システム間連携

TIS株式会社
テクノロジー&イノベーション本部
テクノロジー&エンジニアリングセンター

商標

- OracleおよびJava、WebLogicは、オラクルおよびその関連会社の登録商標です。
- HULFTは株式会社セゾン情報システムズの登録商標です。
- Linuxは米国およびその他の国におけるLinus Torvaldsの登録商標です。
- WebSphereは世界の多くの国で登録された
  International Business Machines Corporation の商標です。
- JBossは米国およびその他の国における Red Hat, Inc. またはその子会社の
  登録商標です。
- RabbitMQはVMware, Inc.の米国およびその他の国における商標または登録商標です。

その他の社名、製品名などは、一般にそれぞれの会社の商標または登録商標です。
なお、本文中では、TMマーク、Rマークは明記しておりません。

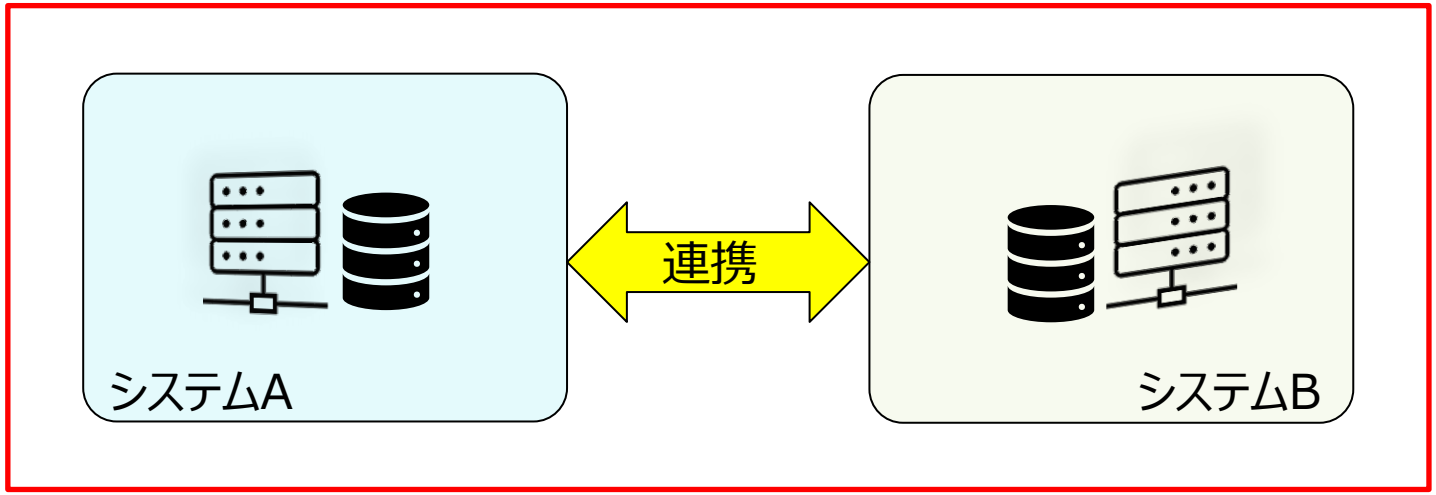
目次

- システム間連携とは
- 方式のパターン
  - データベース共有
  - ファイル連携
  - リモート呼び出し
  - キューによる連携
- システム間連携の理論（CAP定理）

システム間連携とは

システム間連携とは

複数のシステムを連携させて業務処理を遂行すること。



なぜシステム間連携を学ぶ必要があるのか(1/4)

定石を知っておくことでメリットがある

- 新規構築時に、実現案をすぐに思いつける。
- 保守対応時などに、既存システムの方式を理解しやすくなる。

なぜシステム間連携を学ぶ必要があるのか(2/4)

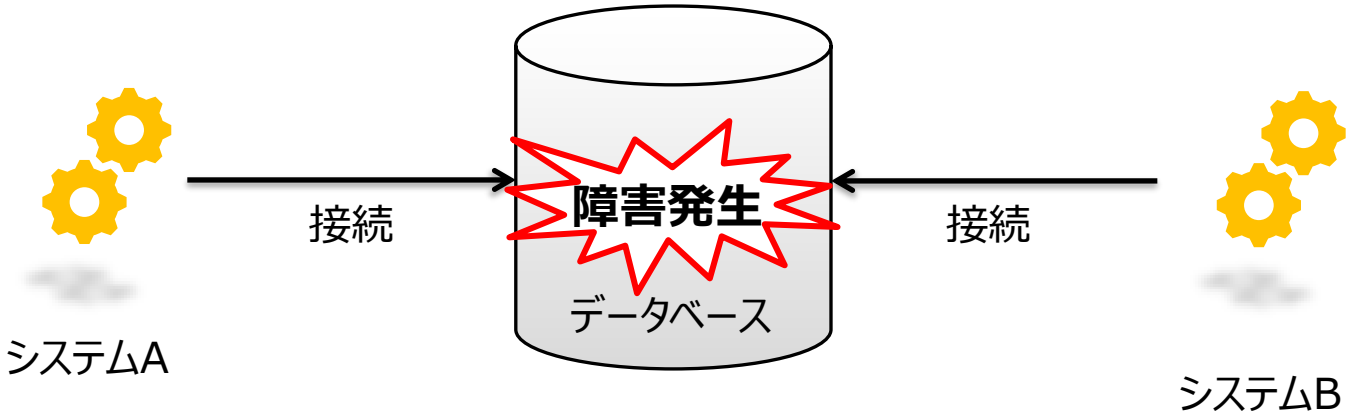
知識がないことによるデメリットを防ぐ

- 誤った方式を選択したことによるトラブルの発生。
- 方式設計の考慮不足によるトラブルの発生。

例を紹介します

なぜシステム間連携を学ぶ必要があるのか(3/4)

誤った方式を選択してしまうとトラブルが起きる。

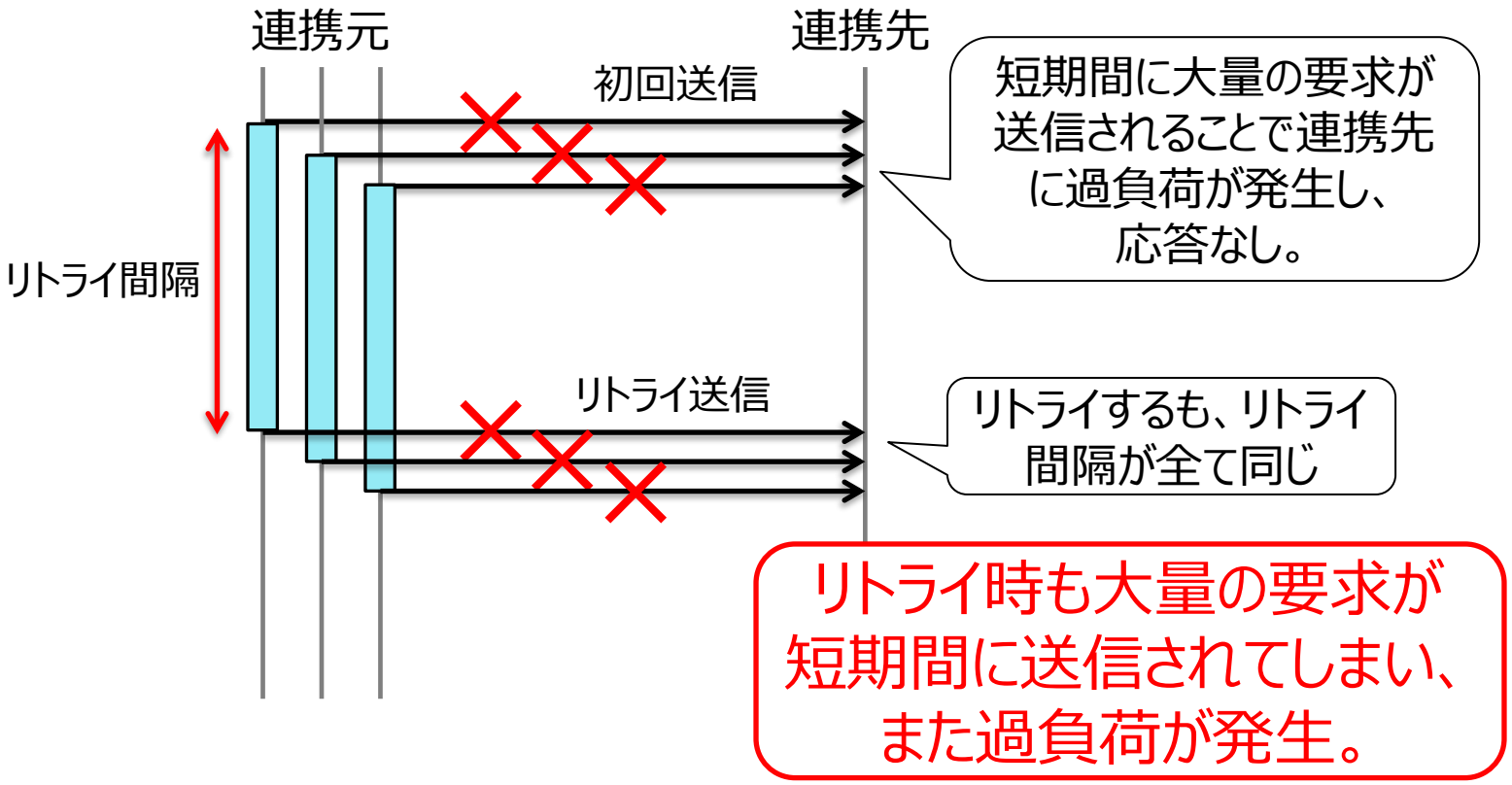


安易にデータベースを共有すると、
データベースの障害により
全てのシステムが停止してしまう。

対策については後半で解説します

なぜシステム間連携を学ぶ必要があるのか(4/4)

方式設計の考慮不足によりトラブルが発生



対策については後半で解説します

方式のパターン

本講座では、システム間連携の方式（仕掛け）として、以下の4パターンを紹介します。

- データベース共有
- ファイル連携

→ データを連携する方法

- リモート呼び出し
- キューによる連携

→ 処理を連携する方法

データベース共有

データベース共有とは

複数のシステムでデータベースを共有することで、
データを連携する方式。

以下の2種類がある。

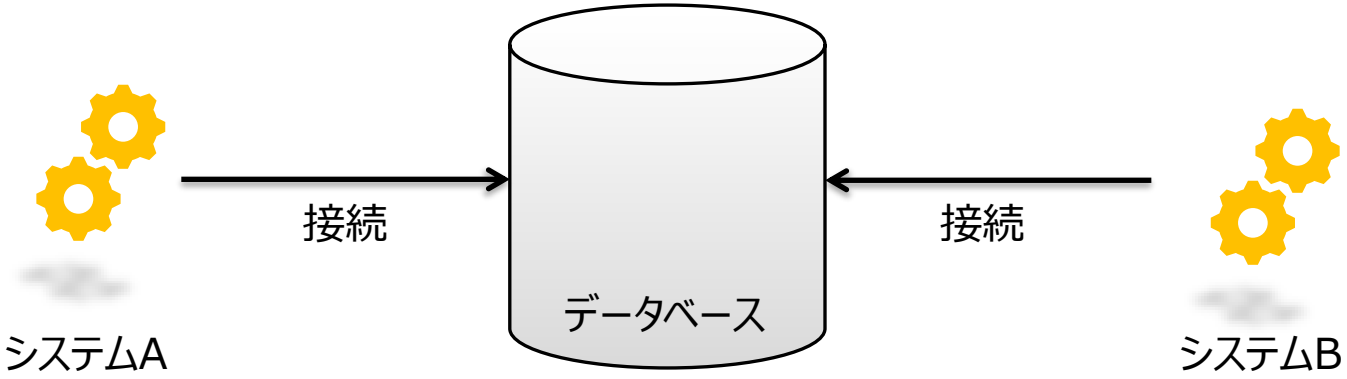
- 単純な共有、データベースリンク
- マテリアライズド・ビューの利用

データベース共有方式の種類

- 単純な共有、データベースリンク
- マテリアライズド・ビューの利用

単純な共有

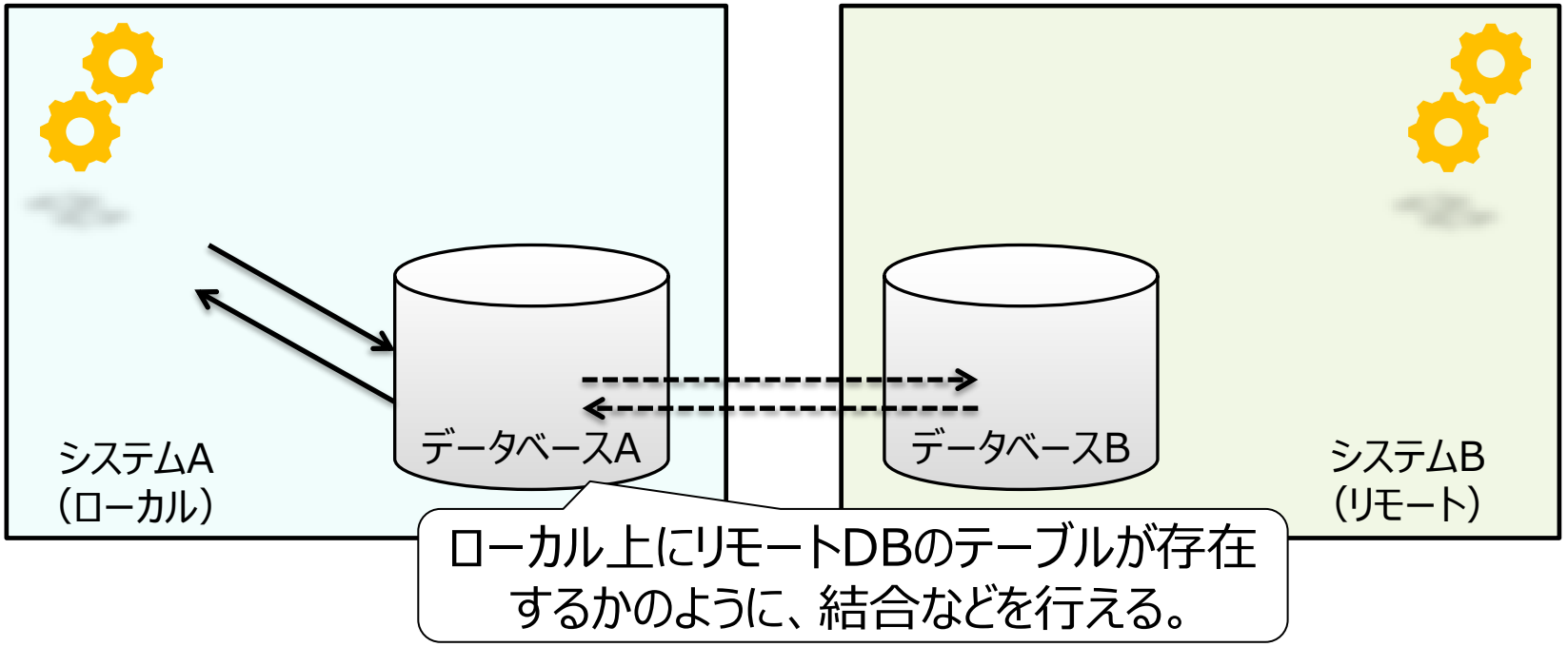
各システムのアプリケーションが共通のデータベースにアクセスする方法
（複数システムで一つのデータベースを共用する）。



データベースリンク(1/2)

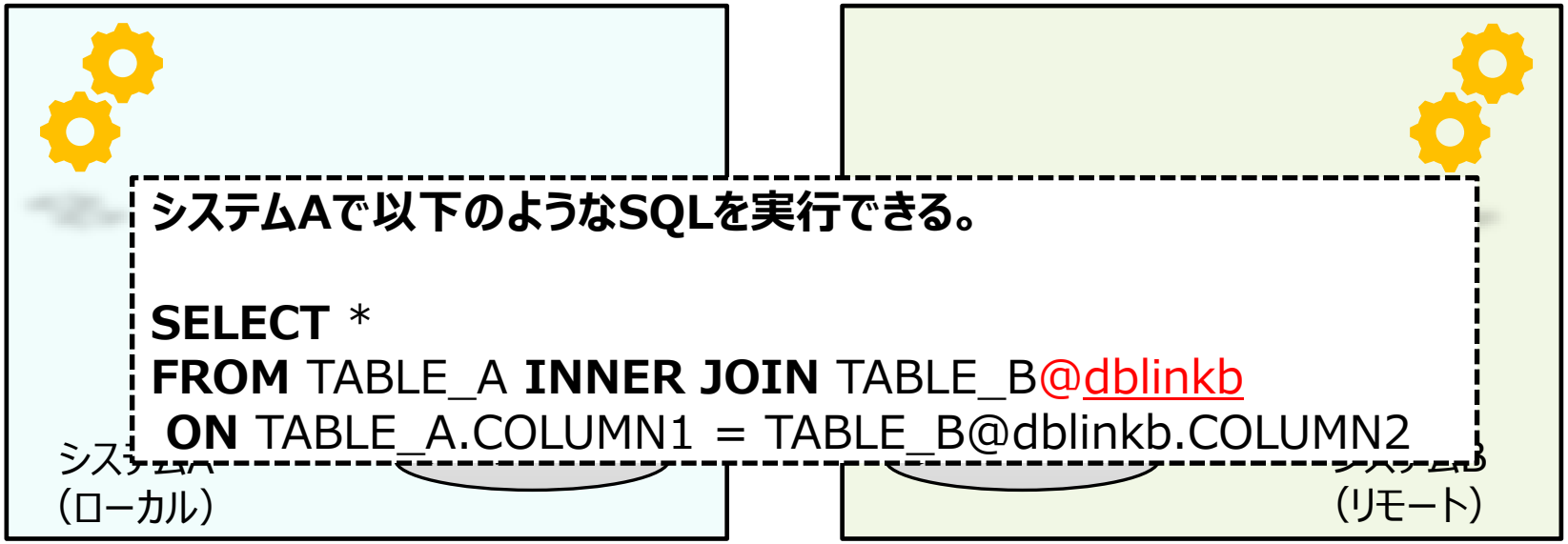
リモートにあるデータベースをローカルにあるかのように扱うことのできる、DBMSの機能。これを使用する方法。

※名称はDBMSによって異なる（データベースリンク⇒Oracle）



データベースリンク(2/2)

