マイクロサービスにおける整合性担保の時間を削減する方法

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におけるトランザクション処理に TCC パターンと Saga パターンがある．2つはそれぞれ問題点がある．TCC パターンは, 正常時に正しいデータを取得できるまでの速度が遅い．saga パターンは, ロールバック発生時に正しいデータを取得できる速度が遅い．本稿では, 基本は速度の早い Saga パターンを使用する．本稿で提案する akira は, ロールバック発生時に一定時間正しいデータを取れなくなった場合, そのサービスを TCC パターンにすることでどちらか一方のトランザクショ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, ネットワーク以外のロールバック発生時に正しいデータを取得できるまでの時間を計測する．

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]．

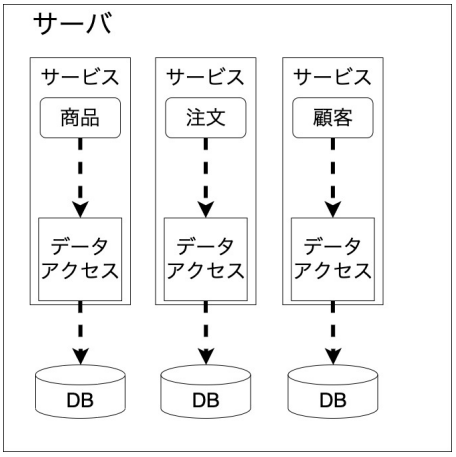

図 1　マイクロサービスアーキテクチャ

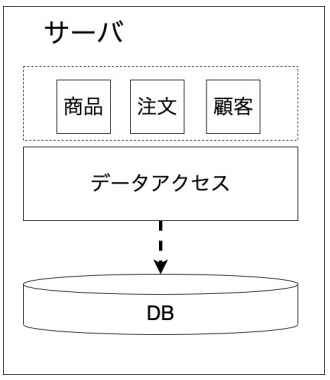

図 2　モノリシックアーキテクチャ

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”

[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
TUT, Hachioji, Tokyo 192–0982, Japan

(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.

1.3 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]. サービスどうし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 パターンの型として, Choreography 型と Orchestration 型の二つがある. Choreography 型は, 各サービスが互いの実行結果に応じて協調して処理を実行し, 最終的にワークフローを完成させるものである. 単純で小規模なシステムに向いている [9]. Orchestration 型はオーケストレータが saga の進行を管理しており, 複雑で大規模なワークフローを構成するシステムに向いている [9]. 図 3 に Choreography 型, 図 4 に Orchestration 型を示す. 図 3 は, 商品サービスでの処理が終わったら次のサービスである注文サービスを呼び出している.

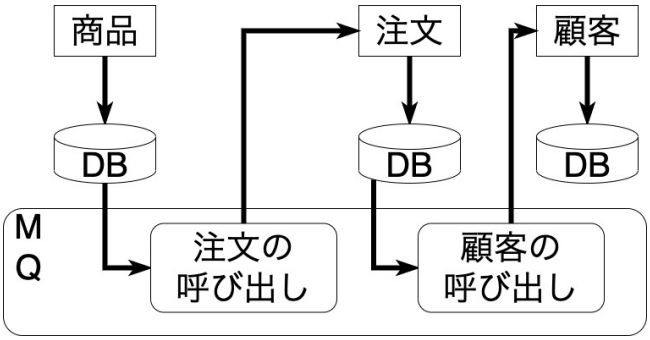

図 3 Choreography 型

商品では, 商品の表示を行っている. 注文では, 在庫の管理を行っている. 顧客では, 支払いを行っている. 図 4 は, コーディネーターを通してサービスが呼び出される. コーディネーターは, すべてのサービスに対して呼び出しを行う. すべてのサービスで正常に処理が終わり結果が返ってきたらコーディネーターのもつデータベースにすべての結果を書き込む. 返ってこない場合は, キャンセル処理を行なう. この２つの大きな違いは, コーディネーターがいるかどうかである. Choreography 型は前のサービスが次のサービスを呼び出すのに対し, Orchestration 型はコーディネーターがすべてのサービスに対して呼び出しを送っている.

