Vol.2021-DPS-186 No.71
Vol.2021-CSEC-92 No.71
2021/3/16

長期的な孤立運用後にサービスを統合可能なデータ管理手法

大坂 優輝[1]　北形 元[2]　長谷川 剛[2]

概要：ネットワークが分断されるような障害が発生した際，平時にサービスのスナップショットを分散的に保存しておくことで，障害時に孤立したネットワーク毎にサービスを復元・稼働させることができる．しかし，孤立したサービスを長期的に運用した後，障害の復旧時にサービスを 1 つに統合する際，各々で更新されたデータの不整合を避けることは難しい．そこで本稿では，サービスの長期的な孤立運用後にデータを統合する際に生じるデータの不整合について考察し，それらの不整合を回避して統合することができるデータ管理手法を提案する．また，試作システムを用いた実験を通じ，提案手法の有効性を示す．

キーワード：データベース，データ統合，競合回避，耐障害性．

A data management approach that enables services to be integrated after long-term isolated operations

Abstract: In the event of a network disruption, it is possible to restore and run services on each isolated network by storing the services' snapshots in a distributed manner during normal periods. However, it is difficult to avoid the inconsistency of the data updated in each isolated service when the services are merged into one after long-term operation. In this article, we discuss the data inconsistencies that occur when integrating services after long-term isolated operation. Then we propose a data management method that avoids these inconsistencies and enables integration. We show the effectiveness of the proposed method through experiments using a prototype system.

Keywords: Database, Data integration, Conflict avoidance , Resilience.

1. はじめに

日本は地理的に様々な自然災害の発生リスクが大きい国であり，災害時にネットワークの障害が発生する場合がある．例として，東日本大震災の際には，NTT ドコモが運営する携帯電話の基地局が約 6500 局停止し，1000 局を下回る程度に回復するまでに 8 日を要した [1]．このように，ネットワーク上で運用するサービスの開発時には耐障害性の考慮が求められ，1 週間以上の長期的なネットワーク障害時におけるサービスの可用性向上が求められる．

ネットワーク障害発生時におけるサービスの可用性を向上させる手法の 1 つとして，サービスを複製し，各地のサーバーに分散して配備することが考えられる．図 **1** にその際に想定されるシナリオを示す．サービスを提供するサーバーを増やすことで，サーバー間のネットワークが分断され，サーバー間の通信が途絶した場合でも，稼働中のサーバーに接続可能であれば，ユーザーはサービスを利用することができる．本稿ではネットワークが分断された状態で，各々のネットワークパーティションにおいて，長期間にわたってサービスの運用を続けることを，長期的な孤立運用と呼ぶ．ネットワークが分断されている間は，各サーバーで実行された操作は，他のネットワークパーティションに存在するサーバーには伝播しないため，各複製サービスが持つデータベースには差異が生じる．したがって，ネットワークが復旧した後，サービスが持つデータベースを統合すると，統合後のデータベースに不整合が生じる場

[1] 東北大学 大学院情報科学研究科
Graduate School of Information Sciences, Tohoku University
[2] 東北大学 電気通信研究所
Research Institute of Electrical Communication, TohokuUniversity

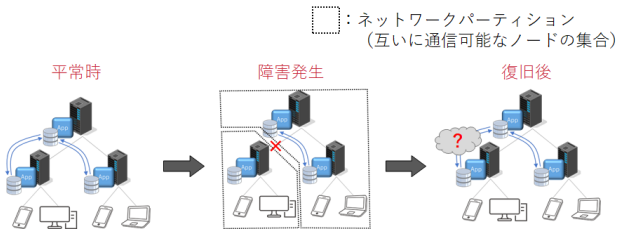

図 1　ネットワークサービスの長期的な孤立運用のシナリオ
Fig. 1　A scenario for the long-term isolated operation of network services

合がある．

不整合を回避してデータベースを統合する手法として，遅延更新複製が知られている [2], [3], [4], [5], [6], [7]．これはローカルサーバーで行った更新を，グローバルサーバーにコミットする前に，更新の競合が発生しないことを確認する手法である．ある 2 つのトランザクションの間に競合が発生する場合には，一方のトランザクションを行ってから，もう一方のトランザクションを行う．それでも競合が解決されない場合は，一方のトランザクションを破棄する．この手法は楽観的アプローチと呼ばれており，競合が少ない，あるいは競合解決が簡単である場合に有効とされる．しかし，長期的な孤立運用の場合，短期のネットワーク分断の場合と比較して，分断中に多数のトランザクションが実行されるため，競合箇所が増加する．また，競合を解決できずにトランザクションを破棄した場合，そのトランザクションを前提としているすべてのトランザクションを破棄しなければならない．したがって，既存研究で提案されている手法をそのまま適用することはできない．

