の数字である. 信頼度は実際の結果と予

想した結果を比べ一致していれば増えていく．ワーカーは success か failed を答える．success ならば Saga パターンで実行する．failed ならば TCC パターンで実行する．その中で 1 番信頼度の高いワーカーの出した予測を尊重する．図 5 では，ワーカーが 3 つ存在している．ワーカー 1 が 1 番信頼度が高いことになる．ここでは，ワーカー 1 の出した success という予測を尊重する．

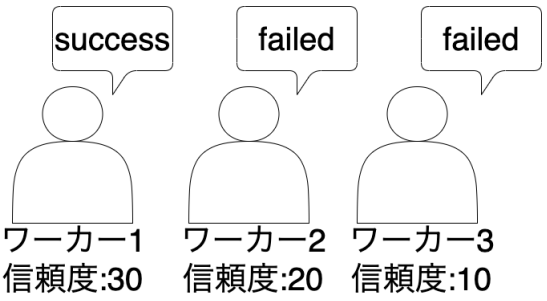

図 5: 切り替え手法

本稿では，3 つのワーカーと 3 つの評価項目を用意した．Saga パターンで時間のかかる処理としては，ポイントを戻すや残高を戻す，在庫を戻すといったロールバック処理である．そのため，ロールバック処理を行う項目を評価項目とする．akira は，3 つのワーカーの中から一番信頼度の高いワーカーの出した予測を選択する．ワーカーはそれぞれ評価項目ごとに違う重みを持っている．以下に 3 つの評価項目を示す．

評価項目 1:ポイント使用の有無
評価項目 2:在庫の数
評価項目 3:ユーザーのキャンセル数

ポイント利用の有無を評価項目 1 にした．これは，Saga パターンで実行した際に決済部分でキャンセルが行われると在庫を戻す処理とポイントを戻す処理の 2 つが発生するために評価項目とした．在庫を評価項目 2 にした．これは，在庫の残りの数が少ない場合に Saga パターンで実行すると注文が完了する前に在庫を減らす処理を行う．そのため，決済部分でキャンセル処理が行われてしまうと在庫を戻すまでに時間がかかる．以上のことから，在庫の評価項目を用意した．ユーザのキャンセル数を評価項目 3 にした．これは，キャンセル率が高いユーザからのトランザクションは失敗しやすいからである．ワーカー 1 は，在庫情報に重きをおいている．ワーカー 2 は，ポイントを使用しているかどうかに重きをおいている．ワーカー 3 は，ユーザのキャンセル回数に重きをおいている．

信頼度付与アルゴリズム

図 6 に実行結果が failed だった場合の図 5 からの信頼度の変化を示す．赤字になっている部分が変化した部分である．failed と予測していたワーカー 2 と 3 の信頼度が 2 足されている．success と予測したワーカー 1 は信頼度が 1 足されている．これをいずれかのワーカーの信頼度が 100 になるまで続ける．

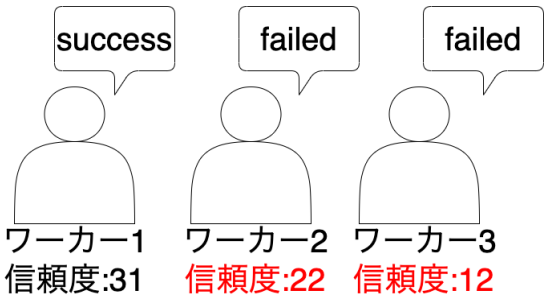

図 6: 信頼度付与

ワーカーの出した予測と実際のトランザクションで答え合わせをする．正解のものは，2 プラスする．不正解のものは信頼度を 1 プラスする．信頼度の上限を 100 とするため，どこかの信頼度が 100 に達したら初期化する．初期化の値としては，100 に達したワーカーのみ 1 となる．それ以外のワーカーは信頼度が 0 になる．信頼度を初期化しないと 1 つのワーカーに意見が偏ることから初期化を実行している．初期化を実行した上で，100 に達したワーカーは信頼度のランクを 1 あげる．1 分 43 秒間はランクを考慮した選択とランクを考慮しない選択を行う．正答率の高い方を以後のワーカーの結果として使用する．

予測アルゴリズム

ソースコード 1 はワーカー 1 の疑似コードである．ポイントを使用していた場合は，ワーカー内の評価項目 1 に 2 プラス，それ以外の項目に 1 をプラスする．使用していなかった場合は，すべての評価項目に 1 プラスする．実際に使用したポイントが評価項目を足したときよりも多ければ success と判断する．それ以外の場合は，failed と判断する．