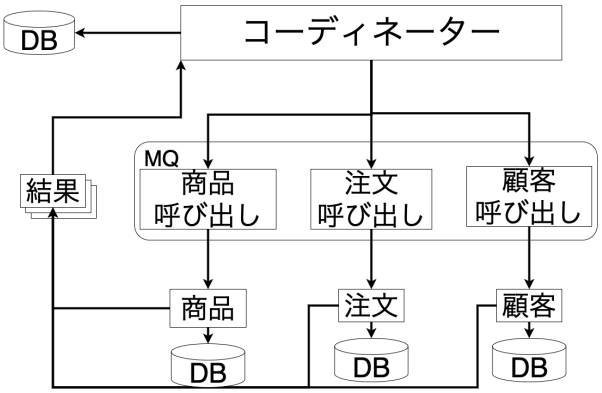

図 4 Orchestration 型

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].

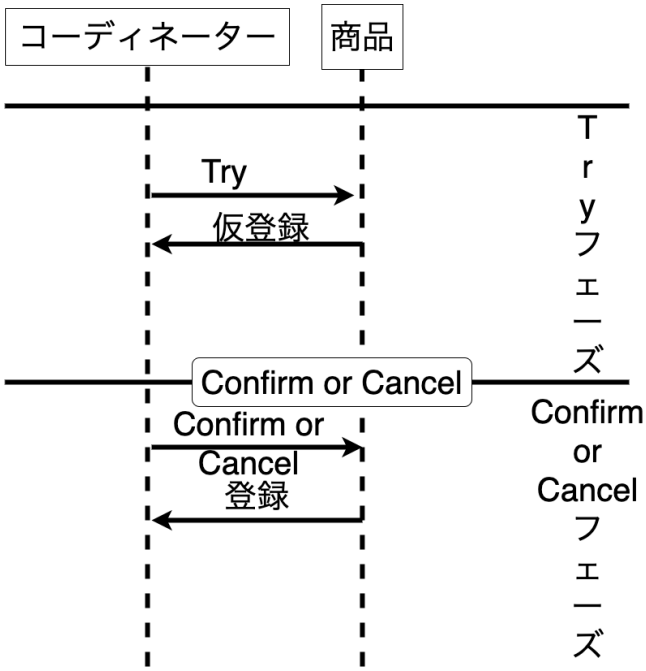

図 5 TCC パターン

Try フェーズと Confirm or Cancel の２フェーズによってトランザクションを行っている. Try フェーズでは, コーディネーターが各サービスに仮状態を登録する. Confirm or Cancel のフェーズでは, サービスの仮登録がすべて完了したら正式に登録し, 処理が終わったことを呼び出し元に返答する. コーディネーターは, Try フェーズで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

はキャンセル要求を送る．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．図 5 に EC サイトを例とした TCC パターンのトランザクションフローについて示す．Try フェーズでは, コーディネーターは商品サービスに対して Try を行う．商品サービスは, 仮登録を行う．他のサービスも同様である．Confirm or Cancel フェーズでは, 商品, 注文, 顧客の各サービスで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．その旨をコーディネーターに返す．

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]．Saga と TCC の比較を図 6 に表す．TCC は通常時, 仮決定の後正式な書き込みを行うため処理完了まで 2 倍の時間がかかる．saga は, 通常時は最初から正式な書き込みを行うため早い．しかし, ロールバック発生時は, 取り消し処理を順番に行うためどちらか一方を使用した場合だと遅くなる．

	正常時の速度	ロールバック発生時の速度
TCC	遅い	早い
saga	早い	遅い

図 6 saga と TCC の比較

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]. READUNCOMMITTED 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 1 にトランザクションの分離レベルについて示す．表の○は問題が発生する．X は発生しない．

表 1 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	○	○	○
READCOMMITTED	X	○	○
REPEATABLEREAD	X	X	○
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, 調整者と参加者に分ける [12]．調整者は, 参加者に向かってコミットしてもよいかと聞く．可能であればコミットを行う．成功した場合は, 成功と返答する．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のを待つ．参加者の中に, 失敗が場合調整者は, ロールバックせよと命令を送る．Butler らは Two-Phase Commit(以下 2PC) 提案している．2PC は, 調整者に障害が発生した場合トランザクションを完了できなくなる．参加者からの返答がないと次の動作ができてない点から速度は遅い．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く向いていない．Pan Fan らの研究では, 2PC を使うことで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, トランザクションプロセス中にロックを保持する．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を 3.3 倍、レイテンシを 67 %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こる．Wilhelm Hasselbring の研究では, マイクロサービスにおける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を行っている．故障を素早く検知し, 復旧させている．復旧は可能であれば自動で行えるようになっている．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ている [15]．トランザクション中は, 書き込みロックをかけることで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できてしまう．読み取りができると虚偽のデータを取得してしまう．また, ロックを掛けることで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, マイクロサービスにおけるトランザクションとして既存手法である TCC パターンと saga パターンのハイブリットである akira を提案する．正常時の処理を saga で行い, ロールバック処理が発生した際に他サービスへの

