マイクロサービスアーキテクチャ向け データ整合性維持技術の提案と評価

大越　淳平†

† 株式会社 日立製作所 研究開発グループ　〒 185–8601 東京都国分寺市東恋ヶ窪一丁目 280 番地
E-mail: †jumpei.okoshi.pc@hitachi.com

あらまし　近年，独立した小さなサービスの集合によりシステムを構成するマイクロサービスと呼ばれる設計手法が注目されている。一方，マイクロサービスを採用したシステムにおいては，複数のサービスに跨る決済処理など，サービス間においてデータの整合性維持が必要となる場合があり，この実装に多大な工数を要する点が問題となっている。本研究では，データ整合性維持に関わる技術群を体系化し，当該技術群を備えた分散トランザクション管理フレームワークを提案する。EC サービスを想定した評価により，開発工数の 51.2%の削減を確認し，マイクロサービス環境下におけるデータ整合性維持の省力化への寄与を確認した。

キーワード　マイクロサービス，データ整合性，分散トランザクション，Saga

1　はじめに

近年，独立した小さなサービスの集合によりシステムを構成するマイクロサービス，もしくはマイクロサービスアーキテクチャと呼ばれる設計手法が注目されている [1]。マイクロサービスにおいては，アプリケーションは独立した複数のサービスとして構築され，各サービスは RESTful API（Representational state transfer application interface）に代表される API を経由して情報の送受信を行う。単一のコンポーネントとしてアプリケーションを構築する従来のモノリシックアーキテクチャと比較し，スケーラビリティやアジリティに優れるとされ，変化の速い近年の IT サービスにおいて頻繁に採用されている。

一方，マイクロサービスを採用したシステムにおいては，複数のサービスに跨る決済処理など，サービス間においてデータの整合性維持が必要となる場合があり，この実装に多大な工数を要する点が問題となっている。Stack Overflow[1] コミュニティを対象とした調査研究 [2] によると，調査対象者の 32.8% がマイクロサービス環境下における複雑な分散トランザクション（Complex distributed transaction）を最も重要な問題と回答している。

マイクロサービス環境下における分散トランザクションの難しさは，サービスの独立性を重視する設計思想，サービスごとにデータベースを有する Database per service と呼ばれるアーキテクチャ，当該環境で実行される長期間に渡るビジネストランザクションの存在など複数の要因に起因する。これらの要因により，従来の 2PC（Two-phase commit）に代表されるプロトコルに基づいた分散トランザクション手法は，可用性，スケーラビリティ，NoSQL やメッセージブローカーなどの現代的な技術の採用において適用が難しい場合も多い [3]。これに対し，Saga [4] に着想を得たデザインパターンとして，Saga パターン（以下，単に Saga）が知られている [3], [6]。Saga は，各サービスにおけるローカルトランザクションのシーケンス制御によりビジネストランザクションを実現するアプローチであり，一般に資源のロックを取らない [2]ため前述のマイクロサービス特有の環境に好適とされる [3]。

しかしながら，Saga は，なんらかの理由（典型的にはノードやネットワークの障害）でローカルトランザクションが失敗した場合に原子性を維持できなかったり，Saga 全体を一つのトランザクションとみなした場合，Read uncommitted 相当の分離レベルとなり，当該分離レベルで発生するデータ整合性に関する異常である各種アノマリー（典型的には Dirty read）が発生し得るなど，データ整合性維持の観点で複数の問題を有する。このため，前述の分離レベルを前提としたシステム設計や，Saga の実行後に状態やデータの突き合わせ処理によりデータの不整合の検知と修正を行う Reconcile と呼ばれる処理，Reconcile での検知や修正が難しいデータ不整合への運用での対処が必要となるが，確立されたフレームワークが存在せず，工数を要する点が問題となっている。

以上の背景を鑑み，本研究では，Saga や Reconcile などデータ整合性維持に必要な技術群を体系として整理し，これらを備えたアプリケーションフレームワークとして，分散トランザクション管理フレームワークを提案する。また，プロトタイプ実装を用いた EC サービスを想定した評価により，マイクロサービスの開発工数の削減に対する寄与を評価する。