ソースコード 1

```
1  function worker1():
2      global ca1, work1, points,hyouka11,
       ↪  hyouka12, hyouka13
3      if points != 0:
4          hyouka11 += 2
5          hyouka12 += 1
6          hyouka13 += 1
7      else:
8          hyouka11 += 1
9          hyouka12 += 1
10         hyouka13 += 1
11     sum = hyouka11 + hyouka12 + hyouka13
12     if points > sum:
13         work1 = "success"
14     else:
```

```
        work1 = "failed"
    return work1
```

ソースコード 2 はワーカー 2 の疑似コードである．在庫数が評価項目 2 から評価項目 1,3 の和を引いた値未満の場合は，ワーカー内の評価項目 2 に 2 プラス，それ以外の項目に 1 をプラスする．それ以外の場合は，すべての評価項目に 1 プラスする．在庫数が評価項目を足したときよりも多ければ success と判断する．それ以外の場合は，failed と判断する．

ソースコード 2

```
function worker2():
    global hyouka21, hyouka22, hyouka23
    global ca2, work2, zaiko
    if zaiko <
    ↪  hyouka22-(hyouka21+hyouka23):
        hyouka21 += 1
        hyouka22 += 2
        hyouka23 += 1
    else:
        hyouka21 += 1
        hyouka22 += 1
        hyouka23 += 1
    sum = hyouka21 + hyouka22 + hyouka23
    if sum <= zaiko:
        work2 = "success"
    else:
        work2 = "failed"
    return work2
```

ソースコード 3 はワーカー 3 の疑似コードである．すべての評価項目を足してユーザーのキャンセル回数よりも少ないときは，評価項目 3 を 2 プラスして，他の項目は 1 プラスする．それ以外の場合は，すべての項目を 1 プラスする．すべてを足し合わせた時，キャンセル数よりも少ない場合 success と判断する．それ以外の場合は，failed と判断する．

ソースコード 3

```
function worker3():
    global ca3, work3, hyouka31, hyouka32,
    ↪  hyouka33
    cancel_count = user()
    if cancel_count > hyouka31 + hyouka32
    ↪  + hyouka33:
        hyouka31 += 1
        hyouka32 += 1
        hyouka33 += 2
    else:
        hyouka31 += 1
        hyouka32 += 1
        hyouka33 += 1
    sum = hyouka31 + hyouka32 + hyouka33
    if sum < cancel_count:
        work3 = "success"
    else:
        work3 = "failed"
    return work3
```

プログラムの流れを図 7 に示す．ユーザーからリクエストが来ると，alkira から Saga パターン，TCC パターンのどちらか選択する．Saga パターンの場合，成功すると結果の返答を行う．失敗時はロールバック処理を行う．TCC パターンの場合，成功すると結果の返答を行う．失敗時は仮登録の取り消しを行う．

ユースケース・シナリオ

本稿で想定したユースケースシナリオは，在庫があり，ポイントを利用可能なオンラインショッピングサイトである．ユーザがアクセスして，商品を選び決済を行うというものである．今回は，ポイントを利用した決済と決済の上限額を設定した．ポイントを利用した決済の場合注文に失敗した場合，ポイントが戻るまでに時間がかかることがある．在庫があることで本当の数が表示されないという問題を再現している．

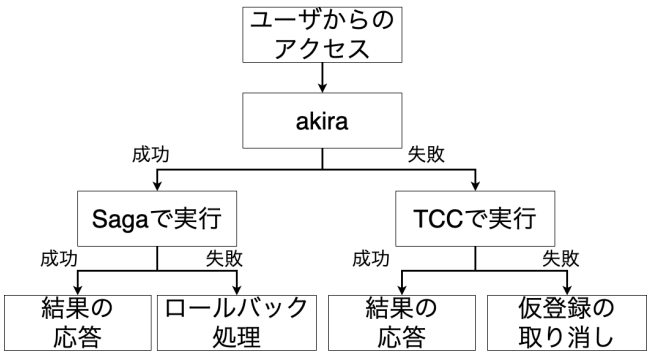

図 7: プログラムの流れ

4. 実装と実験方法

4.1 実装

Saga パターンと TCC パターンを切り替えるソフトウェアを実装する．

4.2 実験方法

実験環境として自分で作成したオンラインで決済を行うオンラインショッピングサイトを使用する．本稿では，Python プログラムを用いて 2000 回リクエストを送った．CPI[*2]の出すレポートでは，写真の販売をするサイトでア