本稿では，まず，上述したシナリオにおいてデータを統合する際に発生する不整合について考察を行い，不整合が発生する条件を明らかにする．次に，それらの不整合を回避する手段を検討する．その際，アプリケーションレベルでのデータの意味を考慮することにより，データベースレベルでの考察だけでは回避することができない不整合を回避することを考える．また，検討結果に基づいて，複製サービスのデータを不整合を発生させずに統合するためのデータ管理機構を提案する．具体的には，ユーザーの操作を記録するコンポーネントをサービスに組み込み，データを統合する際に記録した操作を再現することで，ネットワークの分断が発生しなかった場合に想定される状態を生成する機構を提案する．さらに，提案機構を適用できるサービスとして，安否確認サービスを試作する．この試作サービスに様々なパターンの操作を与えて挙動を確認し，提案機構の有効性を示す．

本稿の構成は以下の通りである．2 章では，データ統合の際に発生する不整合について考察を行う．3 章では，2 章で行った考察をもとに，サービスの長期的な孤立運用後に，データを不整合無く統合可能とするデータ管理機構を提案する．4 章では，提案機構を組み込んだ試作サービスを用い，提案機構の評価を行う．5 章では，本稿のまとめを述べる．

2. データ統合の際の不整合

2.1 不整合の種類

不整合が発生するレベルに着目すると，不整合は大きく 2 つの種類に分類できる．1 つはデータベースレベルの不整合である．これは，トランザクションの ACID 特性のうち，一貫性 (Consistency) が満たされないこと，と定義する．具体的には，トランザクションを同時実行した結果のデータベースの状態と，トランザクションを直列に実行した場合のデータベースの状態に差異が生じることを指す．次にアプリケーションレベルの不整合である．これはトランザクションを実行した結果，意味的に正しいとされるデータベースの状態と，実際のデータベースの状態に差異が生じ，それによりサービスの運用に不都合が生じること，と定義する．

2.2 データベースレベルの不整合

本論文で定義したデータベースレベルの不整合に相当するものとして，一般に，更新喪失，ダーティリード，非再現リード，ファントムリードの 4 つの不整合が知られている [8]．これらの不整合は，トランザクションの同時実行を許していることにより発生する．トランザクションの同時実行には処理効率を高めるメリットがあるが，長期的な孤立運用後の統合においては時間的制約は厳しくないものと考えられる．ゆえに本稿では，データの統合を行う際には，効率より不整合の排除が重要であると考える．したがって，異なる複数のサーバーにおいて同じタイミングに実行された複数のトランザクションがある場合でも，統合の際にそれらの同時には実行しないせず，タイムスタンプやその他の条件によってトランザクションの並び替えを行い，必ずトランザクション 1 つ 1 つを直列に実行する．

2.3 長期的な孤立運用後のデータ統合における問題点

本稿で前提とする，長期的な孤立運用後のデータ統合時に発生する，不整合になりうる現象を以下に示す．これらはすべてデータベースレベルでは不整合を生じないが，アプリケーションレベルでは不整合となる場合がある．

直列的更新喪失

あるサーバーで発生したトランザクションにより行われた更新の内容が，他のサーバーで発生したトランザクションにより行われた更新により上書きされ，先に行われた更新の内容が失われる現象である．図 2 に直列的更新喪失が発生する操作の例を示す．全てのサーバで発生した更新

Vol.2021-DPS-186 No.71
Vol.2021-CSEC-92 No.71
2021/3/16

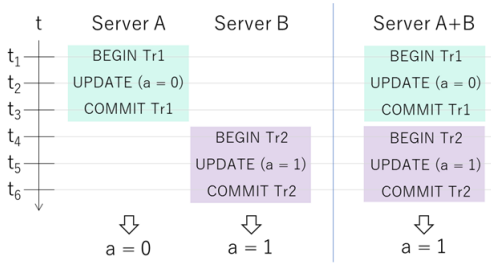

図 2　直列的更新喪失が発生する例
Fig. 2　A example of serial updates loss

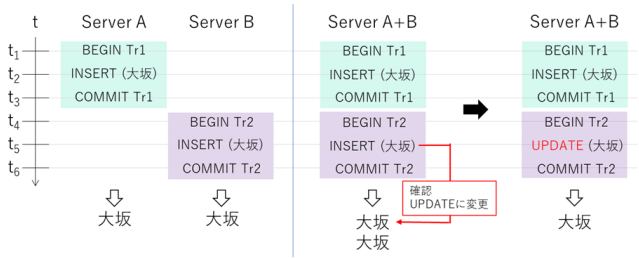

図 3　重複追加が発生する例と解決法
Fig. 3　A example of duplicate additions and a solution

内容のうち，最新のもののみを保持すれば良いアプリケーションや，トランザクションの実行順序が変わっても結果が変わらないようなアプリケーションにおいては，ユーザーの想定通りの挙動になるため，直列的更新喪失は不整合にならない．