影響が大きい DB を持っているサービスを TCC パターンに一定期間変える仕組みとなっている．他にも, TCC パターンや Saga パターンの関係なしにロールバック発生時に順番にキャンセル要求を行なうのではなく一斉に要求を行なう．

提案方式

図 7 に一斉にキャンセル要求を行なう提案を示す．図 7 の場合, service3 でロールバック処理が発生している．通常の Saga パターンならば service3 が service2 にキャンセル要求を送り, service2 はキャンセル処理終了後 service1 に対しキャンセル要求を送るトランザクションフローになっている．本稿の提案では, アラーム機構を用意する．アラーム機構は, 処理の流れをトレースしている．トレースすることで, 図 7 のように service3 でエラーが発生した場合その前の service1,2 に対し一斉にキャンセル要求を送る．アラーム機構は, トランザクションに付与された ID を用いてどのサービスで処理が行われたかを把握している．

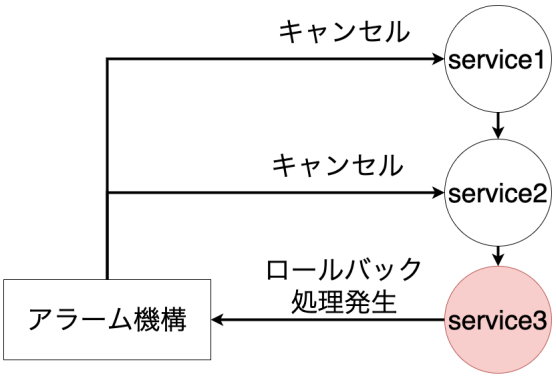

図 7　キャンセル機構

図 8 に Saga パターンと TCC パターンを切り替えるフローチャートを示す．ロールバック中に取得した値を虚偽のデータと定義する．閾値を設定し, 超えたものを TCC パターンに変更する．閾値は, すべてを Saga パターンで動作させたときに DB アクセスが多いサービスとアクセスの少ないサービスを比較した際の虚偽のデータ取得率をもとに決定する．DB アクセスが多いということは虚偽のデータを取得する可能性が高い．

図 8 に切り替え機構を示す．ユーザからアクセスが来るたびに, X の値を 1 プラスする．X の値が閾値を超えていた場合に, TCC パターンに変更する．ロールバック処理が必要であれば Z の値を 1 プラスした後, ユーザへ結果の応答をする．必要ない場合は, ユーザへ結果の応答をする．

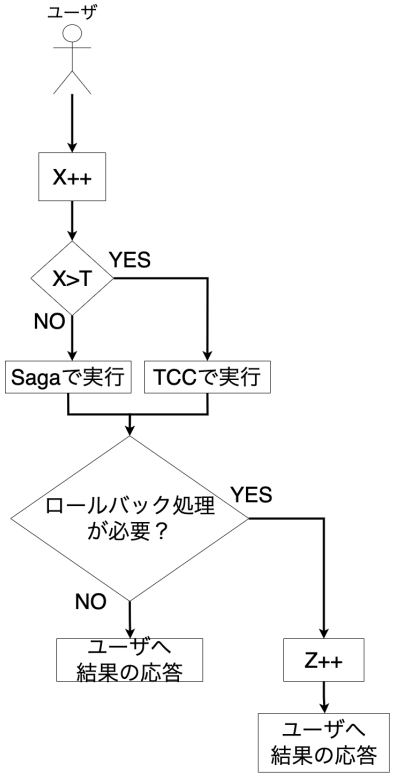

図 8　切り替え機構

図 9 に閾値決定のアルゴリズムを示す．本稿では, 経済産業省が発行している SaaS 向け SLA ガイドラインに基づいて SLA を 99.9 %に設定する．そのため計画サービス時間を 24 時間に, 停止時間を 1.43 分に設定した [16]．DB アクセス数を X と置き, ロールバック中に取得した間違った値の数を Z とする．0.001 は, SLA が 99.9 %に設定し足すことで 100 %になる値．アクセスがない状態はカウントされていないことから, 初期値として X = 0, Z = 0 と置く．以下の式を満たす X の値を閾値とする．