[*2] https://www.cpi.ad.jp/use/pakutaso/

クセスが瞬間アクセスの急増が発生した瞬間に 2000 件のリクエストが発生しているからである．成功する 1000 件のリクエスト, 失敗する 1000 件のリクエストを送った．これは, 切り替えの有効性を示すために連続して成功するトランザクションと連続して失敗するトランザクションを同数送ったからである．この失敗する 1000 件のトランザクションの件数は, 実際には低い値になるが成功時と同じ数で比較するために 1000 件とした．いずれもポイントは, 使用している．

4.3 実験環境

サービスの構築表を表 3 に表す．

表 3: 構築表

サービス	役割
注文	ユーザーから何を注文するかリクエストを受ける
ユーザー	どのユーザーがログインしているかの確認
商品	商品の在庫管理
決済	支払いを行う

ユーザーは, まずログインを行う．次に, メニューの中から注文する商品を選ぶ．商品サービスは選択された商品の在庫を減らす．そして, 決済サービスに値段を伝える．ユーザーは, オンラインで決済を行う．決済サービスは, 上限額に達していないかの確認を行い決済を完了させる．

5. 評価手法と分析手法

5.1 評価手法

既存手法である TCC パターンと Saga パターンとの比較を行う．図 8 にロールバック発生時の時間取得方法を示す．前提としてポイントを利用した注文である．ロールバック発生時では, 注文リストを取得した際に datetime を書き込む．これを t1 とする．図 8 では, 決済サービスでエラーが発生している．決済サービスでエラーが発生すると使用したポイントを戻す処理が発生する．ここで datetime を取得する．これを t2 とする．t2 から t1 を引いた値がロールバック発生時かかった時間となる．図 9 に正常時における処理時間の取得方法を示す．前提としてポイントを利用した注文である．ロールバック発生時では, 注文リストを取得した際に datetime を書き込む．これを t1 とする．ポイントを減らす際に datetime を書き込む．これを t2 とする．t2 から t1 を引いた値が正常時にかかった時間となる．

5.2 分析手法

本稿では, ロールバック発生時の時間と正常時の時間を Saga パターン, TCC パターンどちらか一方のみを使用した場合と akira を使用した時間を比較する．成功時の 1000 件と失敗時の 1000 件のデータを使って比較する．

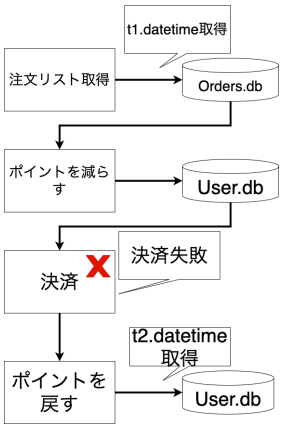

図 8: ロールバック時評価方法

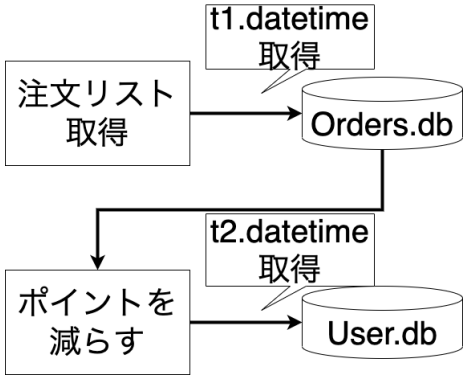

図 9: 正常時評価方法

図 10 に各パターン, akira の正常時の時間頻度を示す．赤が Saga パターン, 青が TCC パターンを示す．横軸が応答時間, 縦軸が発生回数の頻度を示している．左側の発生回数が多いほど時間は早くなる．この図から akira は, 正常時 90 [ms] から 100 [ms] の範囲に約 30 %分布していることがわかる．つまり, 同じく 90 [ms] から 100 [ms] の範囲に約 20 %分布している Saga パターンと同程度応答時間が速いことがわかる．