一方で，直列的更新喪失が不整合となるようなアプリケーションにおいては，データの意味を考慮しても不整合を解決することはできない．そのため，本稿で提案する機構は適用できない．

重複追加

唯一性のあるデータが，複数のレコードとして INSERT される現象である．図 3 に重複追加が発生する操作の例と解決方法を示す．長期的な孤立運用の場合，複数のサーバーで同じものを指すレコードが INSERT されると，統合する際に，本来唯一であるべきものが複数回 INSERT される．

重複追加が不整合と見なされるか否かは，データの唯一性に依存する．災害時に使用されるサービスを用いて例を挙げると，避難所で生活している人の名簿であれば，記録された人は唯一の存在であり，かつ「生活している」というのは動作ではなく状態を表しており，回数を数えるものではない．したがって，重複追加は不整合となる．一方，避難所への入退記録であれば，記録された人は唯一の存在であるが，「避難所への出入り」という動作は延べ人数で回数を数えるものである．したがって，重複追加は不整合にならない．

重複追加が不整合になる場合，データの意味を考慮することで解決可能である．重複追加が不整合と見なされるアプリケーションでは，重複する INSERT は状態の更新と見なすことができる．したがって，INSERT 操作を行う前に，対象のレコードが既に存在するかどうか調べ，存在する場合は INSERT を UPDATE に置換することで不整合を回避することができる．

虚存更新

DELETE が行われたレコードに対して UPDATE を行う現象である．図 4 に虚存更新が発生する操作の例と解決方法を示す．長期的な孤立運用の場合，1 つのサーバーでレコードの DELETE が行われると，他のサーバーでは当該レコードが残っていたとしても，統合時に当該レコードは DELETE され，他のサーバーでの当該レコードに対する更新はすべて失われる．この現象は，DELETE が行われることが想定されるアプリケーションにおいては不整合になると考えられる．

虚存更新はデータの意味を考慮することで解決可能である．データベースの操作において，DELETE 操作の意味は，修正，移動，消滅，逸失の 4 つが考えられる．それぞれが表すものを説明し，避難所で生活する人の名簿を例にとって，具体例を示す．以下，サービスのユーザーが DELETE を実行したネットワークパーティション内に存在するサーバーを自サーバー，他のネットワークパーティションに存在するサーバーを他サーバーと呼ぶ．

修正は，当該レコードを DELETE した後，もう一度同じものを指すレコードを INSERT しなおすことを表す．自サーバーのデータベースには再び同じものを指すレコードが現れる．例としては，名簿の情報を更新する際に，UPDATE の代わりに行うことが考えられる．

移動は，DELETE されたレコードと同じものを指すレコードが，他サーバーで INSERT されることを表す．他サーバーのデータベースに同じものを指すレコードが現れる．例としては，ある避難所で生活していた人が，別の避難所に移動する場合が考えられる．

消滅は当該レコードが指すものが消滅したことを表し，全サーバーのデータベースにおいて同じものを指すレコードが現れることはない．例としては，避難所で生活していたひとが逝去した場合が考えられる．

逸失は当該レコードが指すものが，データベースに記録される範囲外に移動したことを表し，全サーバーのデータベースにおいて同じものを指すレコードが現れることはない．例としては，避難所で生活していた人が自宅に帰った場合が考えられる．

上述した DELETE 操作の意味のうち，データ統合の際に不整合が発生する可能性について留意する必要があるのは，移動である．移動による DELETE の実行が遅れ，DELETE を行う前に他のサーバーで INSERT が実行された場合，移動による DELETE であるにも関わらず，統合

Vol.2021-DPS-186 No.71
Vol.2021-CSEC-92 No.71
2021/3/16

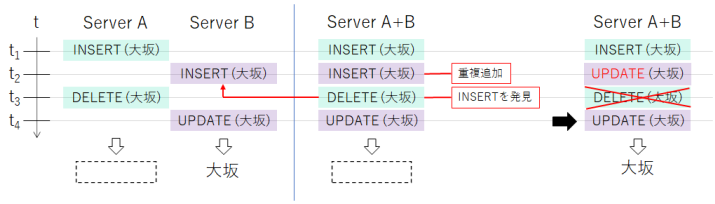

図 4　A example of void record updates and a solution
Fig. 4

後のデータベースから当該レコードが失われる．これを回避するために，DELETE を実行する前に，DELETE が実行された日時より以前に他の全サーバーで実行された，当該レコードと同じものを指すレコードへの書き込み操作について，時間を遡行する方向に検索する．検索の結果，直近に INSERT か UPDATE が実行されているサーバーが存在した場合は，DELETE を実行しないことで，虚存更新を回避することができる．