下図のように、システムAに存在するTABLE_Aと、システムBに存在するTABLE_Bが、共にシステムAにあるかのようにSQLを発行できる。



単純な共有・データベースリンクの注意点

単純な共有やデータベースリンクのみの使用は推奨されない。

- DB障害時やメンテナンスでDBが一時停止する場合に、共有DBを利用している全てのシステムでサービスを継続できなくなる。⇒可用性が低くなる
- 一方のシステムのデータモデル変更が、もう一方のアプリケーションに影響を与えないかを注意し続ける必要がある。⇒保守性が落ちる

⇒ 他の選択肢を先に検討すること。

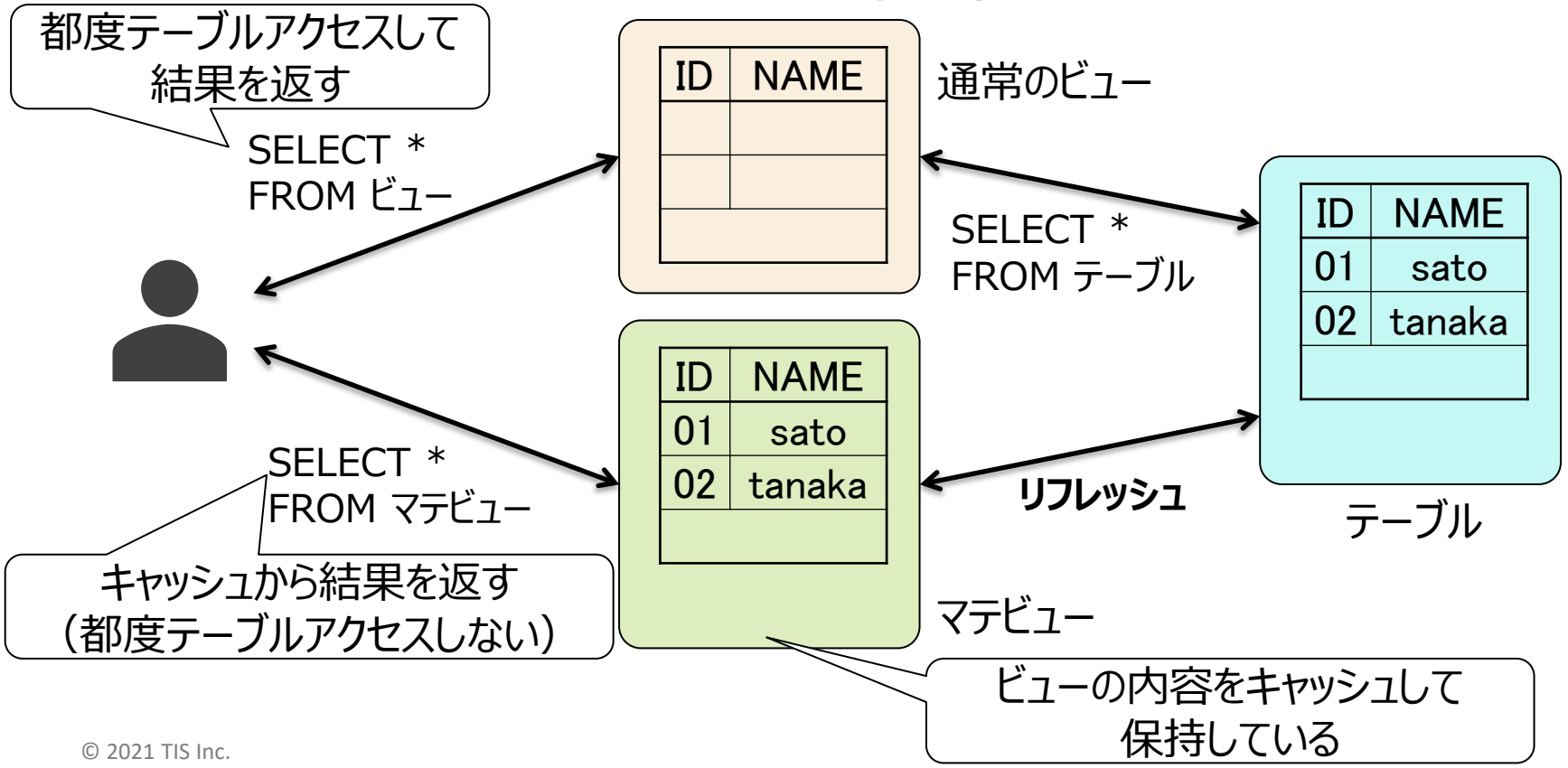
手軽だが、すぐに飛びつくのは避ける。

データベース共有方式の種類

- 単純な共有、データベースリンク
- マテリアライズド・ビューの利用

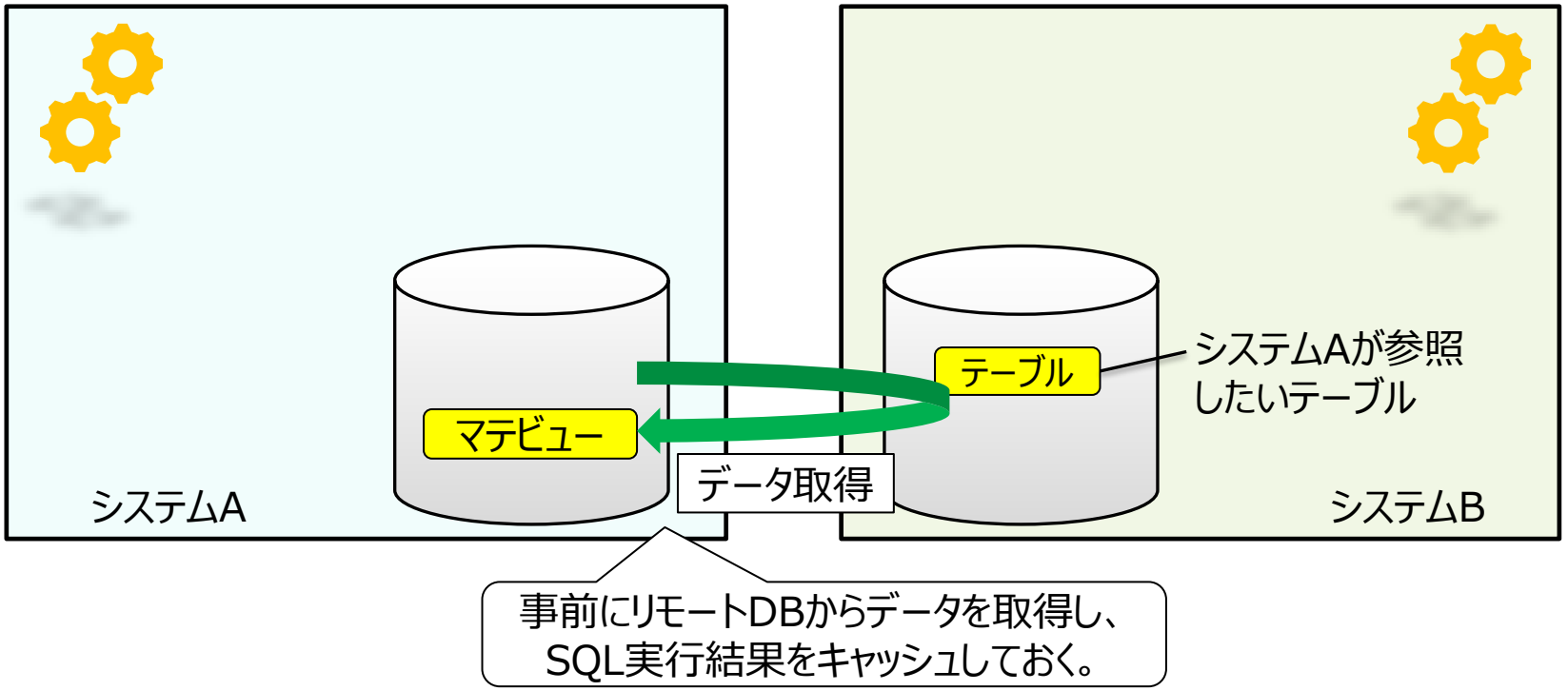
マテリアライズド・ビュー（マテビュー）とは

- SQLの実行結果をキャッシュする。
- 通常のビューと違って検索結果そのものをキャッシュするので、アクセスの度に検索処理が走らない。



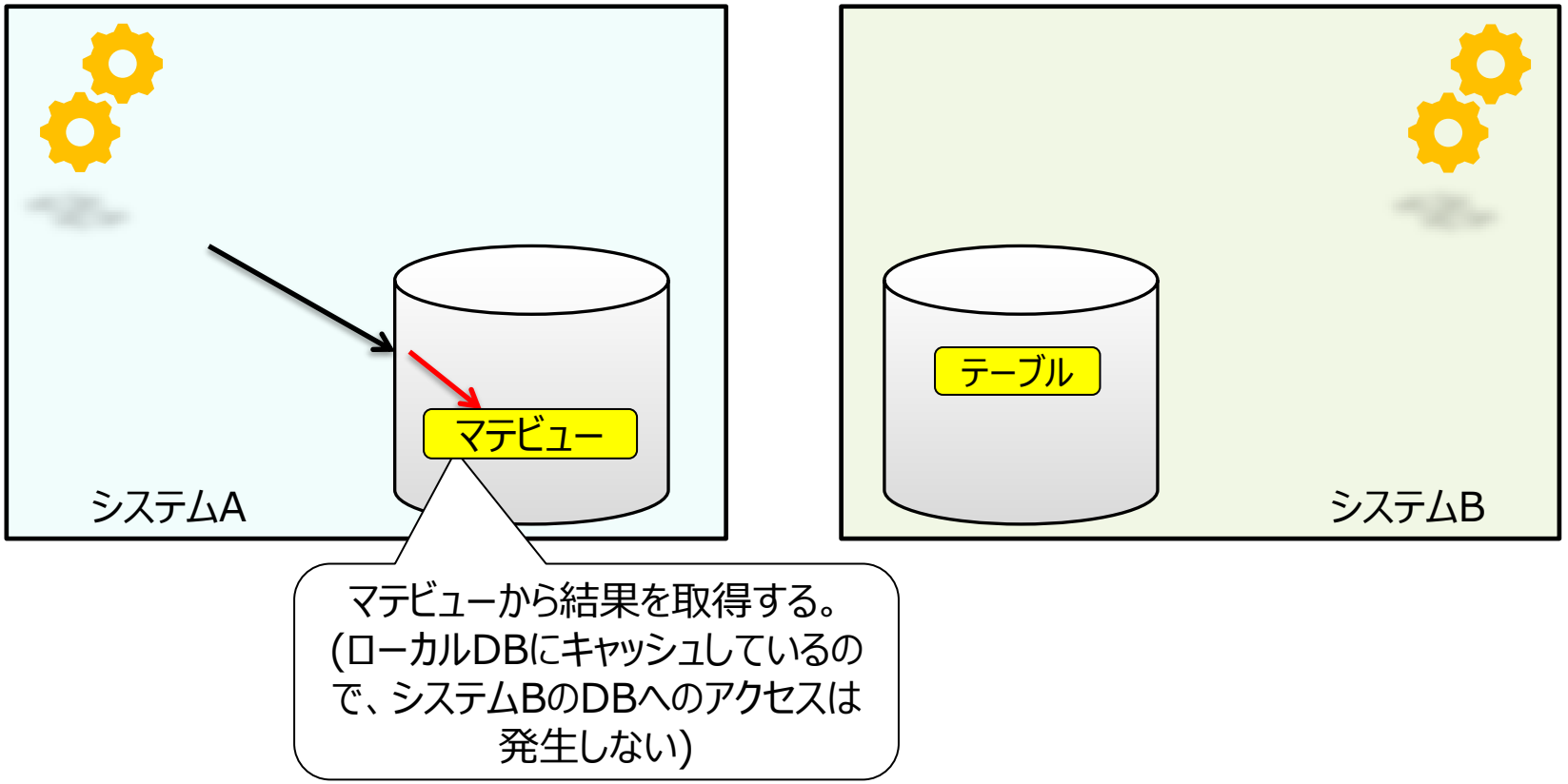
マテリアライズド・ビューの利用 (1/4)

リモートのテーブルをローカルのマテビューとして定義し、データを取得、ローカルDBにキャッシュする。



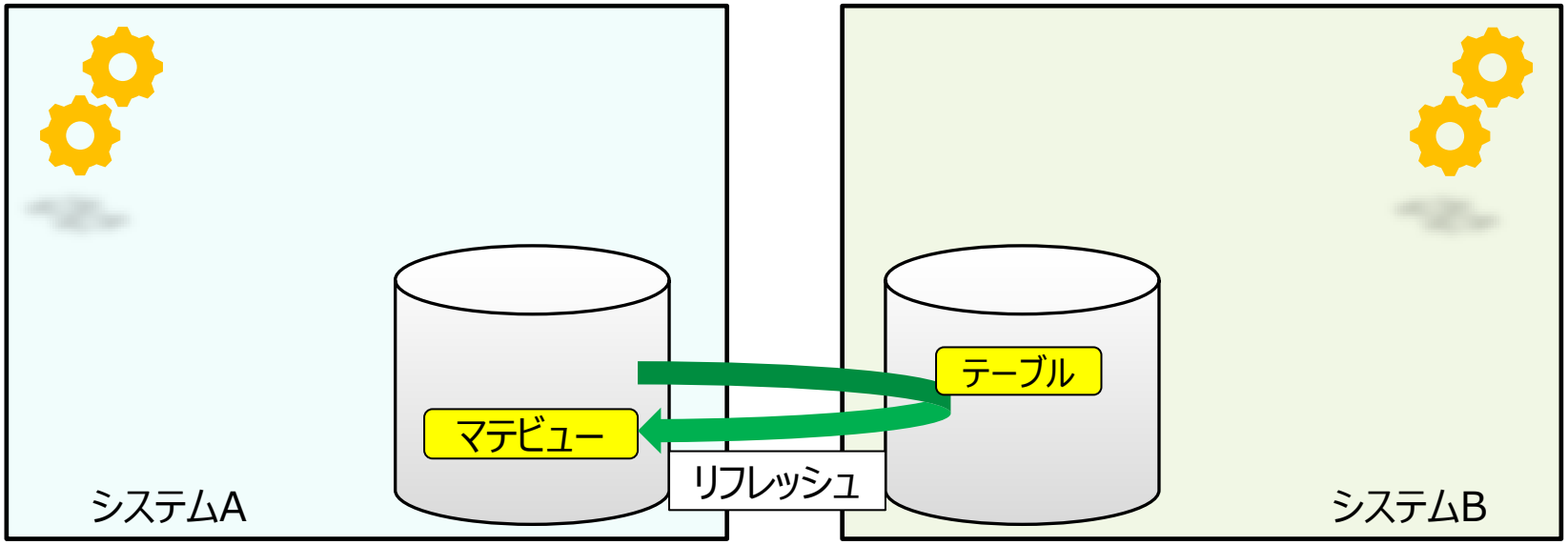
マテリアライズド・ビューの利用 (2/4)

リモートのテーブルデータを取得する際は、リモートにアクセスせず、ローカルのマテビューから取得する。



マテリアライズド・ビューの利用 (3/4)

リモートのテーブルデータに変更があった場合、その更新を取込む必要がある。
そのため定期的にリフレッシュを行う。



マテリアライズド・ビューの利用 (4/4)