$$\frac{Z}{X} > 0.001$$

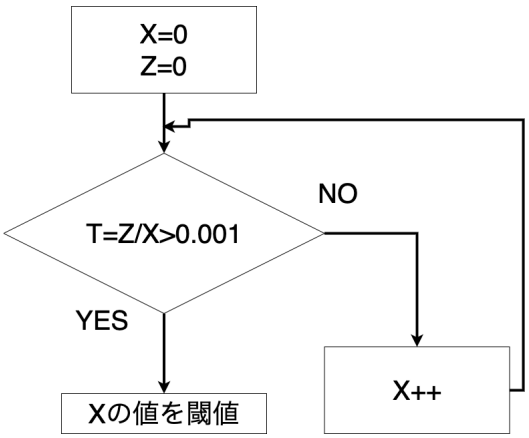

図 9　閾値決定アルゴリズム

TCC パターンから Saga パターンに戻す方法として, 1 分 43 秒毎に閾値の決定アルゴリズムを動かす．

ユースケース・シナリオ

今回設定したユースケースシナリオは, オンラインピザオーダーシステムである. 在庫を設定することで, 一つのデータに多数のユーザからアクセスが来る状況を作り出し整合性の取れる速度の重要度を上げる. 図 10 にピザオーダーシステムに流れを示す. ユーザは, 商品選択をした後注文を確定する. 最後に決済を行なうようなユースケースである.

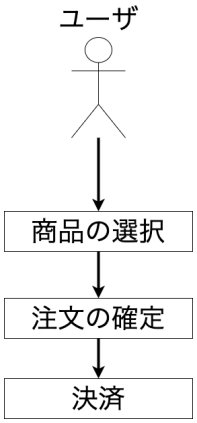

図 10 ピザのオーダーシステム

4. 実装と実験方法

プログラムは, python3.7 でコーディングする. 今回は, プログラムを細かく分割してトランザクションを再現した. 図 10 の場合, 決済が終わった後に在庫がなくキャンセルする処理を行なうことで整合性を取る必要がある場合を再現している.

実装

図 11 にキャンセル機構のフローチャートを示す. ロールバック処理が発生したかを確認する. 発生している場合は, トレース ID から場所を特定する. 前に行った処理にキャンセル要求を送る.

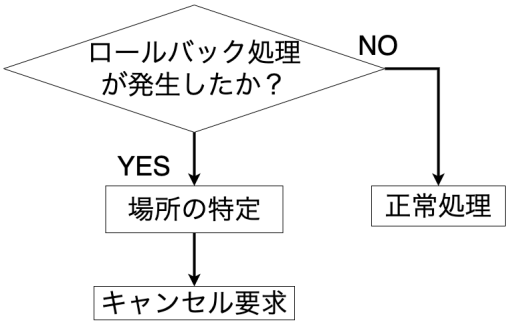

図 11 キャンセル機構

図 12 にシステムの概要を示す. main.py はメニューの選択が終了したらファイルへの書き込みを行う. 最終的に注文を確定かキャンセルを選択する. 確定の場合は, すでに書き込みが行われている. キャンセルは, ファイルに追記された部分を削除する処理をしている. ファイルに注文の確定前に書き込むことで saga パターンを再現している. TCC パターンは, 確定した後に書き込むことで再現している.