識別齟齬

データベースにレコードを INSERT するたびに，自動的に 1 ずつ加算される値をシーケンス ID と呼ぶ．シーケンス ID をデータベースのプライマリキーとして用いた場合に，データ統合時にシーケンス ID が変更されることにより，操作を実行すべき対象のレコードが変化する現象である．図 5 に識別齟齬が発生する操作の例と解決方法を示す．データ統合の際には，自サーバーで実行された INSERT と，他サーバーで実行された INSERT によるレコードがデータベースに追加される．したがって，レコードに付与されるシーケンス ID は統合前後で変化する．このため，長期的な孤立運用においてレコードの識別にシーケンス ID を用いると，UPDATE や DELETE の対象が変化する．この現象は，INSERT のみを実行するアプリケーションであれば発生しないが，この現象が発生するアプリケーションでは，確実に不整合になると考えられる．

この不整合が発生する場合，データの意味を考慮することで解決可能な場合がある．レコードがシーケンス ID 以外に唯一性と普遍性のあるフィールドを持つ場合，そのフィールドをプライマリキーとして用い，レコードを識別することで不整合を回避できる．例えば人の名簿の場合，氏名と生年月日のペア等をプライマリキーとすることで，個人を表すレコードを特定することができる．シーケンス ID 以外にプライマリキーとすることができるフィールドを持たないレコードを扱うアプリケーションの場合は，この不整合を回避することができない．そのため，本論文で提案する機構は適用できない．

また，プライマリキーとしたフィールドの唯一性が欠如した場合，不整合が発生する．例えば人の名簿の場合，プライマリキーとして氏名のみを用いると，同姓同名の人を INSERT することができない．レコードが唯一性と普遍性

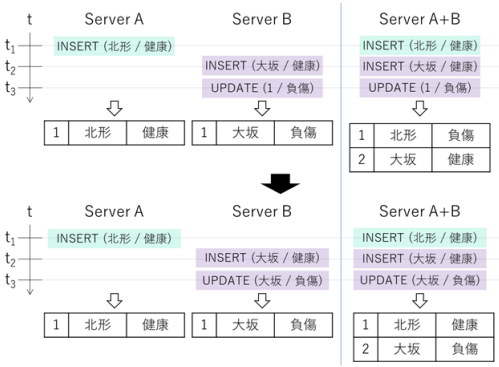

図 5　識別齟齬が発生する例と解決策
Fig. 5　A example of discrepancies in identification and a solution

の高いフィールドを複数持つ場合，それらのフィールドを組み合わせてプライマリキーとすることで，この不整合の発生を抑えることができる．

さらに，プライマリキーとしたフィールドの普遍性が欠如した場合にも，不整合が発生する．複数サーバーで同じものを指すレコードを INSERT する際に，プライマリキーに異なる値を格納した場合，または，一部のサーバーにおいてのみプライマリキーに格納された値を変更した場合に発生する．操作の対象を発見できなくなるため，当該レコードに対する UPDATE や DELETE は，プライマリキーが一致するサーバーで実行されたもののみが反映される．この不整合は，各サーバーにおいてプライマリキーとして最初は同一の値を使用し，一部のサーバーにおいて後からプライマリキーの値を変更した場合であれば解決可能である．データ統合時に，プライマリキーの値を変更する UPDATE を行う際に，他のサーバーで行われた，当該レコードと同じものを指すレコードに対するすべての操作について，対象とするプライマリキーの値を新しい値に置き換えることで，この不整合は回避できる．一方，各サーバーで同じものを指すレコードを INSERT する時点で，異なる値をプライマリキーに格納した場合には不整合を回避できない．そのため，サービスの長期的な孤立運用を行うサービスの開発を行う際に，同じものを指すレコードのプライマリキーには同一の値を格納するよう，サービスのユーザーに対し注意を促すように設計する必要がある．

2.4　統合可能なサービスの条件と統合時に求められる要件

ここまでの考察をもとに，長期的な孤立運用を行った後にデータ統合が可能なサービスの条件と，そのデータ統合を可能とするために統合時に求められる要件をまとめる．

ここでは，長期的な孤立運用を行った後にデータ統合ができないサービスの条件を挙げることで，統合可能なサービスに対する条件を説明する．長期的な孤立運用を行った後にデータ統合ができないサービスの条件は，以下の 3 つ