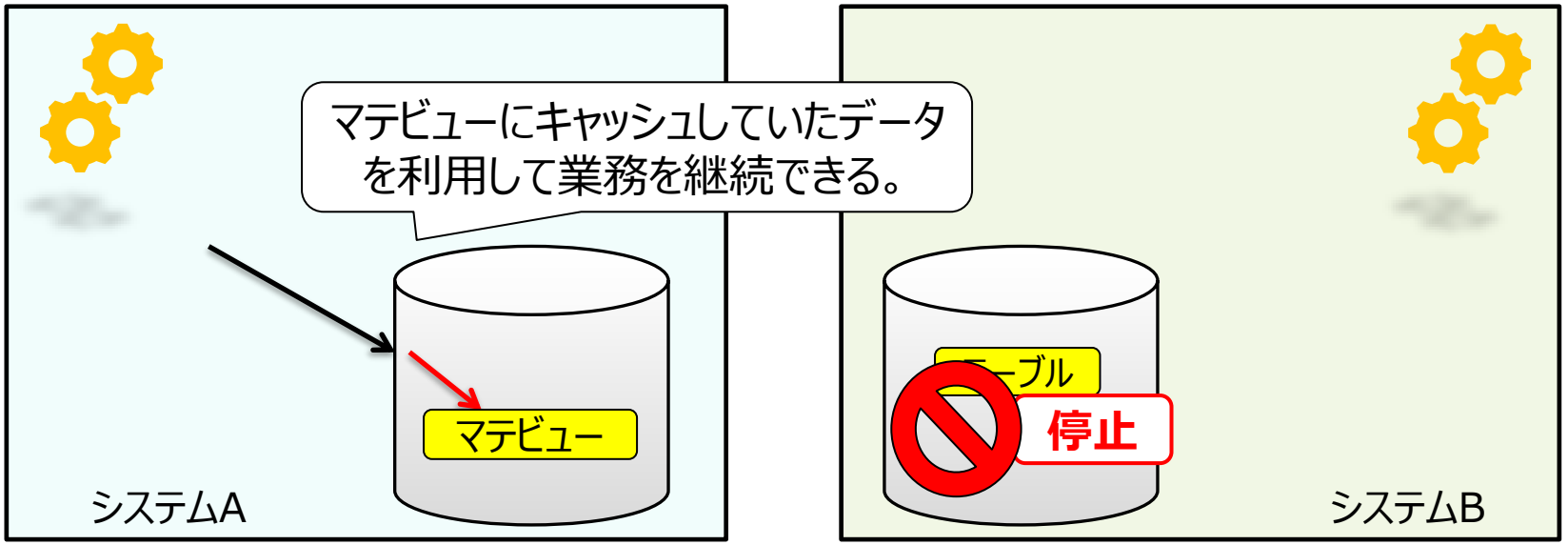
リフレッシュオプション

2通りのリフレッシュ方法が存在。リフレッシュ対象の量や状況に合わせて選択する。

- 完全リフレッシュ
  - 全データを再取得する (洗い替え) 。
- 増分リフレッシュ
  - 更新のあったレコードのみを取り込む。

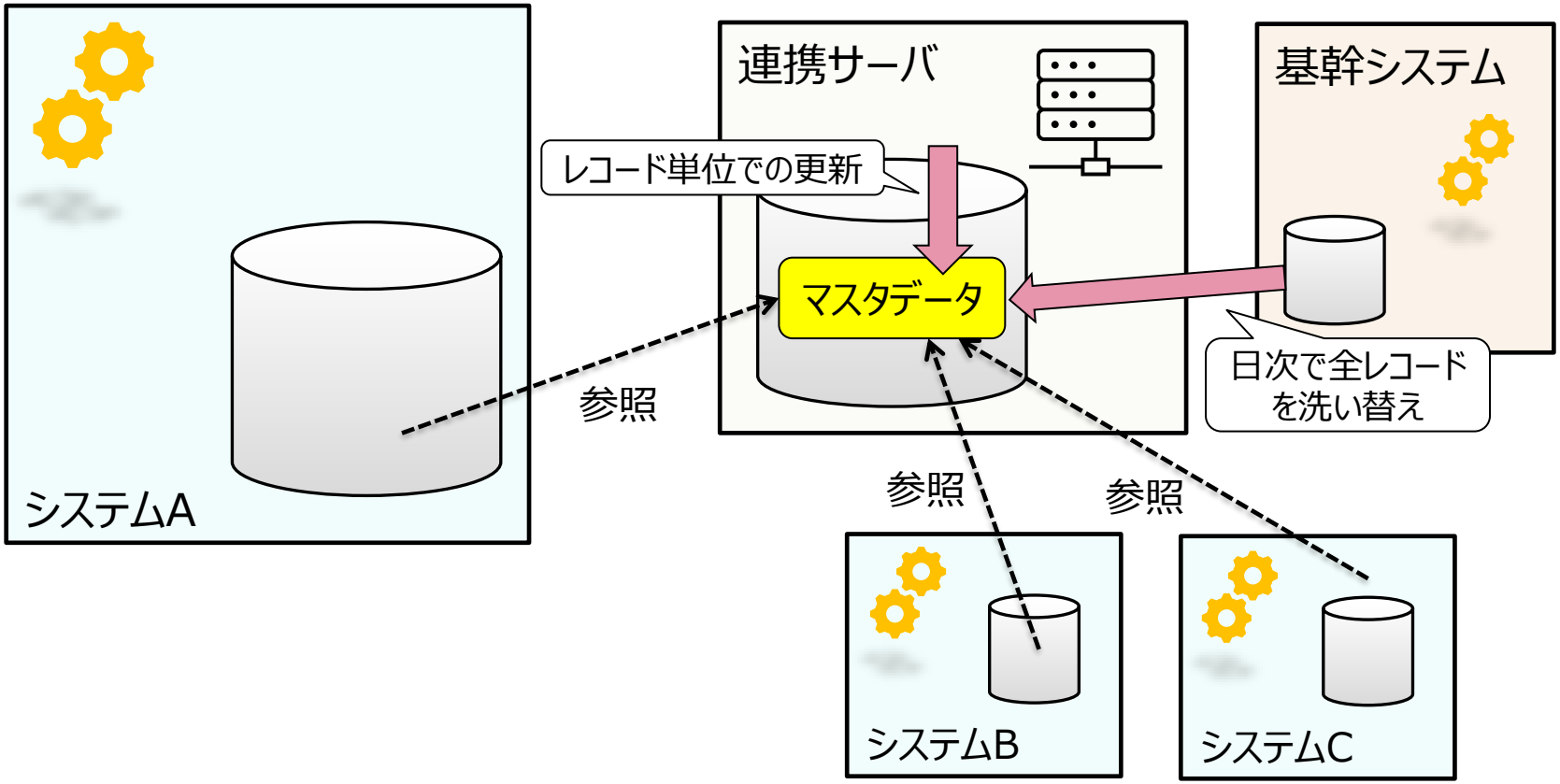
マテリアライズド・ビューの利点

システムBのDBが停止しても、システムAは稼働し続けることができる。⇒可用性の向上



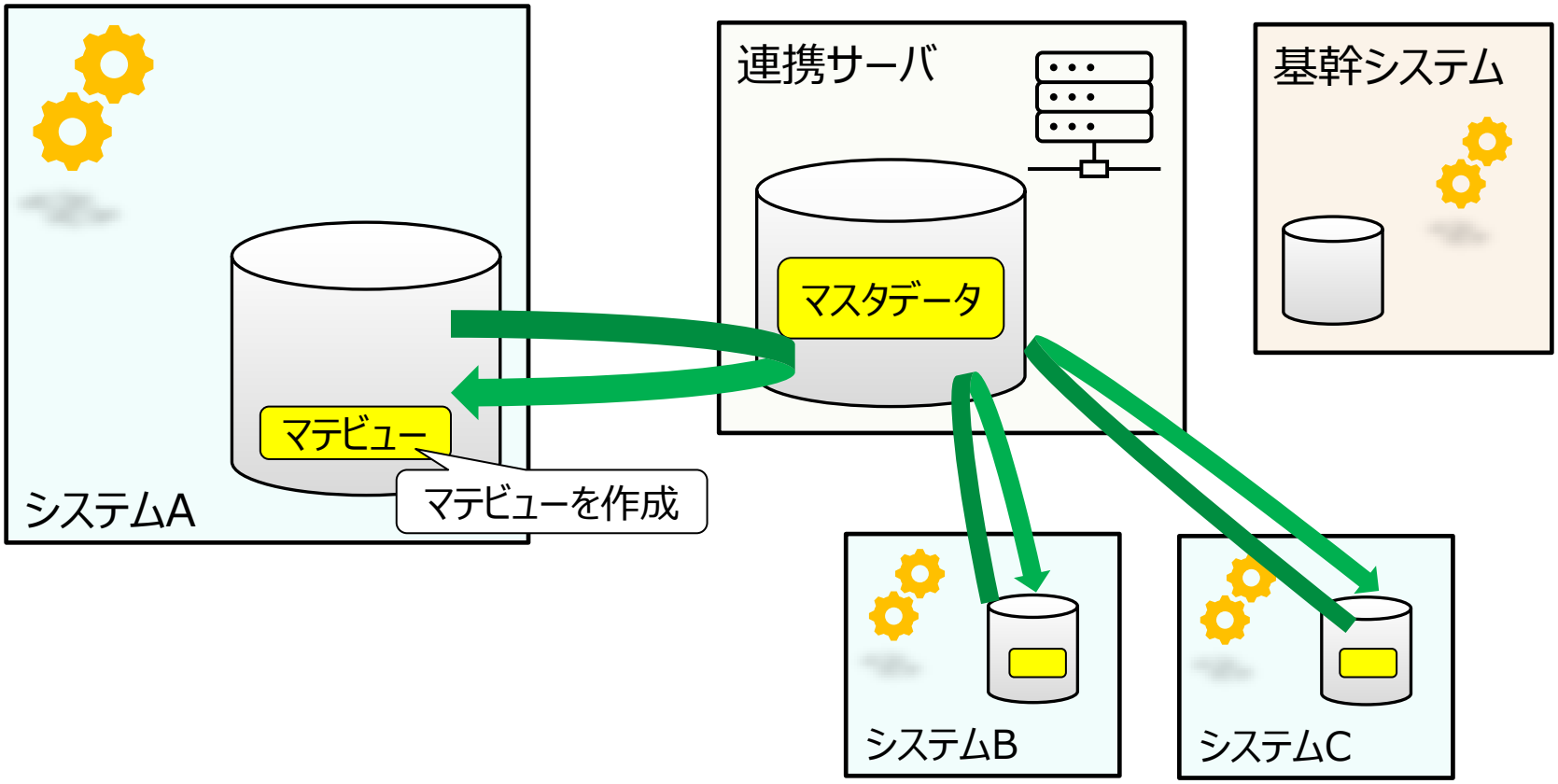
事例(1/4)

複数システムで利用するマスタデータを統合して管理するシステムの事例。
マスタデータは連携サーバ経由でレコード単位の更新が随時発生。
また、日次で基幹システムのデータで全レコード洗い替えを実施。



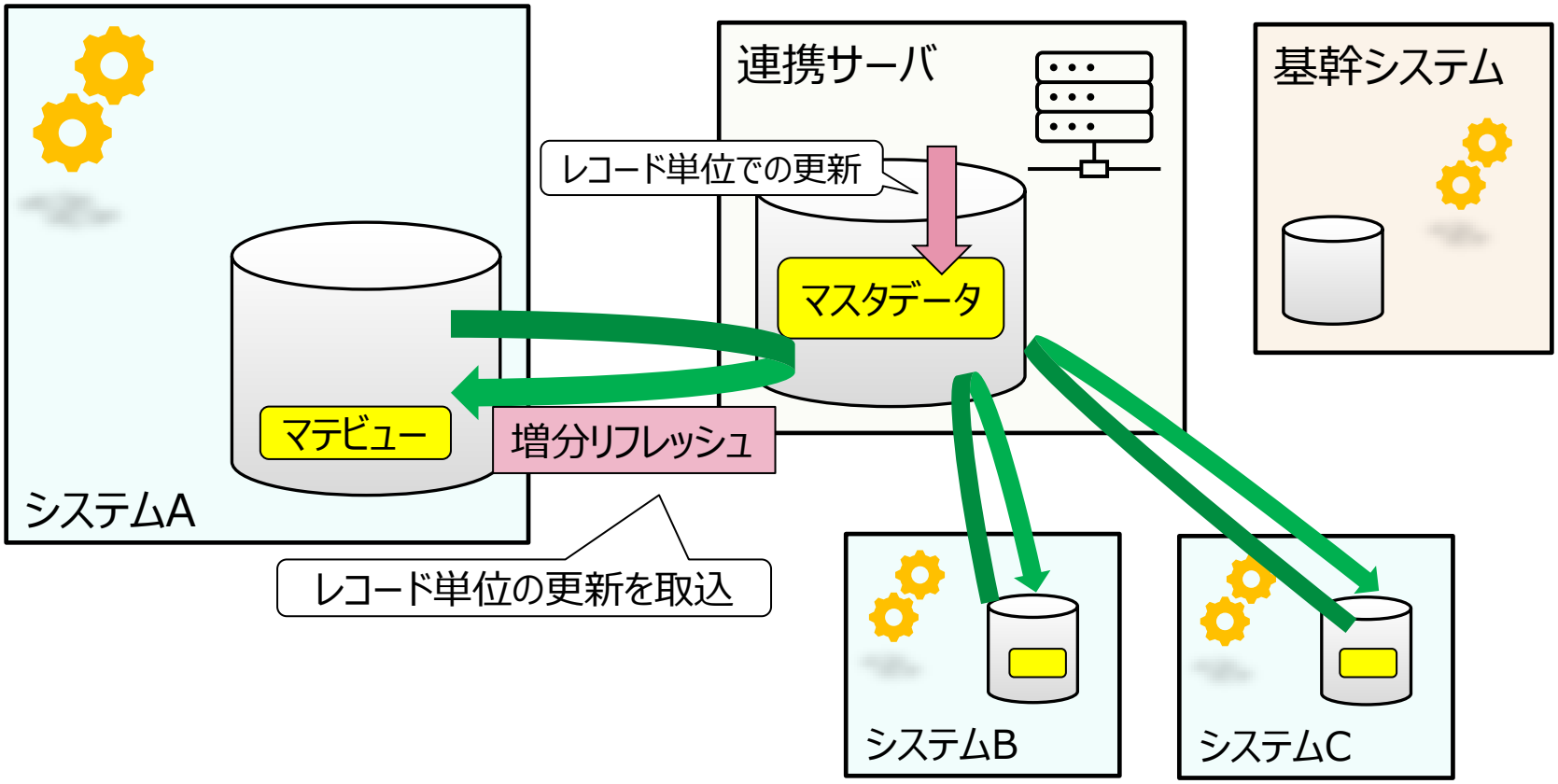
事例(2/4)

マスタデータをマテビューを利用し、ローカルDBにキャッシュする。



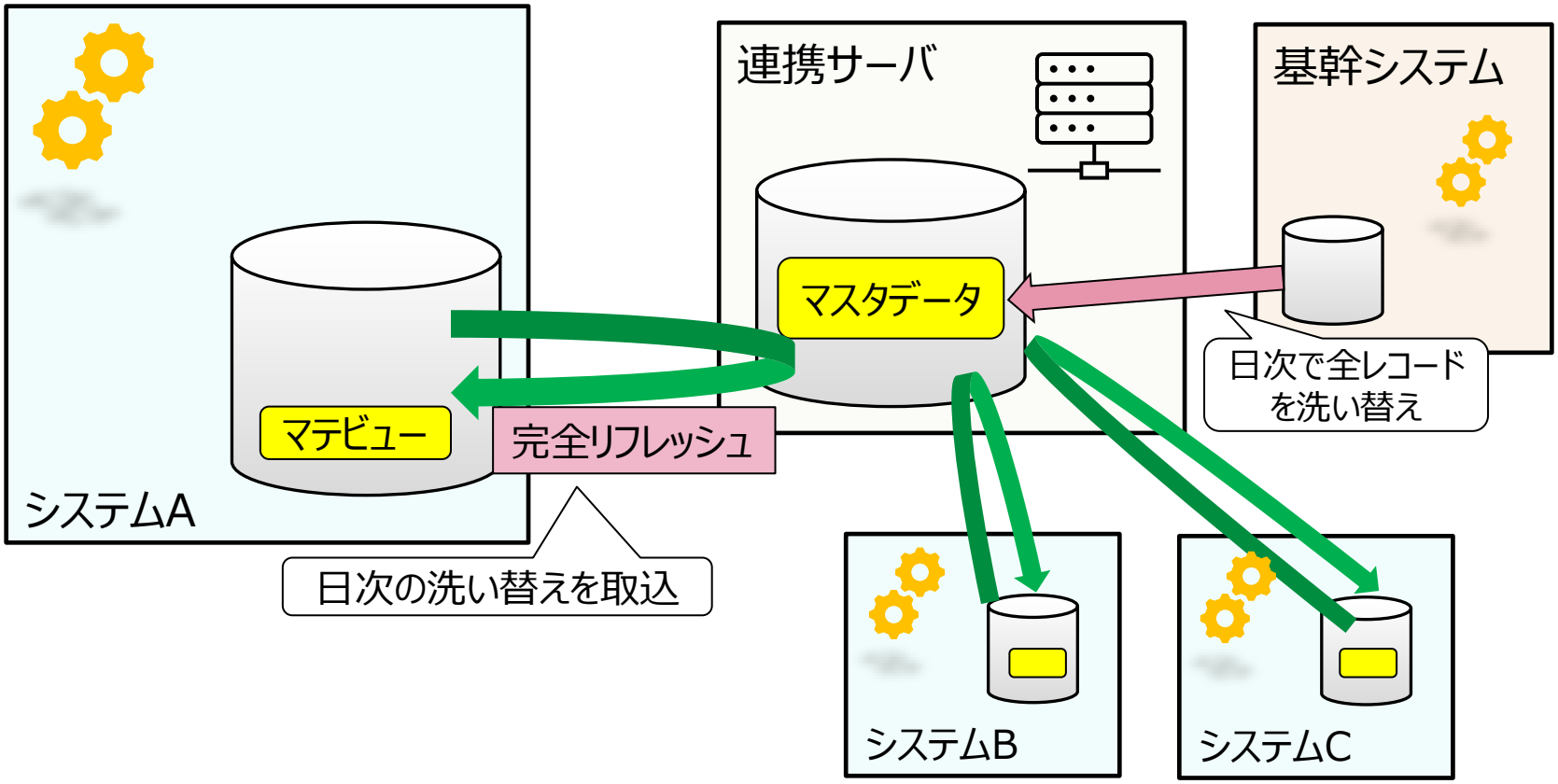
事例(3/4)

連携サーバからの更新を取込むため、30分毎に増分リフレッシュを実行。



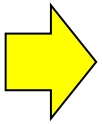
事例(4/4)