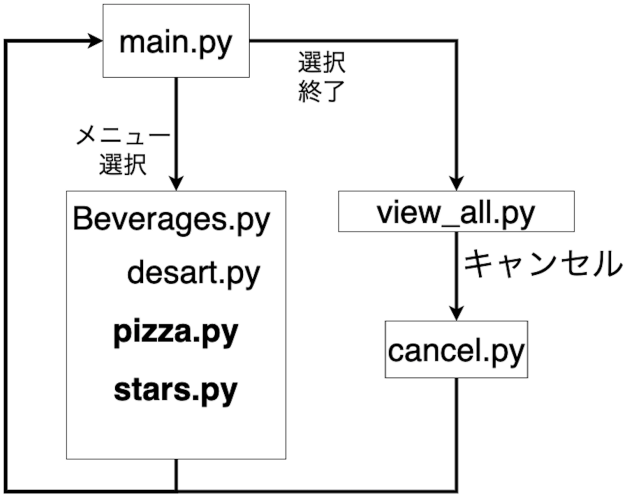

図 12 ピザのオーダーシステム概要

実験環境

図 13 に実験環境を示す. Ubuntu 上に microk8s を構成している. サービスは 4 個存在しておりそれぞれがデータベースを持っている.

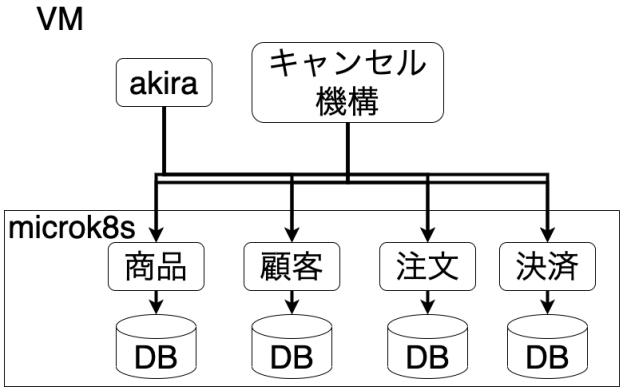

図 13 実験環境

5. 評価手法と分析手法

図 14 に評価方法を示す. 通常時は, 要求が来てから結果が返ってくるまでの時間を計測する. ロールバック発生時は, 要求が来てから正常なデータに戻るまでの時間を計測する. この 2 つを既存手法である TCC パターンと Saga パターン両方の通常時, ロールバック発生時の時間と比較する. DB アクセス数と比べてどのような関係にあるか, 時間のかかり方を分析する.

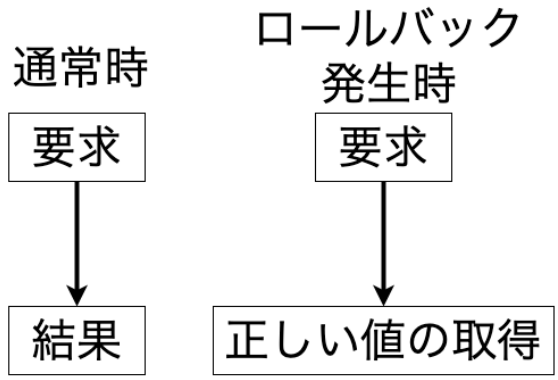

図 14 評価方法

6. 議論

今回は, プログラムの作成を行った. これからは, 閾値の設定とキャンセル機構の作成を行なう. さらに, プログラムの細分化も行なう. 本稿では, SLA を 99.9 %に設定したシナリオだが%を変えたときどのような影響が出るのかも調べる. SLA を変えることで, TCC パターンから Saga パターンに戻すタイミングも変化するためである. 閾値を条件を満たす際の DB アクセス数としたが他の値も試してみて一番データの一貫性の確保が素早い値を探す. また, SLA をもとに閾値を定めたが他の方法もないか模索して見る不調がある. キャンセル機構に障害が発生した際は, Saga パターンの方式でキャンセル要求を送る. 今までのトランザクションを解析することで, どのようなリクエストが閾値を超えるか推定できるようになる.

7. おわりに

本稿では, マイクロサービスにおける代表的なトランザクション手法である TCC パターンと Saga パターンを切り替えながら使用することで整合性担保までの時間を削減する提案をした. SLA を定め, 閾値をもとに Saga パターンから TCC パターンに変更することでロールバックにかかる時間を削減する. 時間を削減することで間違った値を取得することが減る. マイクロサービスを用いた決済サービス, EC サイト運営の際必要になってくる.

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).

[16] 経済産業省：SaaS 向け SLA ガイドライン. `https://www.meti.go.jp/policy/netsecurity/secdoc/contents/downloadfils/080121saasgl.pdf`.