Vol.2021-DPS-186 No.71
Vol.2021-CSEC-92 No.71
2021/3/16

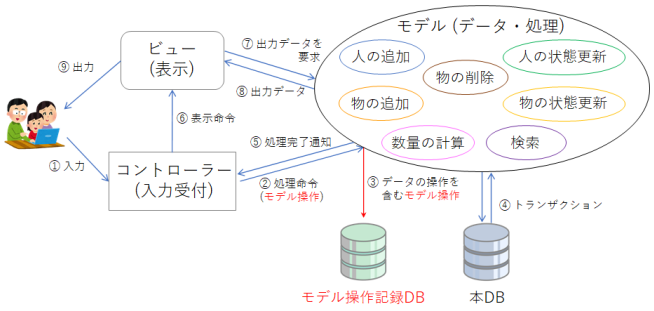

図 6　提案機構の構造
Fig. 6　Structure of the proposed mechanism

にまとめられる．

- 各サーバーのデータをすべて維持する必要があるもの．
- トランザクションの実行順序を変更すると結果が変わるもの．
- 唯一性と普遍性が十分に高いフィールドを持たないレコードを扱うもの．

これらの条件に該当しないサービスであれば，長期的な孤立運用を行った場合も，ネットワーク障害の復旧後にデータを統合することが可能であると考えられる．

次に，データを統合する際に求められる要件を整理すると，以下の 4 つにまとめられる．

- INSERT を実行する前に，対象のレコードが既に存在するかどうか調べ，存在する場合は INSERT を UPDATE に置換する．
- DELETE を実行する際に，DELETE が実行された日時より以前に他の全サーバーで実行された，当該レコードと同じものを指すレコードへの書き込み操作について，時間を遡行する方向に検索し．直近に INSERT か UPDATE が実行されているサーバーが存在した場合には，DELETE を実行しない．
- シーケンス ID 以外の，唯一性と普遍性のあるフィールドをプライマリキーとして用いる．
- プライマリキーの値を変更する UPDATE を行う際に，他のサーバーで行われた，当該レコードと同じものを指すレコードに対するすべての操作について，プライマリキーの値を新しい値に置き換える．

これらの操作が可能となるようにデータ管理機構を設計する必要がある．

3.　データ管理機構の提案

前節でまとめた，データ統合時に求められる要件を考慮し，複製サービスの長期的な孤立運用の後に，不整合を回避してデータを統合することを可能とする，データ管理機構を提案する．

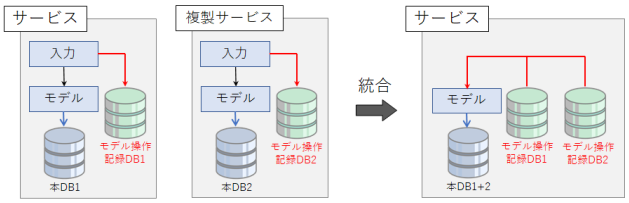

図 7　提案機構を用いたデータの統合
Fig. 7　Integration of data using the proposed mechanism

3.1　提案機構の概要

図 6 に MVC モデルを例にとった提案機構の構造を示す．MVC モデルとは，アプリケーション開発において用いられる設計方法の 1 つである [9]．アプリケーションを，値の処理や一時的な保持を行う Model，ユーザーの入力を受け，情報を表示する View，View から入力を受け取り，Model 内の適切なメソッドに対し実行命令を送る Controller の 3 つのコンポーネントに分けて設計する．

本稿では，MVC モデルにおける，Controller から Model に対して送られる命令を記録するコンポーネントをサービスに組み込むことを提案する．以下，この命令をモデル操作と呼び，モデル操作を記録するコンポーネントをモデル操作記録データベースと呼ぶ．図 7 に提案機構を用いたデータ統合の様子を示す．複製サービスの長期的な孤立運用の後，それらのデータを統合する際には，各複製サービスが持つモデル操作記録データベースからモデル操作を読み出して実行することで，ネットワーク障害が発生しなかったと仮定した際に想定されるモデルの挙動を再現する．

3.2　モデル操作を記録する利点

モデル操作の単位で記録することの利点を述べる．他の手法として，記録して統合に使用できる単位としては，ユーザーが View に対して行った入力と，Model がデータベースに対して行ったトランザクションが考えられる．これらとの比較を行う．

まず，ユーザーが View に対して行った入力を記録した場合，モデル操作を記録する場合と比較して冗長になる．ユーザーが View に対して行う入力の中には，データベースの操作を含まないものが存在する．モデル操作の単位で記録することにより，データベースの操作を含まない入力を Controller で除外して記録することができ，効率的である．