基幹システムからの洗い替えを効率的に取り込むため、夜間に完全リフレッシュを実行。



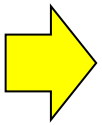
マテリアライズド・ビューを利用する場合の注意点

- 直近のリフレッシュ以降にリモートのデータが更新されていた場合、次のリフレッシュまでそのデータ（最新のデータ）を参照できない。

  業務上許容できるか確認しておくべき

- 単純な共有時と同様、データモデル変更時の他システムへの影響の考慮は必要。

  データモデルの変更が起きにくいことを確認しておくべき

データベース共有の使いどころ

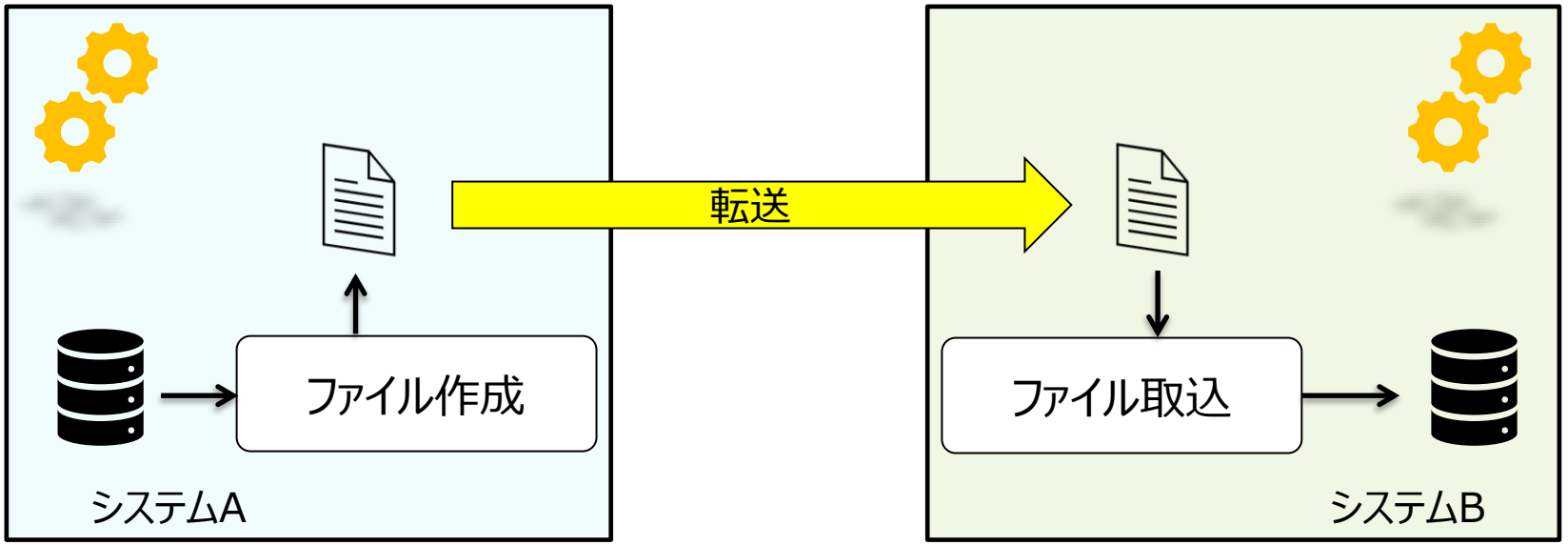
複数システム間で同一かつ大量のデータを共有したい場合。

ファイル連携

ファイル連携とは

連携データをファイルにまとめて連携先へ転送する方法。代表的なものに以下の2種類がある。

- ファイル転送プロトコルの利用(SFTP,SCP)
- HULFT（製品）の利用



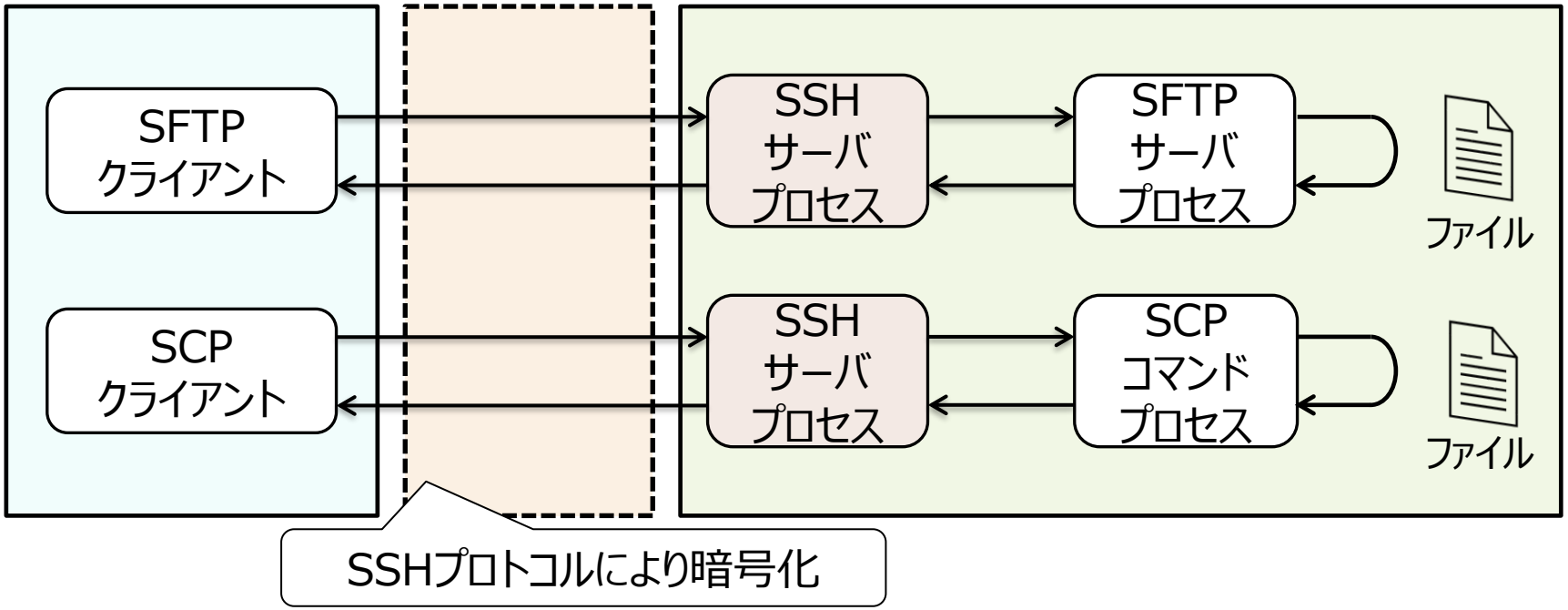
ファイル連携方式の種類

- ファイル転送プロトコルの利用(SFTP,SCP)
- HULFTの利用

ファイル転送プロトコルの利用

以下のファイル転送用通信プロトコルを使用して
ファイルを転送する。

- SFTP（SSH File Transfer Protocol）
- SCP（Secure Copy Protocol）



SFTP,SCPの特徴(1/2)

- SSHプロトコルを使用している。
  - セキュアな認証が可能
  - 通信経路上のデータは暗号化される

❌ FTPではこれらが実現されない

このため、FTPは安易に使用すべきではない。

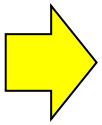
- 一般的なLinuxサーバにはインストール済みなので導入コストが低い。
  - コスト（お金）がかからない
  - インストールの手間がかからない

SFTP,SCPの特徴(2/2)

エンタープライズシステムで利用するには、ファイル転送前後の処理を作り込む必要がある。

ファイル転送前後の処理とは・・・

- 文字コード変換
- 転送エラー時の再送制御
- ファイル転送完了の判定



シェルスクリプトなどで
実装（手作り）する

ファイル連携方式の種類

- ファイル転送プロトコルの利用(SFTP,SCP)
- HULFTの利用

HULFT

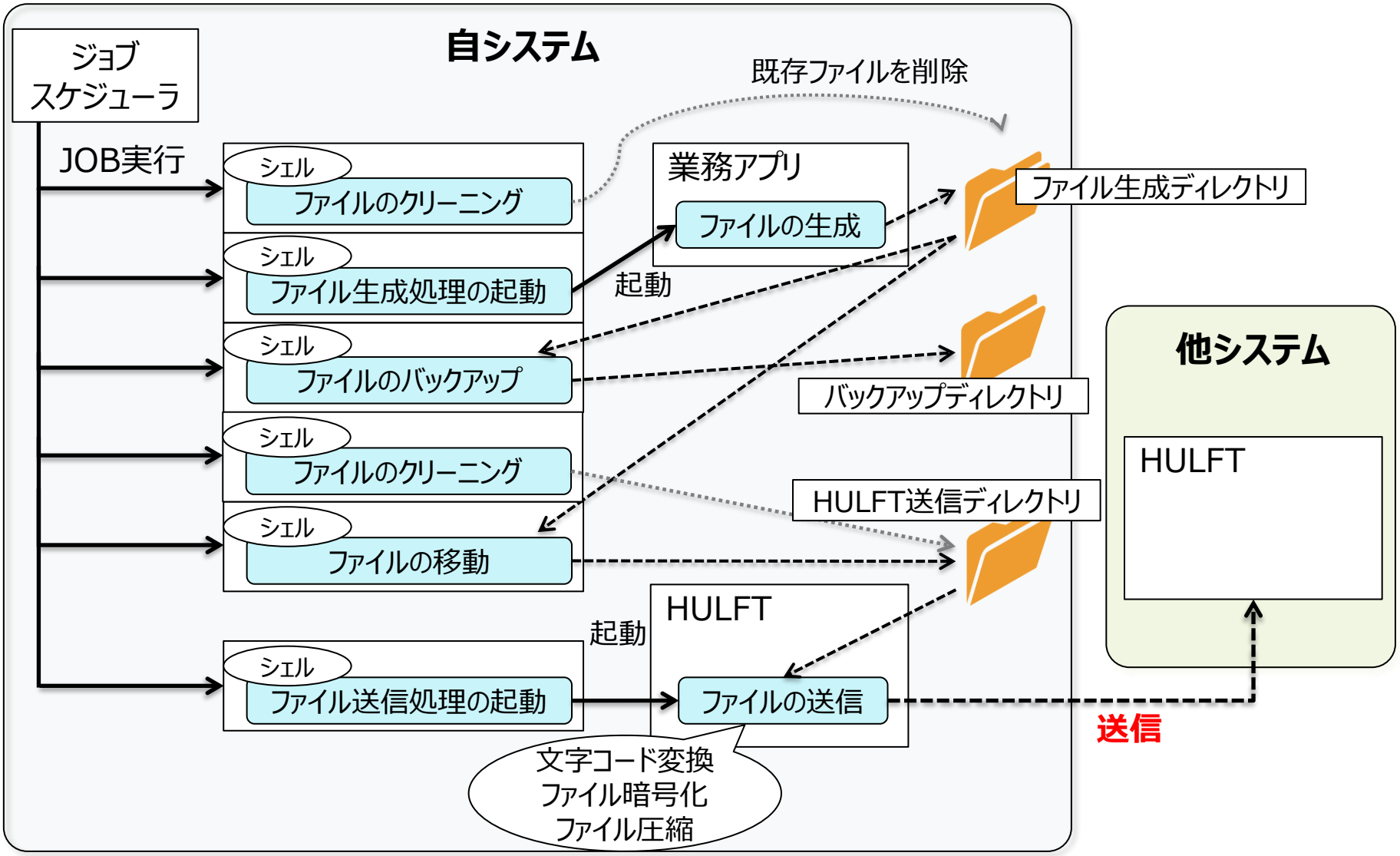
- セゾン情報システムズが開発・販売
- ファイル転送に必要な機能を集約したミドルウェア
- 専用の通信プロトコルを使用

HULFTの特徴

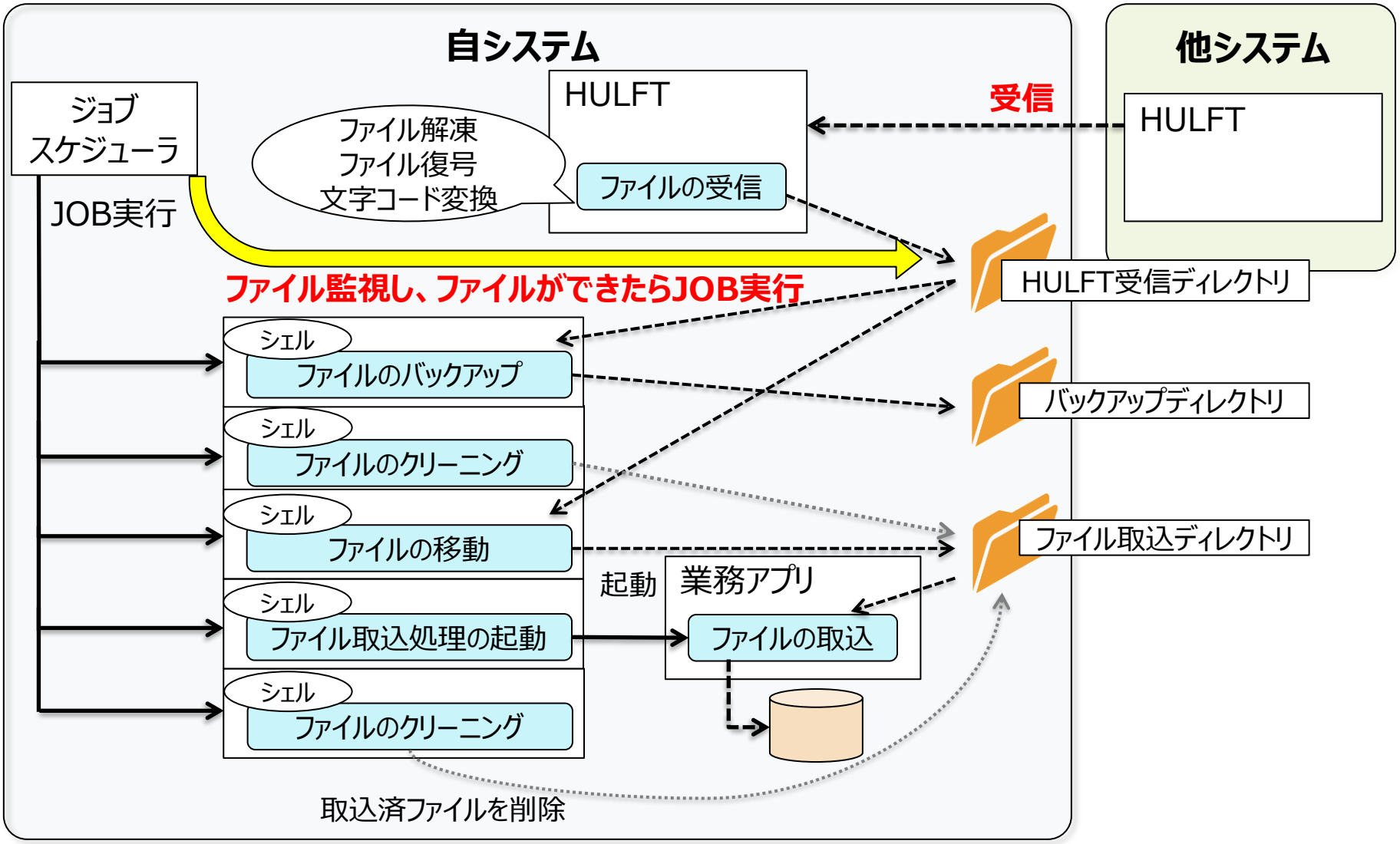
ファイル転送処理に加えて、ファイル転送前後で必要な処理も提供している。

- ファイル転送要求
- ファイル転送処理
- 転送モード設定
- ディレクトリ指定
- ファイル指定
- ファイル整合性の検証
- 圧縮・展開
- 暗号化・復号
- 文字コード変換
- 送受信履歴の出力
- 転送処理のジョブ連携
- データ転送異常時のリカバリ