本稿の構成は次の通りである。2 章において関連研究について述べる。3 章においてデータ整合性維持に関わる技術の体系と提案手法について述べる。4 章において提案手法の評価について述べる。5 章においてまとめと今後の課題について述べる。

1：https://stackoverflow.com/

2：実装によりセマンティックロックと呼ばれるアプリケーションによる資源のロック手法が知られる。

2 関連研究

マイクロサービス環境下で Saga をサポートしたフレームワークとして複数の仕様や実装が提案されている [5]。例えば，MicroProfile LRA（Long Running Action）[3]は，Eclipse Foundation[4]管理下で開発が進められているマイクロサービス向けの Java の仕様の一つであり，Saga に代表される長期間に渡る処理を管理するための API を規定している。同報告 [5] によれば，フレームワークごとに単一障害点の有無などに実装の差があるものの，非同期処理を特徴とした現代的なマイクロサービスアーキテクチャに好適なフレームワーク群と結論している。

一方，Saga における問題として，ACID（Atomicity, Consistency, Isolation, Durability）特性の分離性（Isolation）の欠落による，Dirty read に代表される各種アノマリーの存在が知られる [6]。加えて，ローカルトランザクションの制御を担う調停者に該当するサービス（調停者サービス）が単一障害点となり，当該調停者サービスが故障で停止した場合，原子性（Atomicity）が維持されない可能性もある。これらの問題に対し，結果整合（Eventual consistency）[7] やカウンターメジャー [6] が存在する。結果整合は，あるシステム内において一時的なデータ不整合の発生を許容しつつ，更新や障害がない場合に最終的にデータ不整合が解消されることを保証するデータ整合性モデルである。当該データ整合性モデルを前提に各マイクロサービスを開発することにより，ある Saga で発生したデータ不整合が他のサービスに波及することが抑制される。また，カウンターメジャーは，セマンティックロックに代表される各種アノマリーの防止手段であり，結果整合を適用できない領域で必要とされる。

しかしながら，障害や業務上のエラーなどによって発生した原子性に関わるデータ不整合については依然として問題が残っており，これらのデータ不整合を検知し修正する Reconcile と呼ばれる処理が必要となる（実践としては [9] が詳しい）。また，Saga や Reconcile に関わるサービスが複数存在した場合，システム全体でのデータ整合性維持のため，これらのサービスをシームレスに連携させる必要も生じる。

3 提案手法

3.1 技術の体系

前述の通り，マイクロサービス環境下でのデータ整合性維持においては，単に Saga を実行する以外にも，Reconcile の実行や Saga と Reconcile の連携など包括的な枠組みが必要となる。本研究では，システム全体でのデータ整合性維持に向け，データ整合性維持に関わる技術群を次の通り体系化した。

実行時保証　Saga パターンや Saga の実行時のデータ整合性維持に必要なデザインパターンの集合である。詳細は，既報 [8] にて報告済みである。

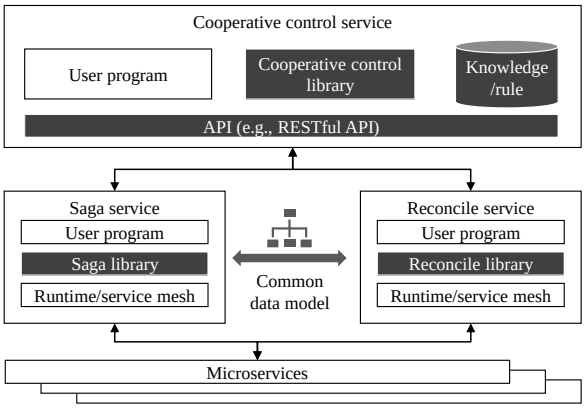

図 1　分散トランザクション管理フレームワーク

実行後保証　Saga の実行で発生したデータ不整合の検知と修正処理である Reconcile に関わる技術である。本稿にて詳細を報告する。

実行時・実行後保証連携　Saga と Reconcile の連携に関わる技術である。前述の通り，マイクロサービス環境下においては，各サービスは独立したサービスとして構築されるため，Saga や Reconcile に関わるサービスが複数存在する場合，疎結合性を維持しつつ，データ不整合を速やかに検知し修正するシームレスなサービス連携が必要となる。