次に，Model がデータベースに対して行ったトランザクションを記録した場合，モデル操作を記録する場合と比較して，データを統合する際に必要となる情報が不足する．前節でまとめたように，サービスの長期的な孤立運用からの統合時には，トランザクションを実行する前に，データの意味を考慮し，条件に合わせて INSERT の代わりに UPDATE を行うなど，トランザクションの内容を変更する必要がある場合がある．トランザクションの単位では

Vol.2021-DPS-186 No.71
Vol.2021-CSEC-92 No.71
2021/3/16

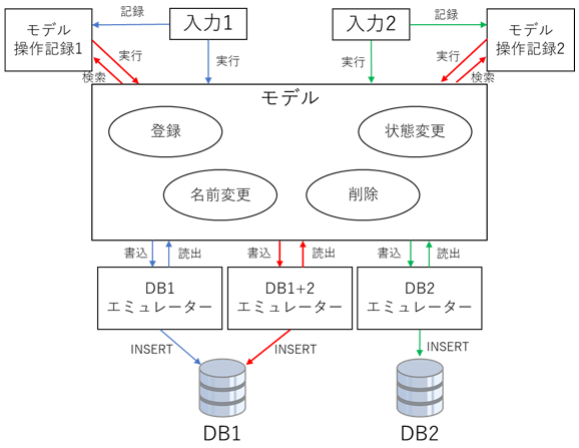

図 8　評価用サービスの設計
Fig. 8　Designing services for evaluation

データの意味を含まないため，統合時にデータの意味を考慮して操作を切り替えることは難しい．モデル操作の単位で記録することにより，ユーザーの操作の意図を含んだ形で記録ができるため，データの意味を考慮した統合が可能となる．

4. 提案機構の評価

本稿で前提としているシナリオでの運用が考えられ，長期的な孤立運用後のデータ統合が可能なサービスの条件に当てはまるサービスとして，安否確認サービスの設計・実装を行った．また，実装したサービスを用いて実際にデータ統合を行い，提案機構の評価を行った．

4.1 評価用サービスの設計

図 8 に評価用の安否確認サービスの設計を示す．入力はモデル操作の形で，プログラムから直接与える．入力をモデルに対して実行すると同時に，モデル操作記録データベースに記録を行う．入力の結果はデータベースのエミュレーター上で一時的に保持し，すべての入力を処理し終えた後に，エミュレーターのデータをデータベースに対し INSERT する．モデル内で行う処理としては，新しい人の登録，人の状態変更，人の名前変更，人の削除の 4 つを実行可能である．統合する際のモデル操作記録データベースの読み込み順序はタイムスタンプ型を採用する．実行可能なプログラムは 3 種類作成する．まず，入力 1 だけを行うプログラムと，入力 2 だけを行うプログラムをそれぞれ作成する．次に，入力 1, 2 を両方行った後に，モデル操作記録 1，モデル操作記録 2 と，データベース 1 エミュレーター，データベース 2 エミュレーターを読み込み，それぞれの情報を踏まえて不整合を回避しつつ統合し，データベース 1+2 エミュレーターを経由して，データベース 1 に出力するプログラムを作成する．

4.2 評価用サービスの実装

まず，実装と実験の条件を述べる．データベースのサーバーとしては，vSphere Client 上の Ubuntu 20.04 LTS 64 bit の仮想マシンを用いた．データベースを操作する言語としては MySQL を用いた．また，プログラムの実装と実行はすべて，Windows10 64 bit で行った．また，プログラムは Eclipse 上で実行した．実装には Java を用いた．

この条件のもとで，3.1.1 項で述べた設計にしたがって，評価用の安否確認サービスの実装を行った．モデル操作記録データベースと，データベースエミュレーターは Java の Arraylist クラスを用いて実装した．また，モデル内で行う処理について，それぞれ，新しい人の登録は Regist，人の状態変更は ChangeStat，人の名前変更は ChangeName，人の削除は Delete として定義した．

4.3 評価

実装した安否確認サービスに対し，重複追加，虚存更新，識別齟齬が発生する操作を与え，統合結果に不整合が生じていないことを確認して，提案機構が有効であることを述べる．

重複追加の回避についての評価

重複追加が発生する操作を与えて統合し，統合の結果において重複追加が回避できていることを確認する．図 9 に入力と結果を示す．入力 1, 2 で発生した操作を時系列順に並べると，以下のとおりである．

(1) 入力 1 において，Sato Ichiro, injured の登録
(2) 入力 1 において，Suzuki Jiro, injured の登録
(3) 入力 2 において，Sato Ichiro, safe の登録
(4) 入力 2 において，Takahashi Saburo, safe の登録