ファイル連携の事例
～HULFTを使用したファイル配信～



ファイル連携の事例
～HULFTを使用したファイル集信～



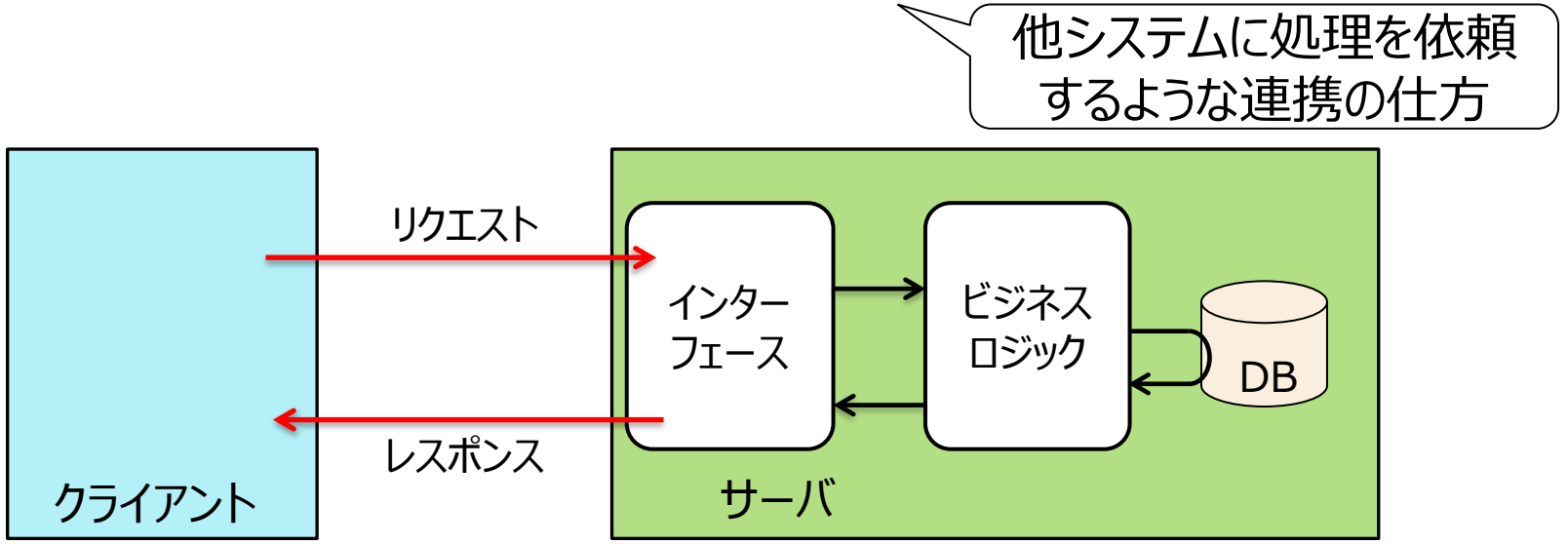
ファイル連携の使いどころ

大量のデータを連携したい場合。

リモート呼び出し

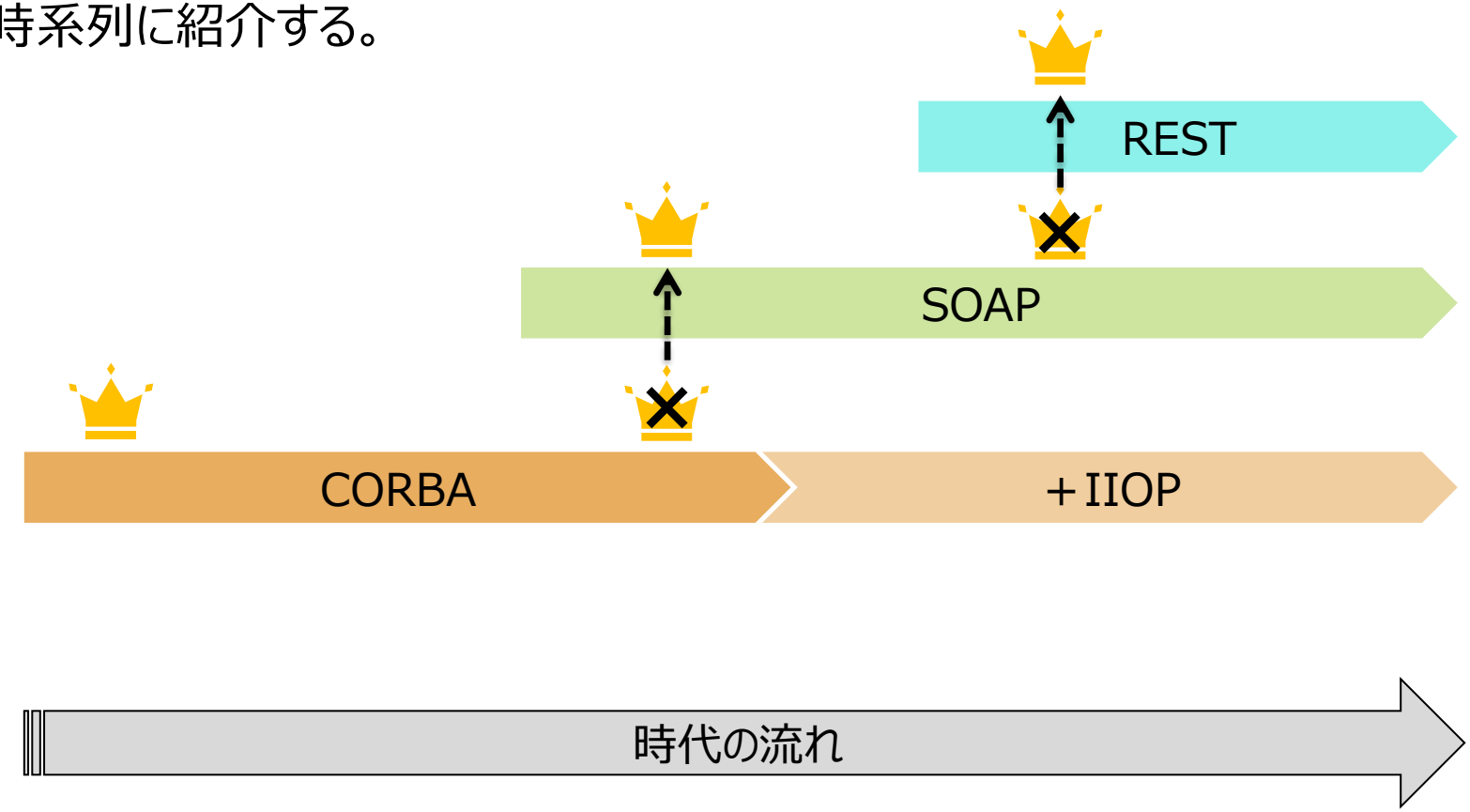
リモート呼び出しとは

サーバは外部から呼び出すことのできるインターフェースを用意する。クライアントはそのインターフェースを利用することでサーバのビジネスロジックを呼び出す。



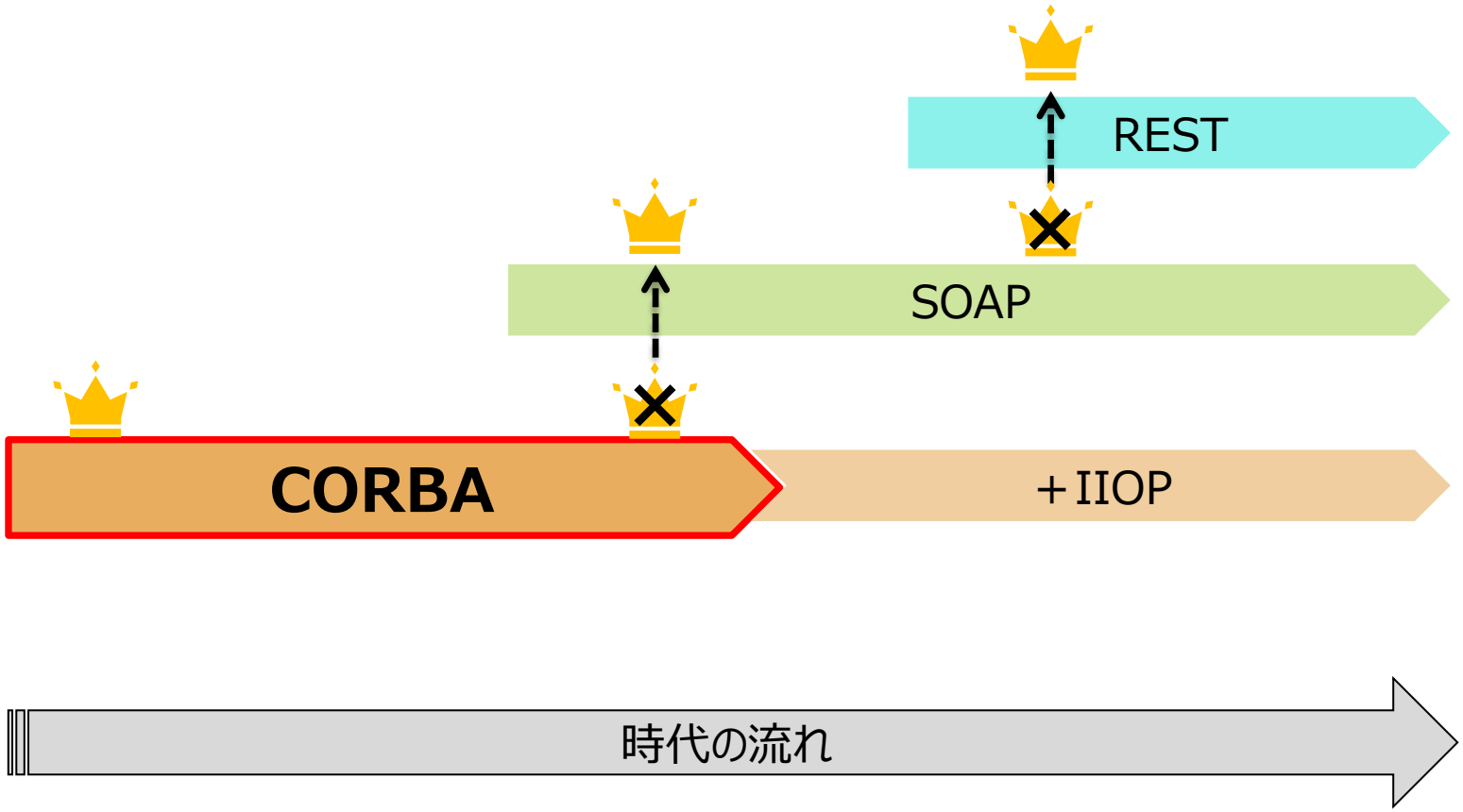
リモート呼び出し方式の歴史

リモート呼び出し方式には色々な技術が存在する。
本講座では、以下のよく使う（使われていた）ものを
時系列に紹介する。



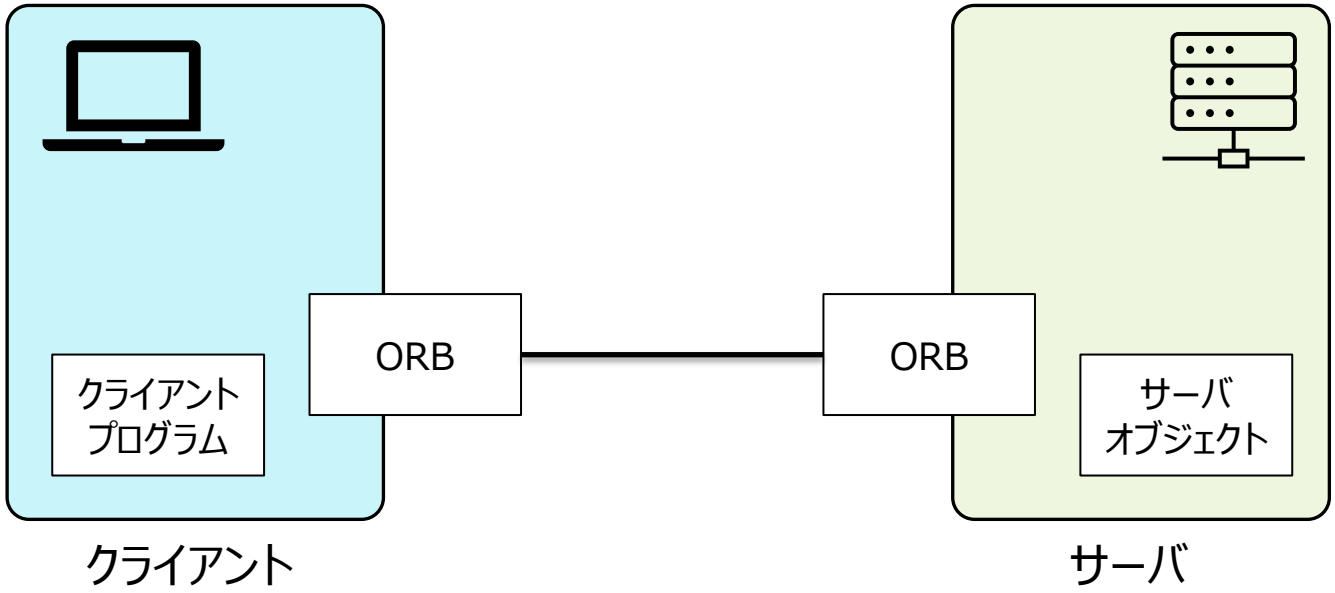
リモート呼び出し方式の歴史

1．CORBA



CORBA（コルバ）

異機種間で機能の相互呼び出しが可能な標準規格。

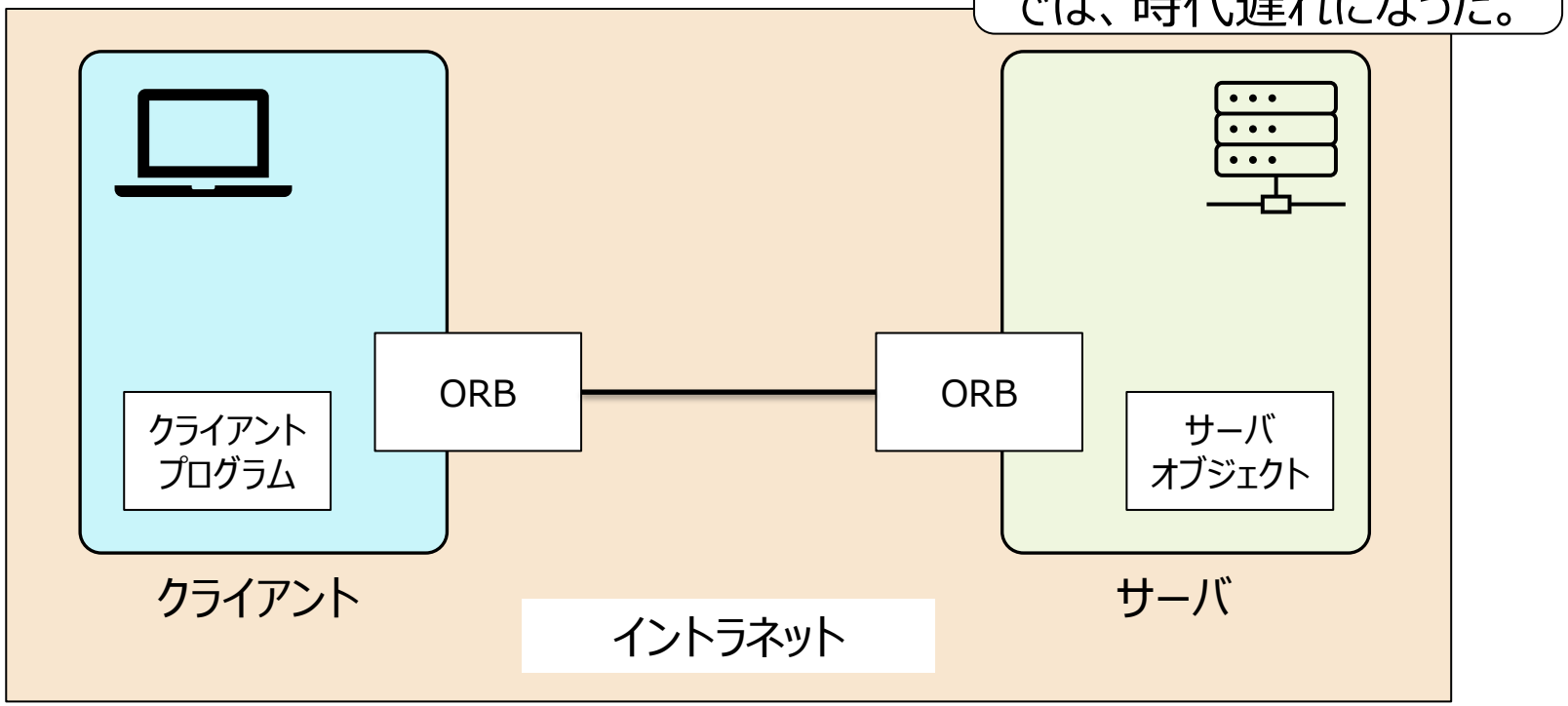


ORB：Object Request Broker　…　別システムの機能を呼び出すためのミドルウェア
CORBA：Common Object Request Broker Architecture
異機種間：各ハードウェアベンダーが出していた、異なるアーキテクチャやOSを採用した機種間

CORBAの問題点

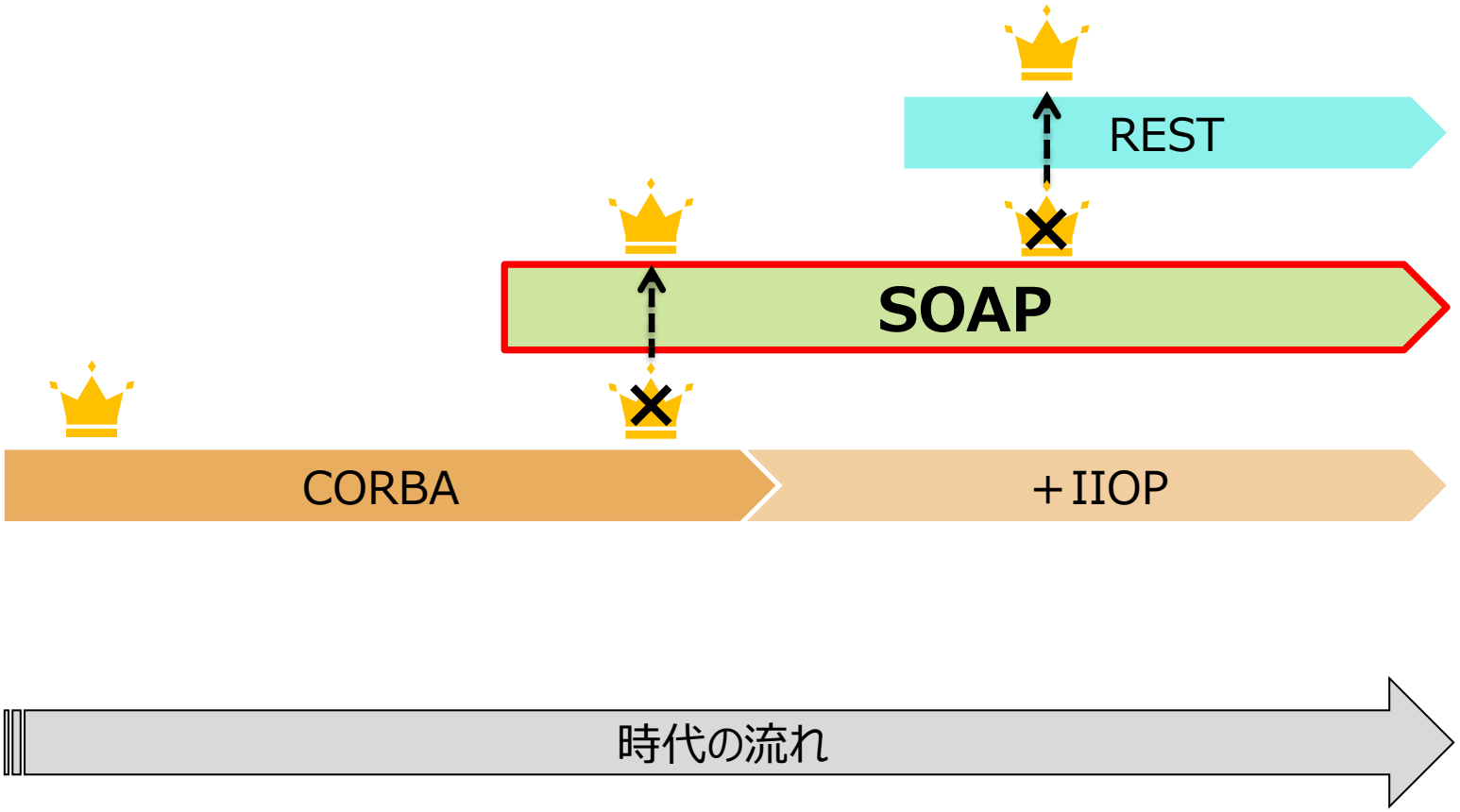
CORBAによるシステム間連携では、イントラネット内のCORBA準拠のシステム同士でしか通信を行えず、インターネットを越えられなかった。