3.2 提案フレームワーク概要

本研究では，前節で述べた技術体系を鑑み，以下の特徴を有した分散トランザクション管理フレームワークを提案する（図 1）。なお，本フレームワークとしては，前述の技術体系や技術体系に対応する各種ライブラリ，各ライブラリで横断利用可能な共通データモデル，連携制御方式など，マイクロサービス環境化でのデータ整合性維持に必要な考え方，アルゴリズム，データモデルの包括的な提供を対象とする。

共通データモデル　Saga や Reconcile など複数のサービスに跨ってビジネストランザクションを管理する場合，共通して利用可能な共通データモデルの提供が必要となる。本共通データモデルは，図 1 の Common data model に対応し，前述の 3 つの技術体系の全てで横断的に活用される。

Saga 実行管理　図 1 の Saga library，及び前述の技術体系の実行時保証に対応し，Saga の実行に必要なフロー制御や状態管理を担う。詳細は，既報 [8] にて報告済みである。

Reconcile アルゴリズム　共通データモデルに基づき，データ不整合の検知と修正によりデータ不整合を解消する Reconcile アルゴリズムを備える。図 1 の Reconcile library，及び技術体系の実行後保証に対応する。

連携制御　Saga や Reconcile など異種のサービスの連携処理はサービス同士の依存性を高め，サービスの独立性やアジリティに悪影響を生じさせる可能性がある。本フレームワークでは，図 1 の Cooperative control service 内の各コンポーネント（Cooperative control library 等）に対応し，各サービスの独立性とシームレスな連携を両立する異種サービスの連携制御を

3：https://microprofile.io/project/eclipse/microprofile-lra
4：https://www.eclipse.org/org/foundation/

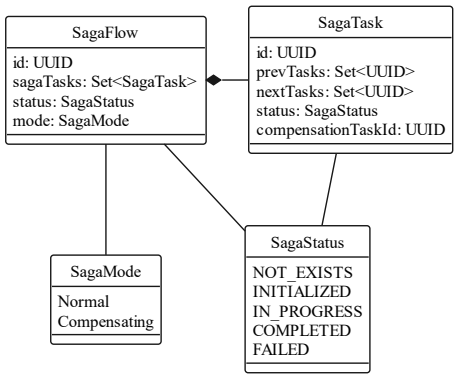

図 2　共通データモデル

Algorithm 1 Reconcile

Input: sagaFlow, participants, rule
Output: operation

```
1: if sagaFlow.getStatus() == COMPLETED then
2:     return null
3: end if
4: for all sagaTask ← sagaFlow.getSagaTasks() do
5:     if sagaTask.getStatus() == IN_PROGRESS then
6:         participant ← participants.getParticipant(sagaTask)
7:         participantStatus ← participant.getStatus(sagaTask)
8:         sagaTask.updateStatus(participantStatus)
9:     end if
10: end for
11: sagaFlow.updateStatus()
12: operations ← getOperations(rule, sagaFlow)
13: operation ← operations.getHighestPriorityOperation()
14: return operation
```

提供する。

以下，既報 [8] にて報告済みの Saga 実行管理を除いた各特徴の詳細について述べる。

3.3　共通データモデル

図 2 に提案する共通データモデルを示す。共通データモデルは，Saga フロー全体を管理する SagaFlow と Saga フロー内で実行されるローカルトランザクションに対応する SagaTask からなる。また，Saga フローやローカルトランザクションの状態管理のため，SagaMode と SagaStatus が用いられる。まず，既報 [8] の Saga ライブラリ内にて当該データモデルを用いた Saga を実行する。その後，何らかの理由により途中で問題が発生（複数の SagaTask の状態が FAILED となり，最終的に SagaFlow の状態が FAILED となったなど）した場合，当該データモデルに基づいた Saga の実行結果を Reconcile ライブラリに伝搬させることでデータ不整合の検知と修正を行う。