結果を見ると，Sato Ichiro が二度現れることはなく，代わりに Sato Ichiro の状態が変化している．したがって，Sato Ichiro の 2 度目の INSERT が UPDATE に置き換えられ，重複追加を回避して統合できたことが分かる．

虚存更新の回避についての評価

虚存更新が発生する操作を与えて統合し，統合の結果において虚存更新が回避できていることを確認する．図 10 に入力と結果を示す．入力 1, 2 で発生した操作を時系列順に並べると，以下のとおりである．

(1) 入力 1 において，Sato Ichiro, injured の登録
(2) 入力 2 において，Sato Ichiro, injured の登録
(3) 入力 1 において，Sato Ichiro の削除
(4) 入力 2 において，Sato Ichiro, safe の状態変更

結果を見ると，入力 1 による削除は行われず，入力 2 による状態変更が実行されている．したがって，DELETE を行う際に他のモデル操作記録データベースを時間を遡る方向に検索し，INSERT が発見されたことで DELETE を

```
private void addUserInputs1()
{
    userInputs.add(new UserInput(Action.Regist, new User("Sato Ichiro", "injured"), null, 0));
    userInputs.add(new UserInput(Action.Regist, new User("Suzuki Jiro", "injured"), null, 1));
}
private void addUserInputs2()
{
    userInputs.add(new UserInput(Action.Regist, new User("Sato Ichiro", "safe"), null, 2));
    userInputs.add(new UserInput(Action.Regist, new User("Takahashi Saburo", "safe"), null, 3));
}
```

```
+----+-----------------+---------+
| id | name            | status  |
+----+-----------------+---------+
|  1 | Sato Ichiro     | safe    |
|  2 | Suzuki Jiro     | injured |
|  3 | Takahashi Saburo | safe    |
+----+-----------------+---------+
```

図 9　重複追加の回避
Fig. 9　Avoiding duplicate additions

```
private void addUserInputs1()
{
    userInputs.add(new UserInput(Action.Regist, new User("Sato Ichiro", "injured"), null, 0));
    userInputs.add(new UserInput(Action.Delete, new User("Sato Ichiro", null), null, 2));
}
private void addUserInputs2()
{
    userInputs.add(new UserInput(Action.Regist, new User("Sato Ichiro", "injured"), null, 1));
    userInputs.add(new UserInput(Action.ChangeStat, new User("Sato Ichiro", "safe"), null, 3));
}
```

```
+----+-------------+--------+
| id | name        | status |
+----+-------------+--------+
|  1 | Sato Ichiro | safe   |
+----+-------------+--------+
```

図 10　虚存更新の回避
Fig. 10　Avoiding void record updates

キャンセルして，虚存更新を回避して統合できたことが分かる．

識別齟齬の回避についての評価

識別子の普遍性欠如が発生する操作を与えて統合し，統合の結果において識別子の普遍性欠如が回避できていることを確認する．図 11 に入力と結果を示す．入力 1, 2 で発生した操作を時系列順に並べると，以下のとおりである．

(1) 入力 1 において，Sato Ichiro, safe の登録
(2) 入力 1 において，Suzuki Jiro, safe の登録
(3) 入力 2 において，Sato Ichiro, safe の登録
(4) 入力 2 において，Suzuki Jiro, safe の登録
(5) 入力 2 において，Sato Ichiro から Satoh Ichiroh の名前変更
(6) 入力 2 において，Suzuki Jiro から Suzuki Jiroh の名前変更
(7) 入力 2 において，Suzuki Jiroh の削除
(8) 入力 1 において，Sato Ichiro, injured の状態変更
(9) 入力 1 において，Suzuki Jiro の削除

結果を見ると，Sato Ichiro は入力 2 により Satoh Ichiroh に改名され，状態は入力 1 により injured に変更されている．また，Suzuki Jiro は両方で Delete が行われたために削除されている．したがって，一方で名前を変更した場合も，他方での同じものを指すレコードへの操作は問題無く実行されており，識別子の普遍性欠如を回避して統合できたことが分かる．

以上の実験結果から，本稿の提案機構は長期的な孤立運用後のデータ統合に対し有効であることが示された．

5. おわりに

本稿は，ネットワーク分断時に長期的な孤立運用を行った複製サービスを対象とし，分断復旧後にそれらの複製サービスの統合を可能とすることを目的とした．目的の達成のため，本稿ではデータを統合する際に発生する不整合についての考察と，その考察をもとに，長期的な孤立運用後のデータ統合を可能とするデータ管理機構の提案を行った．