インターネット全盛の時代では、時代遅れになった。



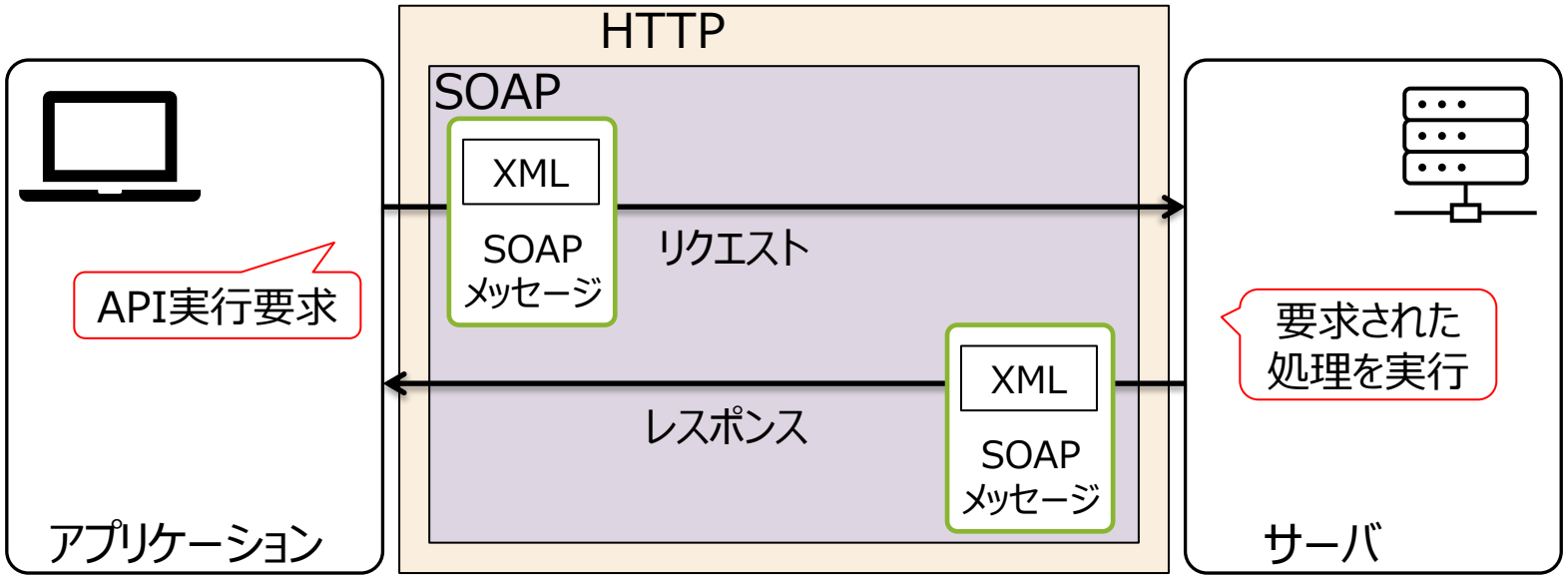
リモート呼び出し方式の歴史

2．SOAP



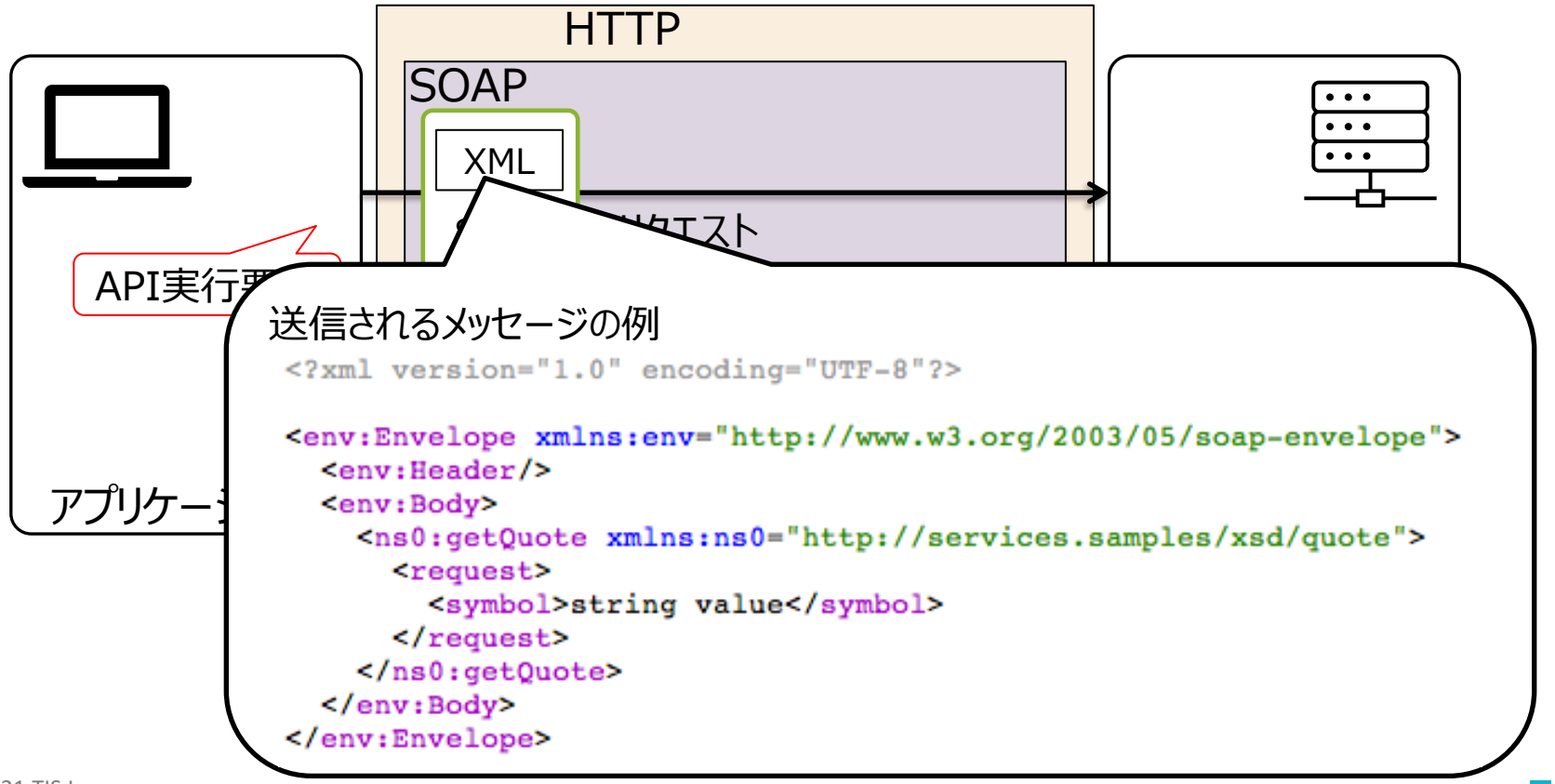
SOAPを利用したHTTPメッセージング(1/3)

SOAPはXML形式のメッセージを送受信するプロトコル。
やり取りするメッセージの形式などを規定しており、メッセージ自体の通信は殆どの場合HTTPを利用して動作する。



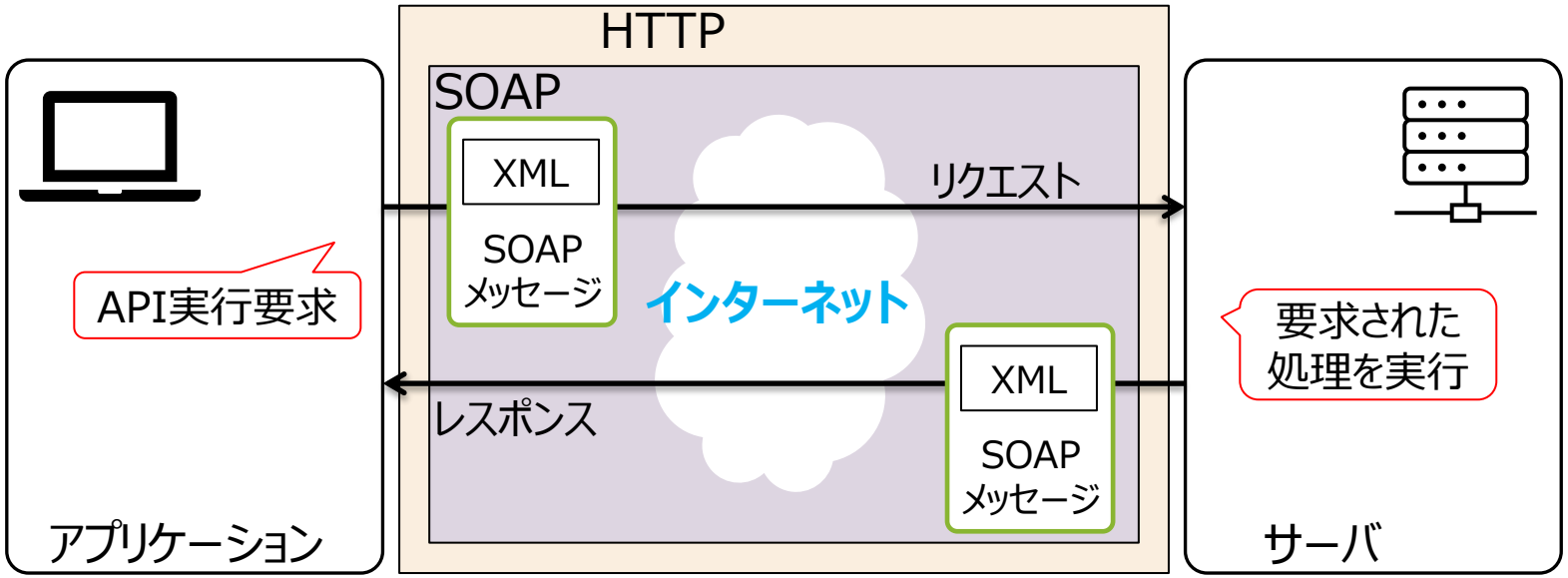
SOAPを利用したHTTPメッセージング(2/3)

SOAPはXML形式のメッセージを送受信するプロトコル。
やり取りするメッセージの形式などを規定しており、メッセージ自体の通信は殆どの場合HTTPを利用して動作する。



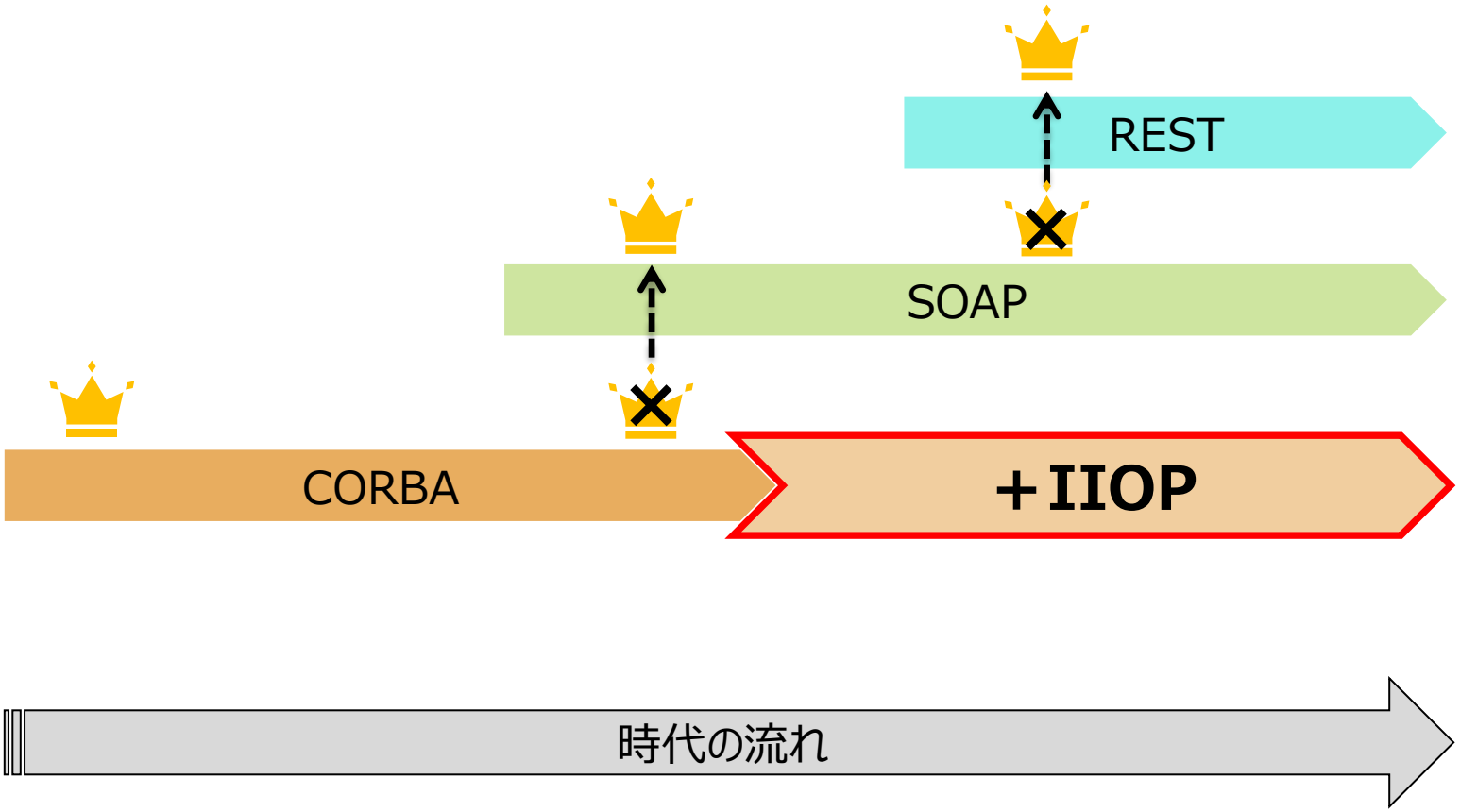
SOAPを利用したHTTPメッセージング(3/3)

HTTPを利用することで、インターネットを超えたメッセージ送受信が可能になった。



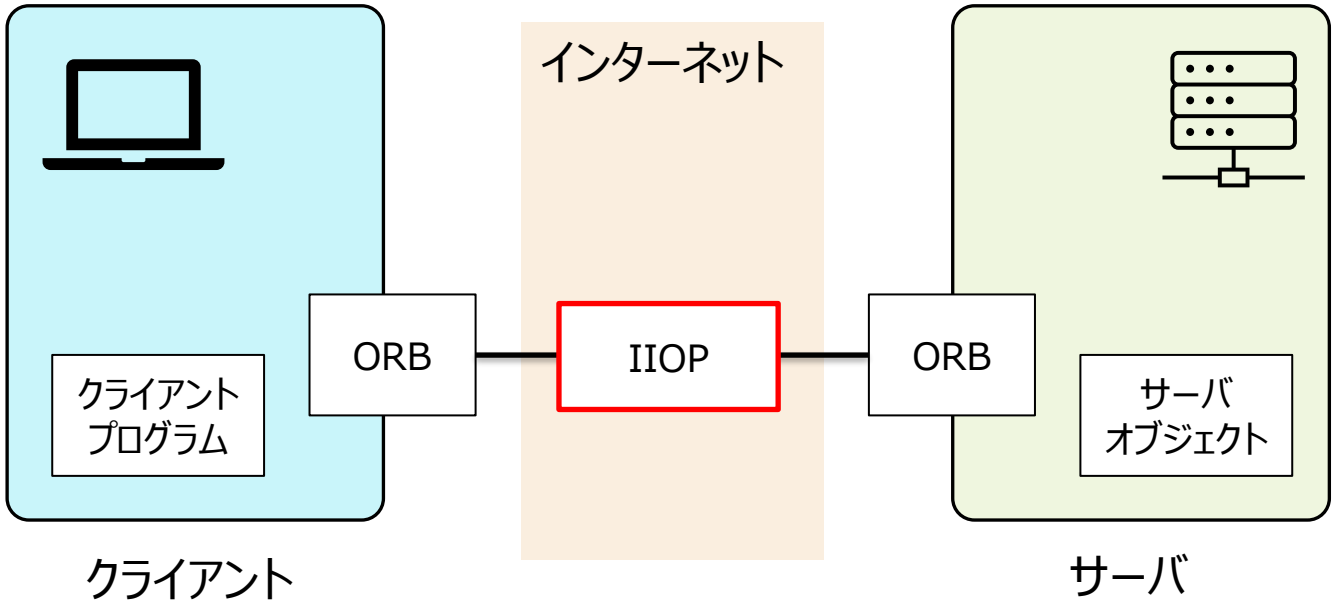
リモート呼び出し方式の歴史

3．CORBA　IIOP



CORBA　IIOP

インターネットなどのTCP/IPネットワークで接続されたコンピュータ間での機能呼び出しをCORBAで実現するプロトコル。



IIOP：Internet InterORB Protocol

CORBA　IIOP

IIOPによりCORBAでインターネット通信ができるようになった頃には、以下の理由により、より扱いやすいSOAPが使われるようになっていた。

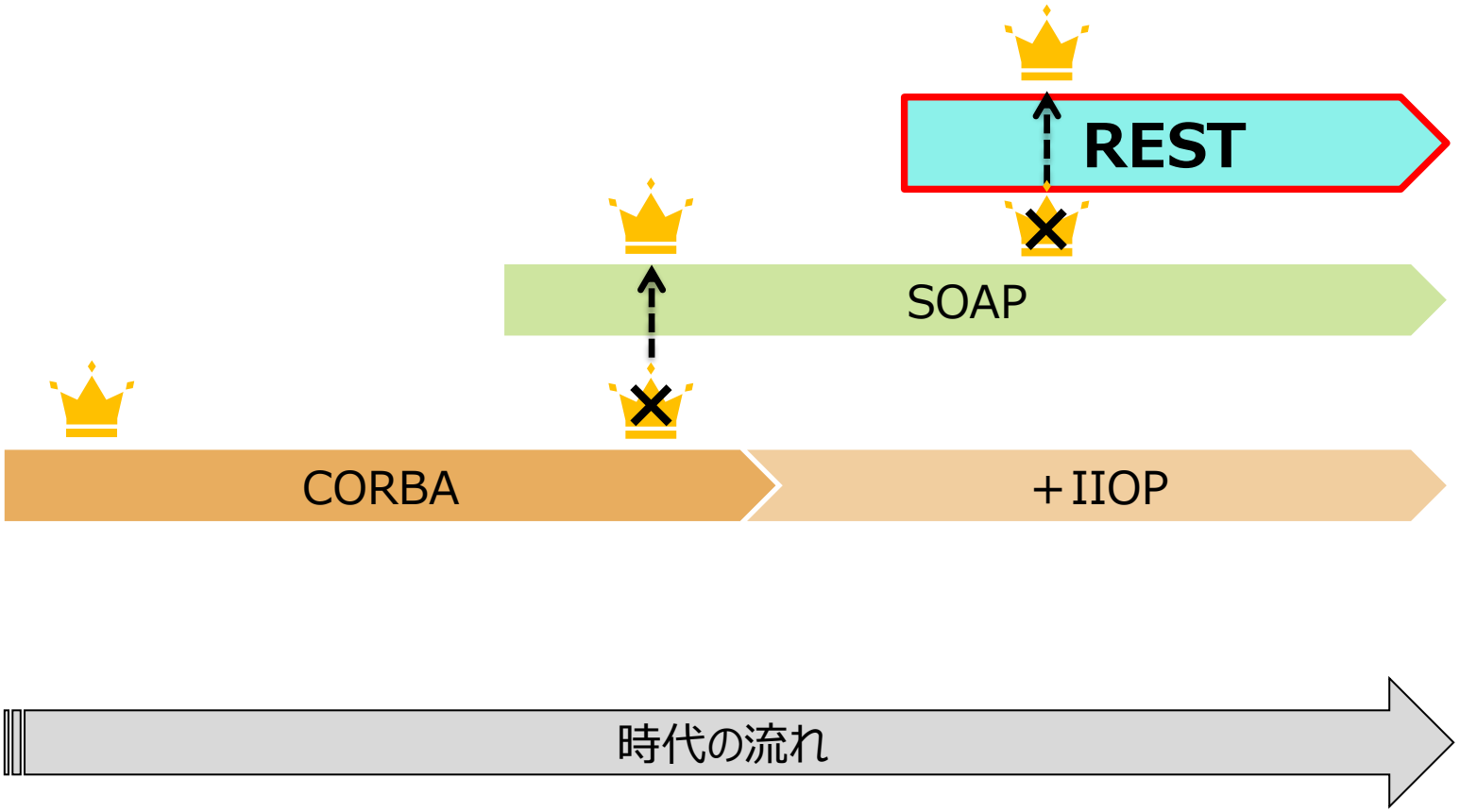
- インターネットを越えたとはいっても、自システム・相手システムともにCORBA準拠の高機能かつ高価な製品を導入する必要があり、ハードルが高い。
- 機能面では劣るが、手軽に導入できるSOAPの方が好まれた。

SOAPの問題点

- メッセージの構造が複雑で、学習コストが高く、動作確認も大変。
- インターフェースや引数の仕様を提供側が自由に決定できるため、統一されたインターフェースになりづらく分かり難い（提供元ごとに学習の必要がある）。