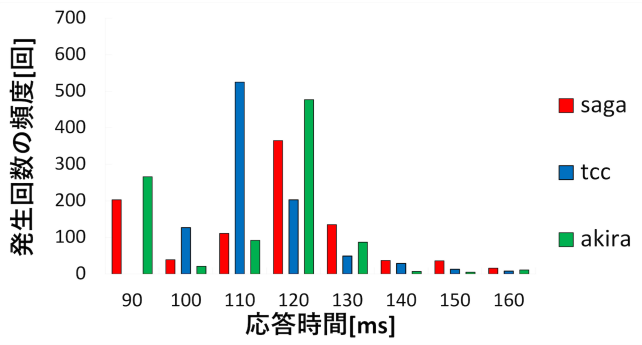

図 10: 正常時の時間分布

図 11 に各パターン, akira のロールバック発生時の時間頻度を示す．赤が Saga パターン, 青が TCC パターンを示す．横軸が応答時間, 縦軸が発生回数の頻度を示している．左側の発生回数が多いほど時間は早くなる．この図からわかることは, TCC パターンが多く分布している 100 [ms]

から 110 [ms] に akira も多く分布してることから TCC パターンと同程度速い事がわかる．

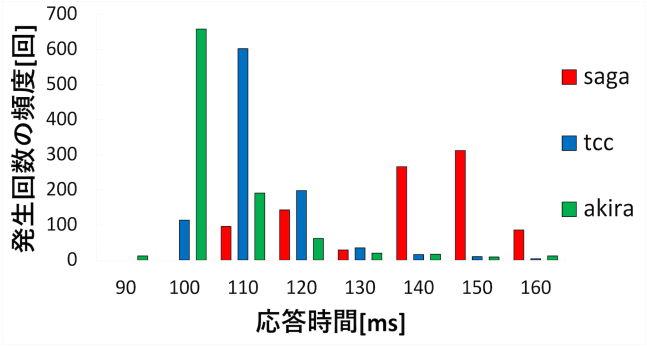

図 11: ロールバック発生時の時間分布

表 4 に実行時間を示す．この表を見ると，akira が正常時 Saga パターンよりも早く，ロールバック発生時 TCC パターンよりも早いことがわかる．また，2000 件中 1989 件合致していた．正解率 99.45 %となっている．平均時間は，2000 件のリクエストを送ったときの 1 リクエストの平均処理時間を求めている．

表 4: 正しいデータを取れるまでにかかった時間

	Saga	TCC	akira
合計 (成功時) [s]	118	146	109
合計 (失敗時) [s]	130	113	101
すべての合計 [s]	249	259	210
1 リクエストの平均 [ms]	130	123	103

図 12 に成功するか失敗するかわからないトランザクションを akira に 1000 件送ったときの結果を示す．

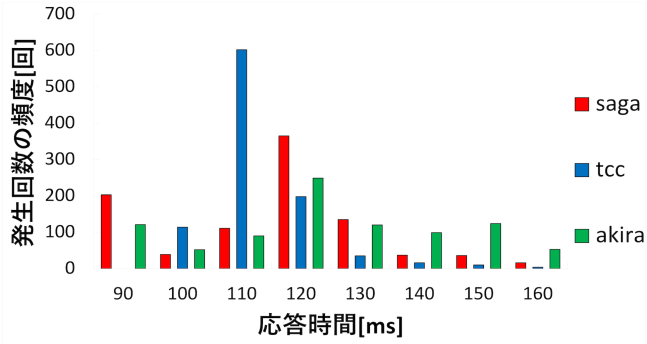

図 12: akira の時間分布

この図からわかることは，140 [ms] から 160 [ms] までの他のパターンがあまりない分布に akira がいることである．ここでは，Saga パターンは正常時，TCC パターンはロールバック発生時の時間を使用した．akira の正常時の正解率は，656 件中 573 件の約 87.3 %となっている．また，ロールバック発生時は 344 件中 35 件の約 10.2 %となっている．このことから，ロールバック発生時の正解率を上げることが時間の削減につながる．

6. 議論

本稿では，信頼度の高いワーカーの意見を尊重した．しかし，一つのワーカーに判断が偏ることが起きる．そのためワーカーの数を増やし，すべてのワーカーで平均値を取る．その平均値よりも信頼度の高いワーカーの中で多数決を取ることで一つのワーカーに判断が偏ることは起きにくくなる．ユーザーのキャンセル率を評価項目 3 としたが，今までのトランザクションでキャンセルされた商品の在庫と近い数または，ユーザのもつ上限額に近い数とすることで精度が上がる．ワーカー内の評価項目においても単純に 1 や 2 を足すのではなく，キャンセルがあった在庫数に近い分布やポイントの量で足す量を変化させる必要がある．足す量の範囲として 0 と 1 を与える．より頻度が高い値に近くなるほど最大値である 1 に近い数字を足す．頻度が低い数字であれば最小値である 0 に近い数字を足す．本稿では，ユーザーの数は 1 つで実験したが実際のユーザーの数はもっと多い．このことから，同時に複数のアクセスが来ることがわかる．同時に複数のアクセスが来ることを考慮しておく必要がある．例えば，上限額がユーザーによって違うため取得するのに時間がかかる事がある．実験の環境として同一コンピュータ上でサーバーを立てて実験を行ったが，Kubernetes 上に実装した際は，ネットワークを経由するためデータに変動がでる．