3.4　Reconcile アルゴリズム

Algorithm 1 に Reconcile アルゴリズムを示す。当該アルゴリズムは，図 2 に示す共通データモデルからなる sagaFlow，参加サービス群を示す participants，及び Reconcile の操作を定める rule を入力とし，Reconcile の操作である operation を出力する。以下に処理の詳細を述べる。

1–3 行目の処理のガード節により完了済みの SagaFlow が除外される。なお，Reconcile の主な処理対象は，INITIALIZED，IN_PROGRESS，FAILED の途中状態となった SagaFlow 群である。

4–11 行目の処理により，SagaFlow に含まれる SagaTask のうち，IN_PROGRESS，すなわち進行中だが何らかの理由（典型的にはネットワーク障害など）により停止している SagaTask に対し，状態の確認及び修正を行う。具体的には，6 行目の処理により参加サービス群の中から当該 SagaTask に該当する参加サービスを抽出する。さらに，7 行目の処理により当該 SagaTask に該当する参加サービスのローカルトランザクションの処理結果を取得し，8 行目の処理により状態を更新（成功なら COMPLETED，失敗なら FAILED など）する。最後に，11 行目の処理により，4–10 行目の処理の SagaTask の更新を踏まえて，SagaFlow 全体の状態を更新する。

12–14 行目の処理により，rule と sagaFlow の状態から Reconcile の処理内容を決定し，operation を出力する。まず，12 行目の処理において rule と sagaFlow から Reconcile 処理の候補の一覧を取得する。rule は，SagaFlow の状態，Reconcile 処理，Reconcile 処理の優先度のリストからなり，入力された SagaFlow の状態に応じて，適用可能な Reconcile 処理を operations として優先度付きで出力する。具体的な Reconcile 処理としては，途中で停止した SagaFlow を前進させる前進復帰処理や SagaFlow を後進させる後進復帰処理，もしくは自動での復旧が難しい場合に運用にて対処する処理などが該当する。13，14 行目の処理において，候補となる処理の中から最も優先度の高い処理を抽出し，結果として返す。

アルゴリズムの拡張として，SagaFlow の状態に図 2 で記載した SagaStatus の状態に加え，前進復帰や後進復帰のタイムアウトや補償トランザクションの有無による後進復帰可否などを含めたり，13 行目の処理の決定に過去の履歴を蓄積し Saga の回復処理と Reconcile の反復処理を可能とするなどが挙げられる。

3.5　連携制御

図 1 を用いて連携制御について説明する。サービスの疎結合性とシームレスな連携を両立するため，複数の異種サービスの連携を担う独立した協調制御サービスにより当該連携処理を実現する。具体的には，BRMS（Business rule management system）の考え方をもとに，異種サービスの連携に必要な連携ルールや知識を当該協調制御サービスにて管理するとともに，各サービス向けに連携 API（プロトタイプ実装では RESTful API）を提供する。各サービスは連携処理が必要になった場合（Saga サービスで Saga の実行に失敗しデータ不整合が発生したなど），協調制御サービスに問い合わせを行うことで，連携に必要なデータや連携ルール，処理部分を各サービスから独立した形で管理，及び機能させることが可能となる。

図 3 に連携制御で用いられる連携ルールのデータ構造を，図 4 にプロトタイプでの記述例を示す。基本的な考え方は，実行主

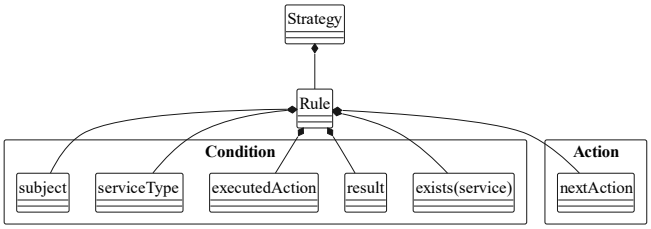

図 3　連携ルール

RuleSet　com.hitachi.dtmfw.sample.serviceOrchestrationService
Unit　SagaServiceUnit
Import　com.hitachi.dtmfw.sample.serviceOrchestrationService.SagaServiceApplication
Queries　query SagaStrategy $l: /sagaServiceApplications end
Notes