リモート呼び出し方式の歴史

４．REST



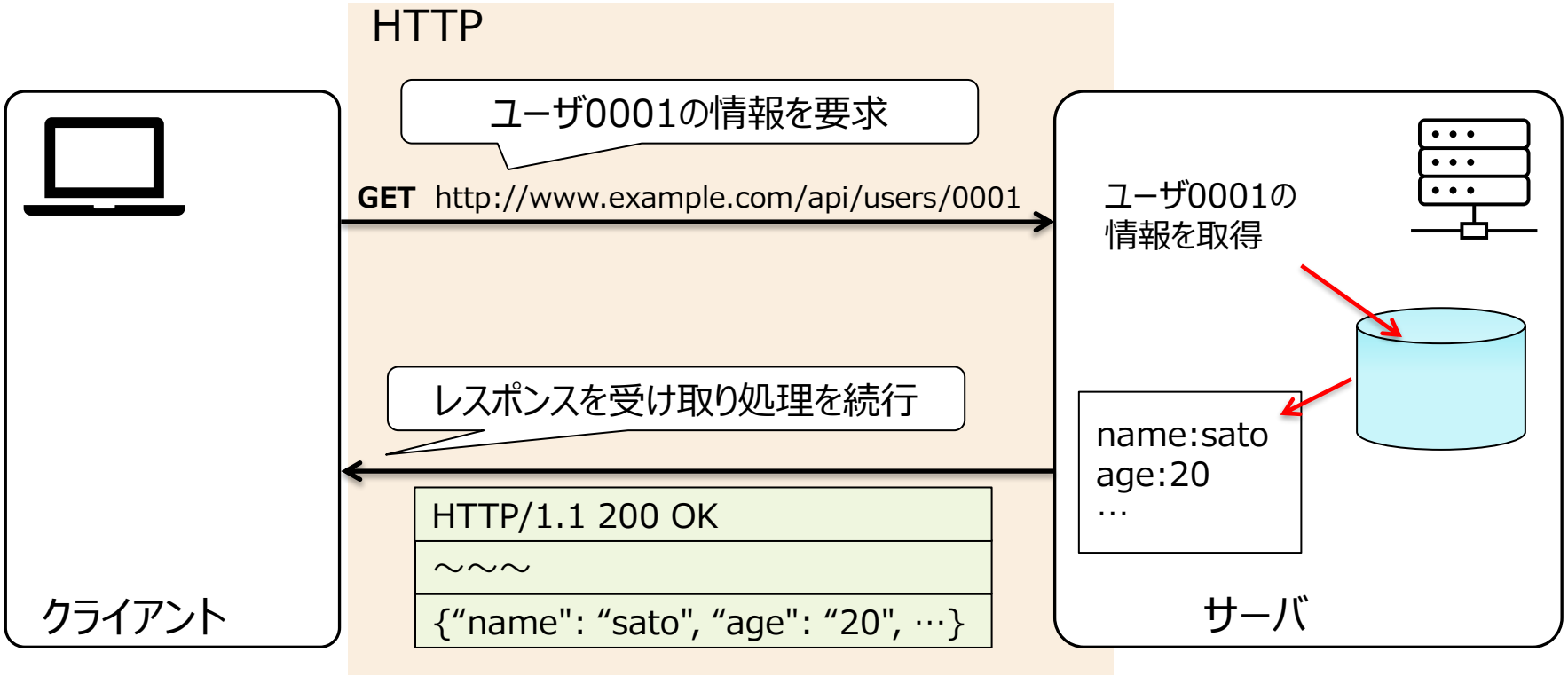
REST（Representational State Transfer）

「汎用的な構文によって一意に識別されるリソース」
を「よく定義された操作」を用いて操作する
アーキテクチャスタイル。

実用上、次ページのように理解しておいて問題ない。

RESTスタイルのHTTPメッセージング

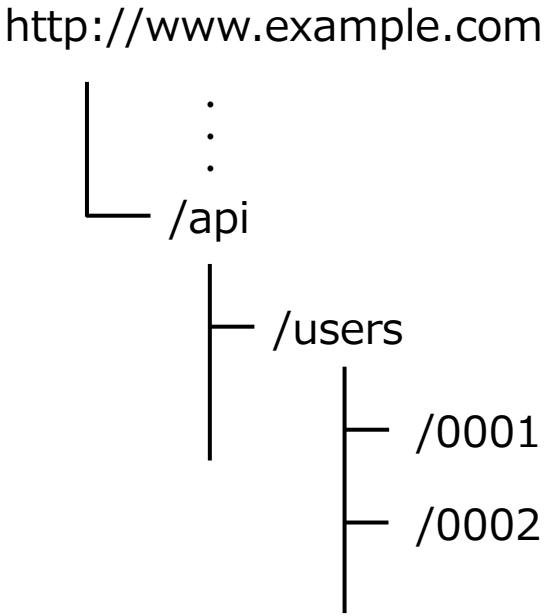
「URIで識別されたリソース」を「HTTPのメソッド」を用いて操作する。



RESTの優れている点(1/2)

どのようなシステムでもAPIが以下の原則に従っているため、技術者にとって理解しやすい。

- 「どんな操作をするか」をHTTPのメソッドに限定する。
- 「何に対して操作するか」をURIで表す。



リソース毎にURIを分ける

URIは名詞のみで構成する

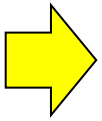
× : ~/users/createUser
× : ~/get0001

“create”や”get”といった動詞が入っている

◎ : ~/0001

RESTの優れている点(2/2)

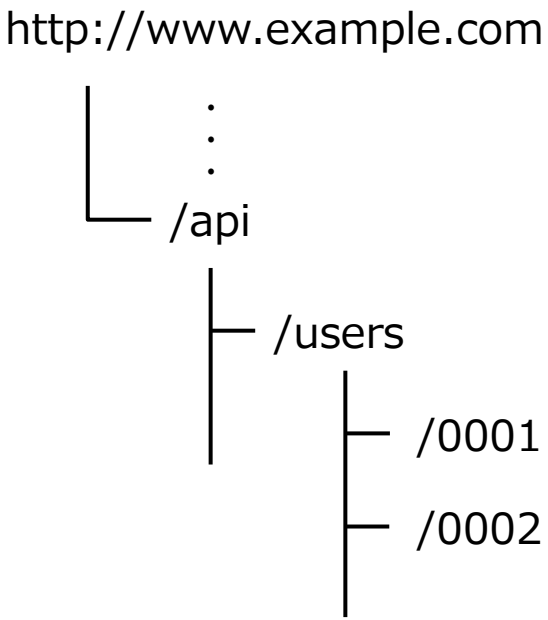
やり取りするメッセージ本文に規定がない（例えば、SOAPは「XML形式」と規定があった）ので、自由な形式を選択でき、よりシンプルな構造にできる。



理解しやすい。動作確認しやすい。

HTTPメソッドの使い分け

「どんな操作をするか」をHTTPメソッドで指定する。
主に使用するのは4つ。



GET http://www.example.com/api/users/0001
➡ ユーザID「0001」のリソースを取得

PUT http://www.example.com/api/users/0001
➡ ユーザID「0001」のリソースを作成or更新
（リソースのURIをクライアントが指定できる場合はPUTを使う）

POST http://www.example.com/api/users
➡ 新規ユーザのリソースを作成
(リソースのURIをクライアントが指定できない場合はPOSTを使う)

DELETE http://www.example.com/api/users/0001
➡ ユーザID「0001」のリソースを削除 (リソースを削除)

RESTに関する参考書籍

まずはこちらから

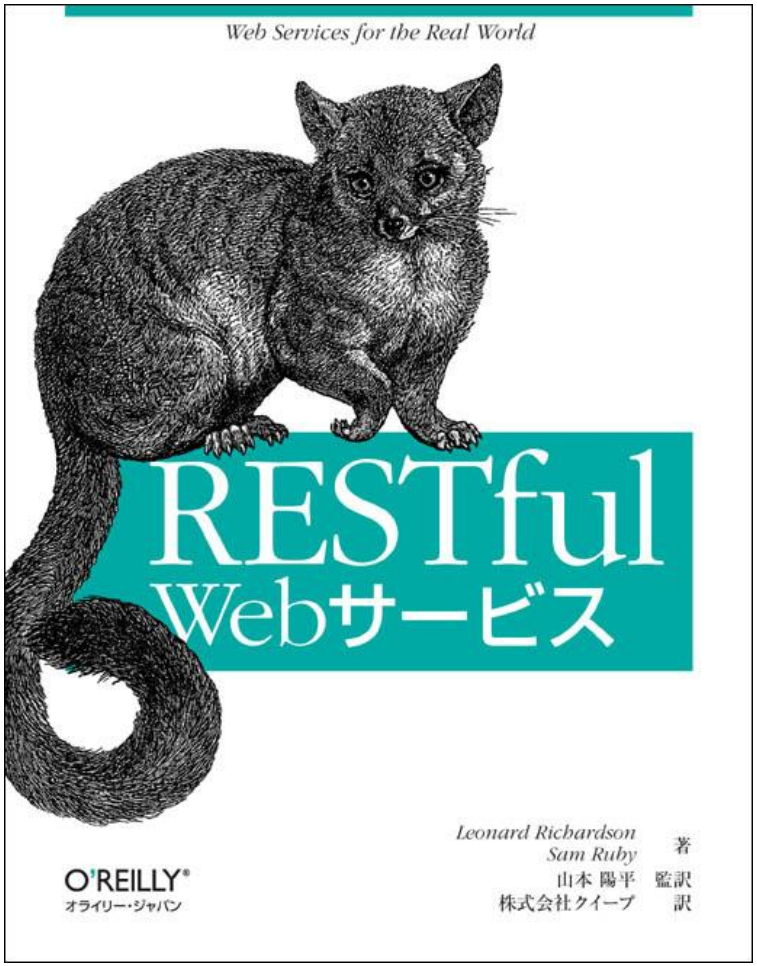
Webを支える技術
　　技術評論社
　　山本陽平　著
　　ISBN 978-4-7741-4204-3

https://gihyo.jp/book/2010/978-4-7741-4204-3
2021年11月25日17時の最新情報を取得

RESTに関する参考書籍

POSTとPUTの使い分けなど、より詳細な内容はこちら

RESTful Webサービス
オライリー・ジャパン
Leonard Richardson, Sam Ruby 著
山本 陽平 監訳
株式会社クイープ 訳
ISBN978-4-87311-353-1



https://www.oreilly.co.jp/books/9784873113531
2021年11月25日17時の最新情報を取得

RESTではないもの

URIが動詞になっている

ユーザ情報を取得
POST http://～～/getUser

参照なのにPOSTを使っている

単にHTTPを使っているだけ

より、REST的

Level 3:Hypermedia Controls
処理結果メッセージに次にアクセス
しそうなURLなども含めるレベル。

実用上目指すべきレベル。

Level 2:HTTP Verbs
Level 1に加えメソッドもきちんと使用。

Level 1:Resources
処理対象の指定にURIを使っているレベル

Level 0:The Swamp of POX

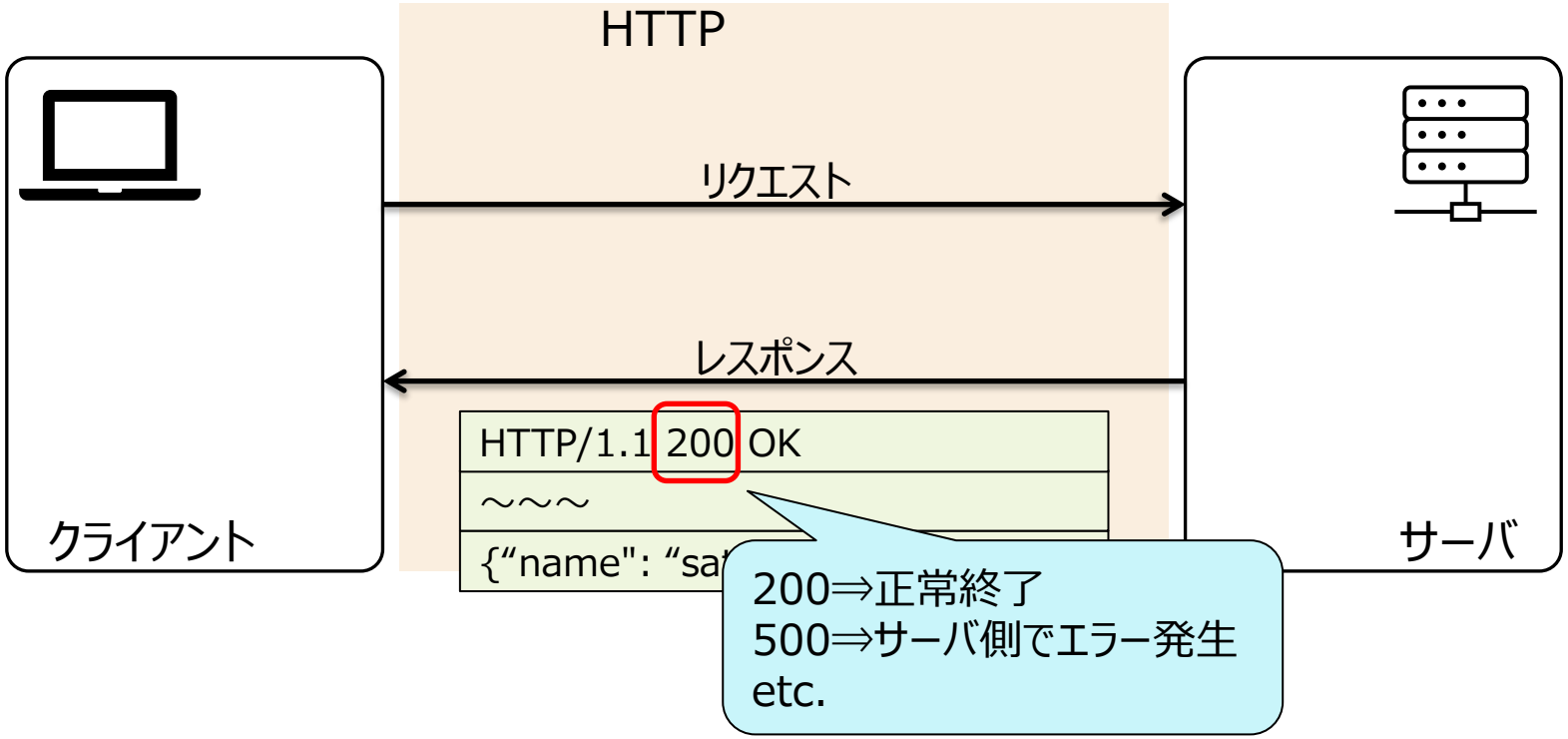
Richardson Maturity Model（http://martinfowler.com/articles/richardsonMaturityModel.html）
をもとに作成。

RESTで考慮すべき点

きちんと
HTTPステータスコード
を使い分けること。

HTTPステータスコードの使い分け

HTTPステータスコードで正常に処理が完了したかを判断できるようにする。



HTTPステータスコードの使い分け例

- 2xx　正常処理
  - 200⇒処理成功
  - 201⇒リソースの作成成功
- 4xx　クライアントに起因するエラー
  - 401/403⇒認証/認可エラー

⇨ クライアントはリトライすべきではない

要求が間違っているので、リトライしても成功することはない。

- 5xx　サーバに起因するエラー
  - 503⇒メンテナンス中や過負荷によって発生するエラー

⇨ クライアントはリトライすると成功する可能性がある

ステータスコードを正しく使い分けていないと…

- 常に200や500しか返さないと
  - クライアントはレスポンスの内容を解析しないと対処方針を決められない。
- 本当はクライアントエラー(精査エラーなど)なのに500を返す（エラーならとりあえず500、のような考え方）と…
  - クライアントに無駄なリトライを促してしまう。