7. おわりに

本稿では，Saga パターンと TCC パターンをトランザクションによって切り替えることで正しいデータを取れるまでの時間を削減した．akira では，信頼度を付与したワーカーがトランザクションの予測を行う．成功 1000 件，失敗 1000 のトランザクションを与える実験の結果として，切り替えを行うことですべての処理が完了するまでの時間が Saga パターンより約 26 %，TCC パターンにおいて約 20 %削減された．しかし，成功するか失敗するか定かではないトランザクションに関しては，失敗時のトランザクション予測がうまく行かずに低い正解率となってしまった．しかし，成功時の予測は高い精度でできているため評価項目と各ワーカーの評価項目の計算式を単純なものではなく在庫数，ポイントを考慮したものにしなければいけない．

参考文献

[1] Cerny, T., Donahoo, M. J. and Trnka, M.: Contextual understanding of microservice architecture: current and future directions, *ACM SIGAPP Applied Computing Review*, Vol. 17, No. 4, pp. 29–45 (2018).
[2] Livermore, J. A.: Factors that Significantly Impact the Implementation of an Agile Software Development Methodology., *J. Softw.*, Vol. 3, No. 4, pp. 31–36 (2008).
[3] Wolff, E.: *Microservices: flexible software architecture*,

Addison-Wesley Professional (2016).

[4] Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J. and Tilkov, S.: Microservices: The journey so far and challenges ahead, *IEEE Software*, Vol. 35, No. 3, pp. 24–35 (2018).

[5] Hasselbring, W. and Steinacker, G.: Microservice architectures for scalability, agility and reliability in e-commerce, *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, IEEE, pp. 243–246 (2017).

[6] Mohan, C., Strong, R. and Finkelstein, S.: Method for distributed transaction commit and recovery using Byzantine agreement within clusters of processors, *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pp. 89–103 (1983).

[7] Maddodi, G., Jansen, S. and Overeem, M.: Aggregate architecture simulation in event-sourcing applications using layered queuing networks, *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pp. 238–245 (2020).

[8] Štefanko, M., Chaloupka, O., Rossi, B., van Sinderen, M. and Maciaszek, L.: The Saga pattern in a reactive microservices environment, *Proc. 14th Int. Conf. Softw. Technologies (ICSOFT 2019)*, SciTePress, pp. 483–490 (2019).

[9] Rudrabhatla, C. K.: Comparison of event choreography and orchestration techniques in microservice architecture, *International Journal of Advanced Computer Science and Applications*, Vol. 9, No. 8, pp. 18–22 (2018).

[10] Mohammed, S. I., Knapp, D. W., Bostwick, D. G., Foster, R. S., Khan, K. N. M., Masferrer, J. L., Woerner, B. M., Snyder, P. W. and Koki, A. T.: Expression of cyclooxygenase-2 (COX-2) in human invasive transitional cell carcinoma (TCC) of the urinary bladder, *Cancer research*, Vol. 59, No. 22, pp. 5647–5650 (1999).

[11] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E. and O'Neil, P.: A critique of ANSI SQL isolation levels, *ACM SIGMOD Record*, Vol. 24, No. 2, pp. 1–10 (1995).

[12] Lampson, B. and Lomet, D.: A new presumed commit optimization for two phase commit (1993).

[13] Fan, P., Liu, J., Yin, W., Wang, H., Chen, X. and Sun, H.: 2PC*: a distributed transaction concurrency control protocol of multi-microservice based on cloud computing platform, *Journal of Cloud Computing*, Vol. 9, No. 1, pp. 1–22 (2020).

[14] Hasselbring, W.: Microservices for scalability: Keynote talk abstract, *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, pp. 133–134 (2016).

[15] Furda, A., Fidge, C., Zimmermann, O., Kelly, W. and Barros, A.: Migrating enterprise legacy source code to microservices: on multitenancy, statefulness, and data consistency, *IEEE Software*, Vol. 35, No. 3, pp. 63–72 (2017).