まず，データを統合する際に発生する不整合について考察を行った．不整合を，単一のデータベースにおいて発生するものと，長期的な孤立運用後の統合時に特有のものに

```
private void addUserInputs1()
{
    userInputs.add(new UserInput(Action.Regist, new User("Sato Ichiro", "safe"), null, 0));
    userInputs.add(new UserInput(Action.Regist, new User("Suzuki Jiro", "safe"), null, 1));
    userInputs.add(new UserInput(Action.ChangeStat, new User("Sato Ichiro", "injured"), null, 7));
    userInputs.add(new UserInput(Action.Delete, new User("Suzuki Jiro", null), null, 8));
}
```

```
private void addUserInputs2()
{
    userInputs.add(new UserInput(Action.Regist, new User("Sato Ichiro", "safe"), null, 2));
    userInputs.add(new UserInput(Action.Regist, new User("Suzuki Jiro", "safe"), null, 3));
    userInputs.add(new UserInput(Action.ChangeName, new User("Sato Ichiro", null), new User("Satoh Ichiroh", null), 4));
    userInputs.add(new UserInput(Action.ChangeName, new User("Suzuki Jiro", null), new User("Suzuki Jiroh", null), 5));
    userInputs.add(new UserInput(Action.Delete, new User("Suzuki Jiroh", null), null, 6));
}
```

```
+----+---------------+---------+
| id | name          | status  |
+----+---------------+---------+
|  1 | Satoh Ichiroh | injured |
+----+---------------+---------+
```

図 11　識別齟齬の回避
Fig. 11　Avoiding discrepancies in identification

区分した．単一のデータベースにおいて発生する不整合に関しては，トランザクションの同時実行は行わないものとして回避した．一方，長期的な孤立運用後の統合時に特有の不整合に関しては，アプリケーションレベルのデータの意味を考慮することにより不整合を回避することを検討した．

また，不整合についての考察に基づき，複製サービスの長期的な孤立運用後にデータを統合する際，不整合を回避して統合可能であるデータ管理機構を提案した．提案機構について MVC モデルを参考に説明を行い，モデル操作の単位で操作を記録することによる利点を示した．

提案機構を組み込んだサービスが正しく統合されることを示すため，評価用の試作サービスを実装した．不整合の考察で示した，長期的な孤立運用後に統合可能な条件に一致し，かつ実際に本稿で前提とするシナリオで運用されることが考えられることから，試作サービスとして，安否確認サービスを選んだ．この試作サービスに，不整合が発生する操作を与えて結果を確認し，提案機構の有効性を示した．

本稿では，トランザクションはすべて直列で実行するものとした．今後の課題として，トランザクションの同時実行を許可した場合に生じる不整合について考察し，その解決法を検討することが挙げられる．

参考文献

[1] 総務省，“平成 23 年版 情報通信白書 第 1 部 第 1 節 1 (1),” https://www.soumu.go.jp/johotsusintokei/whitepaper/ja/h23/html/nc111100.html. (参照 2020/11/22).
[2] F. Pedone and S. Frolund, “Pronto: A fast failover protocol for off-the-shelf commercial databases,” Proceedings 19th IEEE Symposium on Reliable Distributed Systems SRDS-2000IEEE, pp.176–185 2000.
[3] F. Pedone, R. Guerraoui, and A. Schiper, “Transaction reordering in replicated databases,” Proceedings of SRDS’97: 16th IEEE Symposium on Reliable Distributed SystemsIEEE, pp.175–182 1997.
[4] N. Schiper, R. Schmidt, and F. Pedone, “Optimistic algorithms for partial database replication,” Proceedings of the 10th international conference on Principles of Distributed Systems, pp.81–93, Nov. 2006.
[5] S.B. Davidson, “Optimism and consistency in partitioned distributed database systems,” ACM Transactions on Database Systems (TODS), vol.9, no.3, pp.456–481, 1984.
[6] D. Sciascia, F. Pedone, and F. Junqueira, “Scalable deferred update replication,” Proceedings of the International Conference on Dependable Systems and Networks, pp.1–12, 06 2012.
[7] P.T. Wojciechowski, T. Kobus, and M. Kokociński, “State-machine and deferred-update replication: Analysis and comparison,” IEEE Transactions on Parallel and Distributed Systems, vol.28, no.3, pp.891–904, 2017.
[8] 白鳥則郎，三石大，吉廣卓哉，富樫敦，岡崎功，村田嘉利，宇田川佳久，工藤司，村澤茂，國島丈生，青木輝勝，寺西裕一，データベース —ビッグデータ時代の基礎—，第 26 巻，未来へつなぐデジタルシリーズ，共立出版，2018.
[9] G.E. Krasner, S.T. Pope, et al., “A description of the model-view-controller user interface paradigm in the smalltalk-80 system,” Journal of object oriented programming, vol.1, no.3, pp.26–49, 1988.