RuleTable SagaServiceRule

CONDITION	CONDITION	CONDITION	CONDITION	CONDITION	ACTION
$l: /sagaServiceApplications					
subject	serviceType	exists(reconcileService)==$param	executedAction	result	modify($l) { setNextAction($param) };
subject	**serviceType**	**reconcileService**	**executedAction**	**result**	**nextAction?**
"SagaService1"	"Saga"	TRUE	"ExecSagaFlow"	"FailedInNormal"	"RequestReconcile"
"SagaService1"	"Saga"	TRUE	"ExecSagaFlow"	"FailedInCompensation"	"RequestReconcile"

図 4　プロトタイプにおける記述例

図 5　プロトタイプにおける動作例

体，実行結果，次の実行処理，次の実行処理に必要な各種条件の具備となる。実行主体については，subject 及び serviceType が該当し，サービスの識別子やサービスの種別を記述する。実行結果については，executedAction 及び result が該当し，Saga フローの実行やその結果を記述する。次の実行処理については，nextAction が該当し，Saga フローの実行失敗を受けた Reconcile の実行要求などが記述される。また，その実行に必要な諸条件を exists(service) で記述する（exists() は特定のサービスが存在することを諸条件とした実装例）。

図 5 はプロトタイプ実装における動作の例を示したものである。本例では，Saga サービスは自身の Saga フローの実行に失敗し，当該情報を協調制御サービスに送付する。その後，Reconcile サービスが連携ルールに基づいた Reconcile 処理を実施することで，Saga フローの実行の失敗で生じたデータ不整合を解消する。図 5 上部は，システムに Reconcile サービスが存在しない状況での要求例である。この場合，図 3 における exists(service) の判定結果が False となるため，Saga サービスに対し次の処理（NextAction）の指示が無し（null）となる。一方，システムに Reconcile サービスを展開し，当該サービスが知識として協調制御サービスに登録された状況（図 5 下部）において，再度，問い合わせを実施すると，exists(service) の判定結果が True となり，次の処理（NextAction）に適切な処理（この例では，Reconcile サービスへの処理要求とそのサービスの接続情報）が出力される。以上，本連携制御により，マイクロサービスの疎結合性を維持しつつ複雑なルールに基づいたマイクロサービスの協調制御が可能となる。

4　評　　価

4.1　評価方法

本研究では，3 章で述べた提案手法のプロトタイプ実装を用いて，サービスの開発工数の削減効果を評価した。プロトタイプ実装は，Saga サービス，Reconcile サービス，協調制御サービス，参加サービス 1，及び 2 の計 5 サービスからなる。各サービスは，Quarkus[5]を用いて実装し，EC サービスを想定した次の 3 種のシナリオに対応することを要件とした。なお，協調制御サービスについては，Quarkus に加え Kogito[6]を用いてルールエンジンを実装した。

正常シナリオ　正常に終了するベースラインとなるシナリオである。ユーザは Saga サービス（EC サービスに対応）に対し注文リクエストを発行し，Saga サービスは各参加サービス 1，2（決済サービス，及び在庫管理サービスに対応）に対し決済処理や購入処理を行い，結果をユーザに返す。

補償シナリオ　参加サービス 2（在庫管理サービス）に障害が発生した異常ケースのシナリオである。シナリオ 1 の購入処理において，サービス障害が発生し，購入処理が失敗する。何度かリトライを実施した後，補償処理である決済処理のキャンセルにより，データの整合性を維持する。

リコンサイルシナリオ　補償シナリオにおいて，さらに参加サービス 1（決済サービス）に障害が発生し，決済処理のキャンセルが失敗するシナリオである。この場合，Saga サービス単体ではデータの整合性維持が難しくなり，Reconcile によるデータの整合性維持が必要となる。具体的には，Saga サービスは協調制御サービスに対し，Saga の実行結果を通知し，次の処理内容（Reconcile サービスへの Reconcile 処理依頼）を受け取る。次に，Reconcile サービスに対し Reconcile 処理を依頼し，Reconcile サービスは各参加サービスの状態確認や Saga サービスへの前進回復や後進回復の処理命令によってデータの整合性を維持する。

4.2　評価結果

図 6 に評価結果を示す。アプリケーションフレームワークの提供により，開発工数の 51.2%が削減され，未適用時（Before）に 5.8k であったステップ数が 2.8k まで削減された。なお，ステップ数の算出にあたり，(a) アプリケーションフレームワーク未適用時の開発工数はアプリケーションフレームワークの提

5：https://quarkus.io/
6：https://kogito.kie.org/

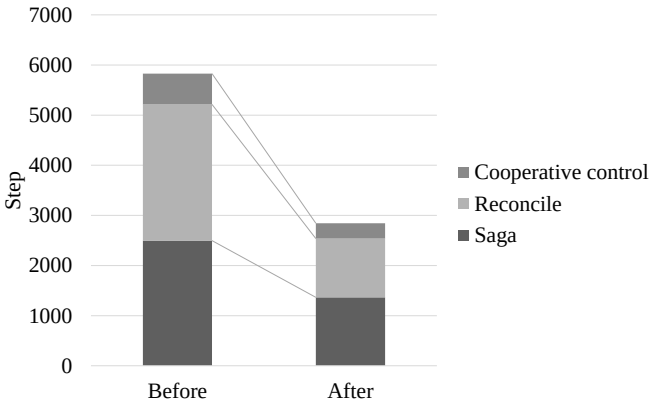

図 6　評 価 結 果

供するライブラリと同等のステップ数とする，(b）参加サービスについては評価外とする，の 2 点を仮定した。

5 結　　論

マイクロサービス環境下におけるデータの整合性維持は重要な問題であり，この解決のため，複数の手法やフレームワークが提案されている。特に，Saga パターンは，当該環境下でサービス間のデータの整合性を維持する手法として広く用いられている。一方で，ACID 特性における分離性（Isolation）の欠如によって各種アノマリーが発生したり，ノードやネットワークの障害により原子性（Atomicity）が維持できなかったりなど，データの整合性維持において問題が残っている。

本研究では，マイクロサービス環境下でのデータ整合性維持に必要な技術群の体系化と当該技術群を備えた分散トランザクション管理フレームワークの提案を行った。また，当該フレームワークの実現に必要な各種アルゴリズムやデータモデルについて検討，及びプロトタイプ実装を行った。評価により，開発工数の 51.2%を削減し，マイクロサービス開発における省力化の観点での有用性を検証した。今後の課題として，2PC や Saga などデータ整合性維持手法の適切な使い分け，既報 [8] や本稿の提案手法で対処できないデータ不整合，特に，結果整合や分離性の欠如に起因するもの，データそのものの突き合わせを必要とするものへの対処などが挙げられる。

文　　献

[1] M. Fowler, “Microservices,” https://martinfowler.com/articles/microservices.html, 2014.
[2] M. Viggiato et al., “Microservices in practice: A survey study,” arXiv, 2018.
[3] K. Dürr et al., “An Evaluation of Saga Pattern Implementation Technologies,” ZEUS, 2021.
[4] H. Garcia-Molina et al., “Sagas,” ACM Sigmod Record, 1987.
[5] M. Štefanko et al., “The saga pattern in a reactive microservices environment,” ICSOFT, 2019.
[6] C. Richardson, “Microservices Patterns,” Manning Publications, 2018.
[7] W. Vogels, “Eventually consistent,” Communications of the ACM, 2009.
[8] 野田昌太郎ら, “マイクロサービスアーキテクチャ向け分散トランザクション管理技術の提案と評価,” FIT, 2021.
[9] Mercari, Inc., “マイクロサービスにおける決済トランザクション管理,” https://engineering.mercari.com/blog/entry/2019-06-07-155849/, 2019.