リモート呼び出しの使いどころ

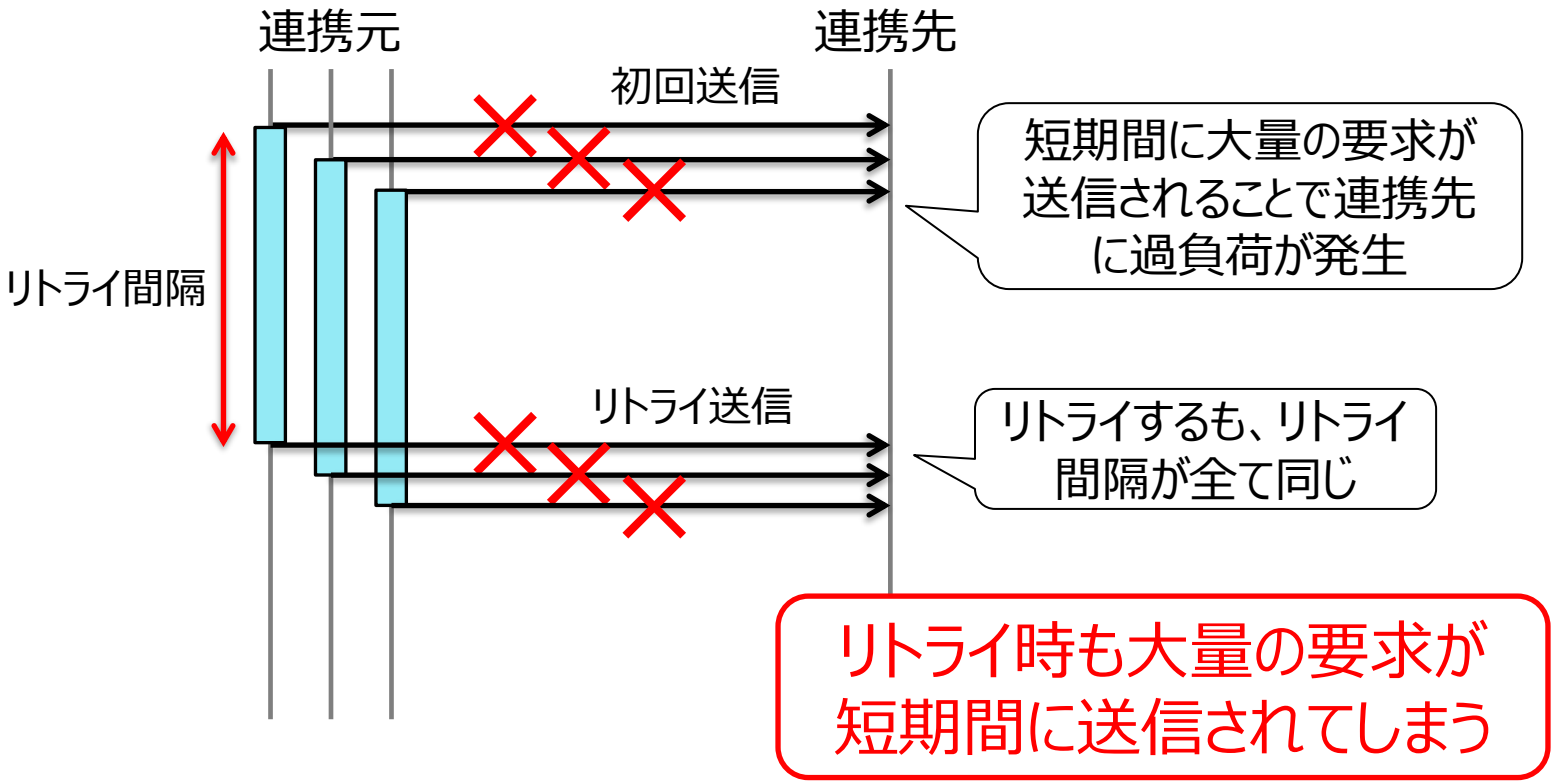
同期処理を実現したい場合。
（最も効率的に同期処理が実現できる）

リモート呼び出しで考慮すべき点

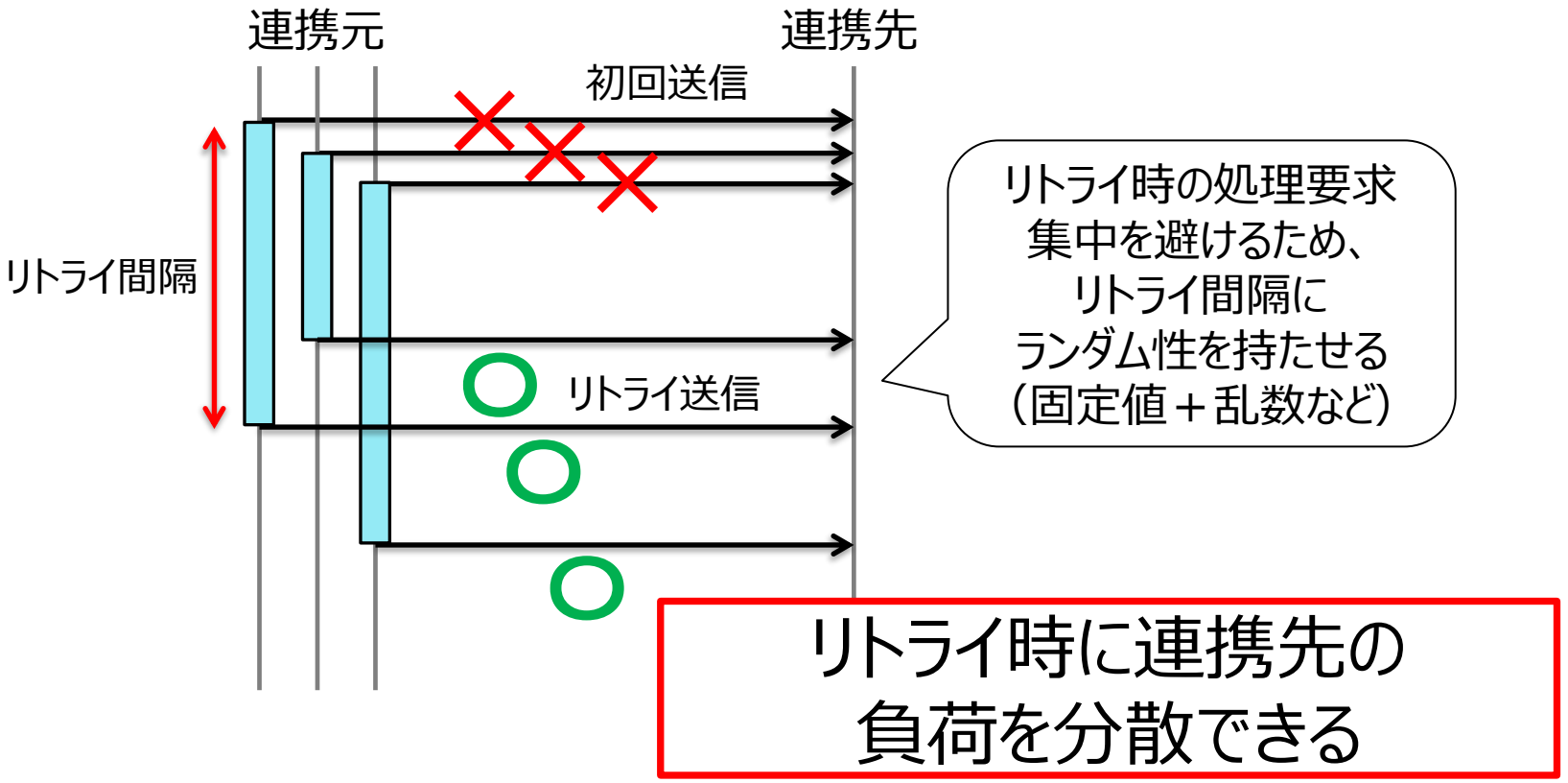
リモート呼び出しで考慮すべき点

- リトライ間隔
- トランザクション間の整合性維持
  - 取消
  - 再送

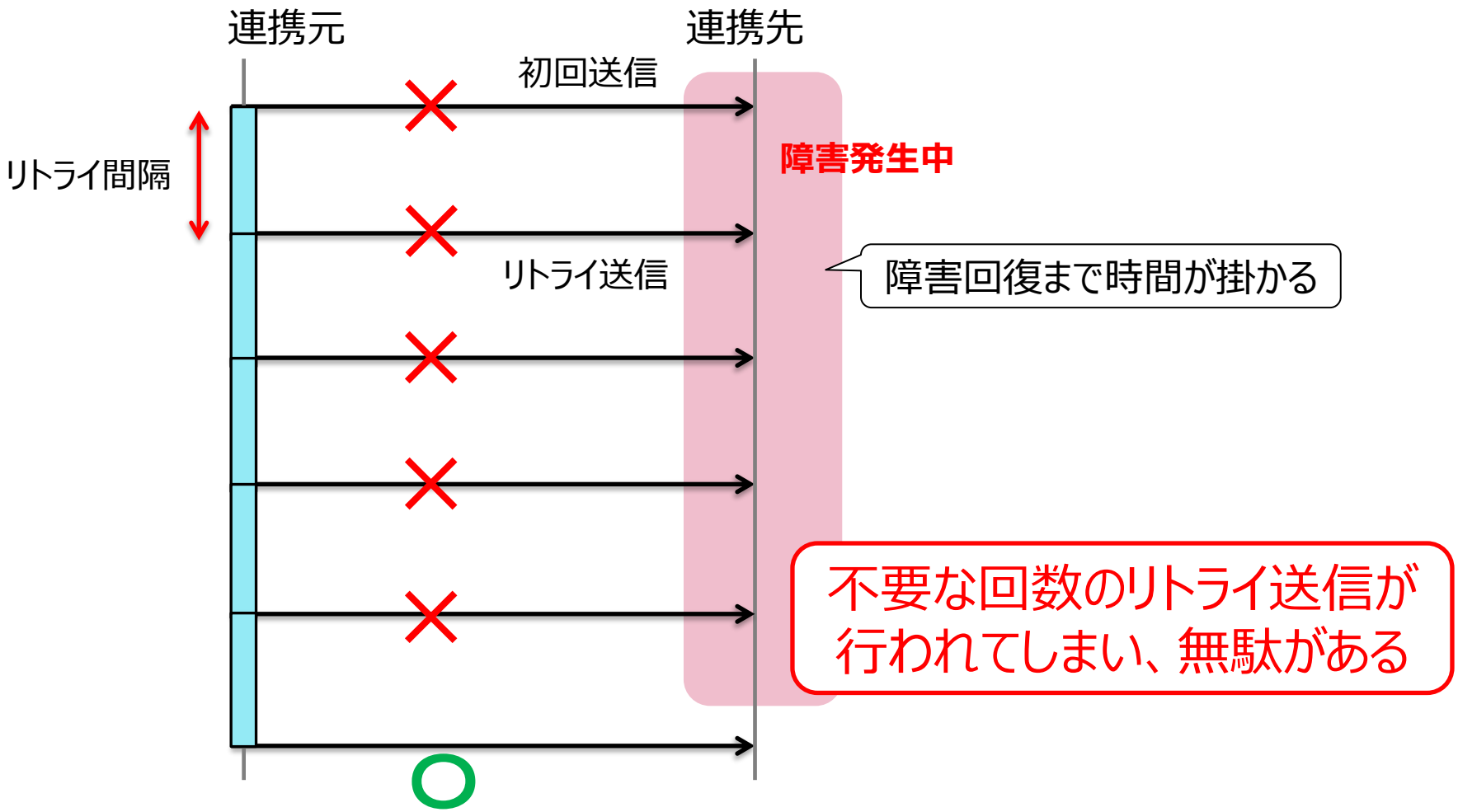
リトライ間隔の設定 (Exponential backoff) (1/4)



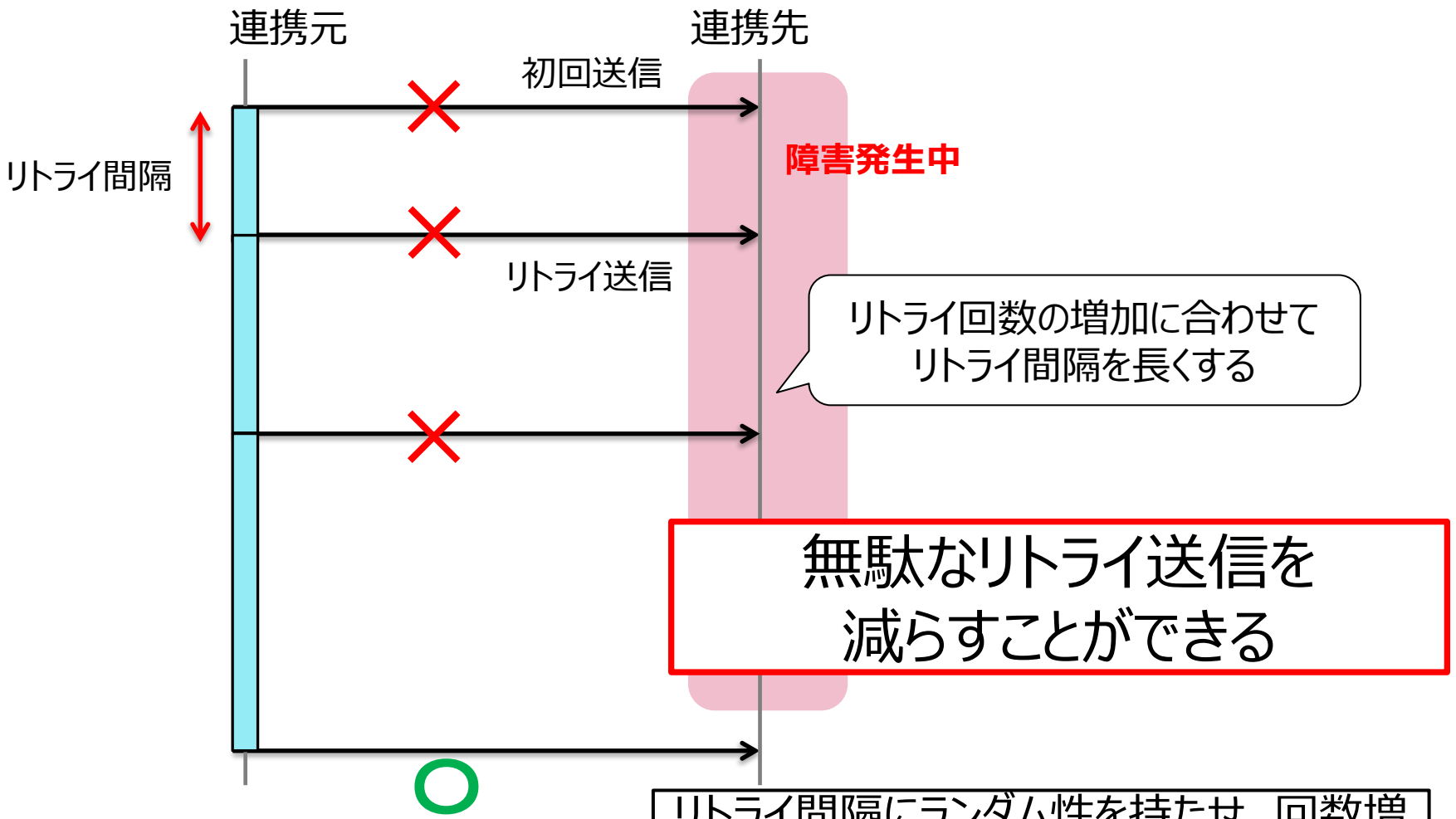
リトライ間隔の設定 (Exponential backoff) (2/4)



リトライ間隔の設定 (Exponential backoff) (3/4)



リトライ間隔の設定 (Exponential backoff) (4/4)

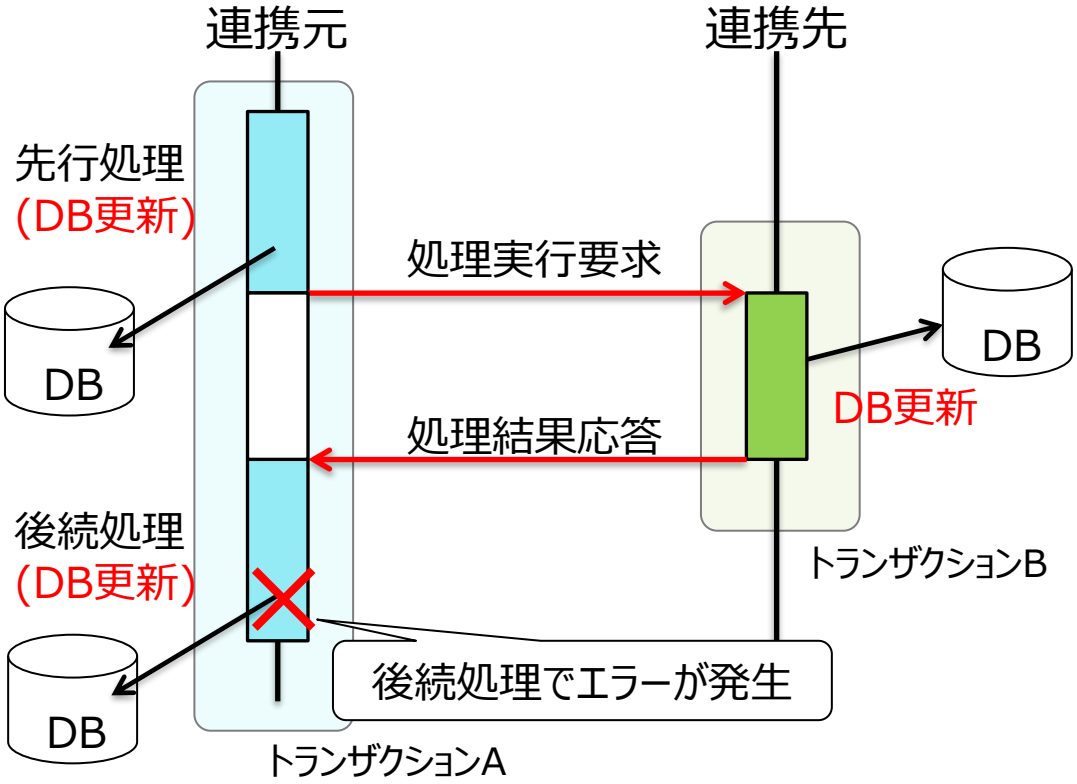


リモート呼び出しで考慮すべき点

- リトライ間隔
- トランザクション間の整合性維持
  - 取消
  - 再送

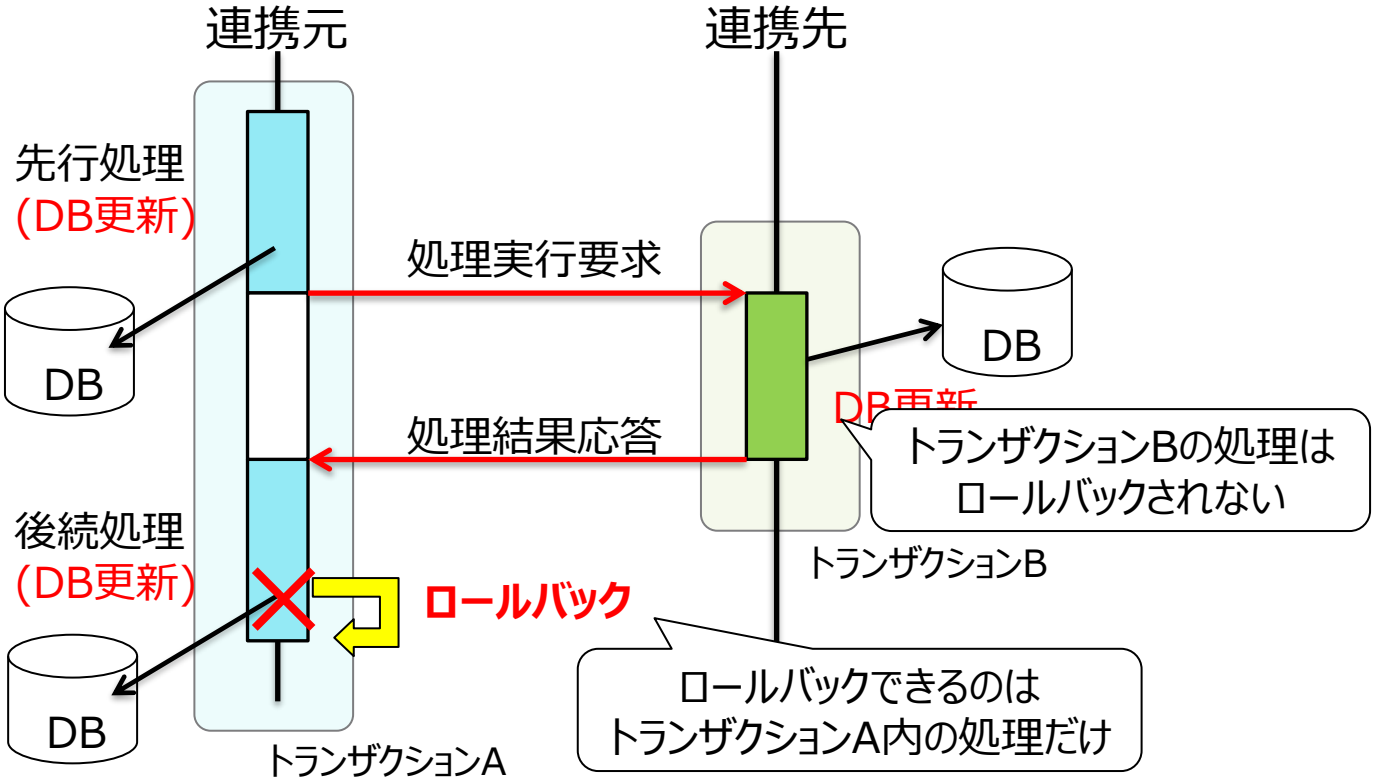
トランザクション間の整合性維持 – 取消(1/6)

連携元と連携先は別システムなので、当然トランザクションが異なる。



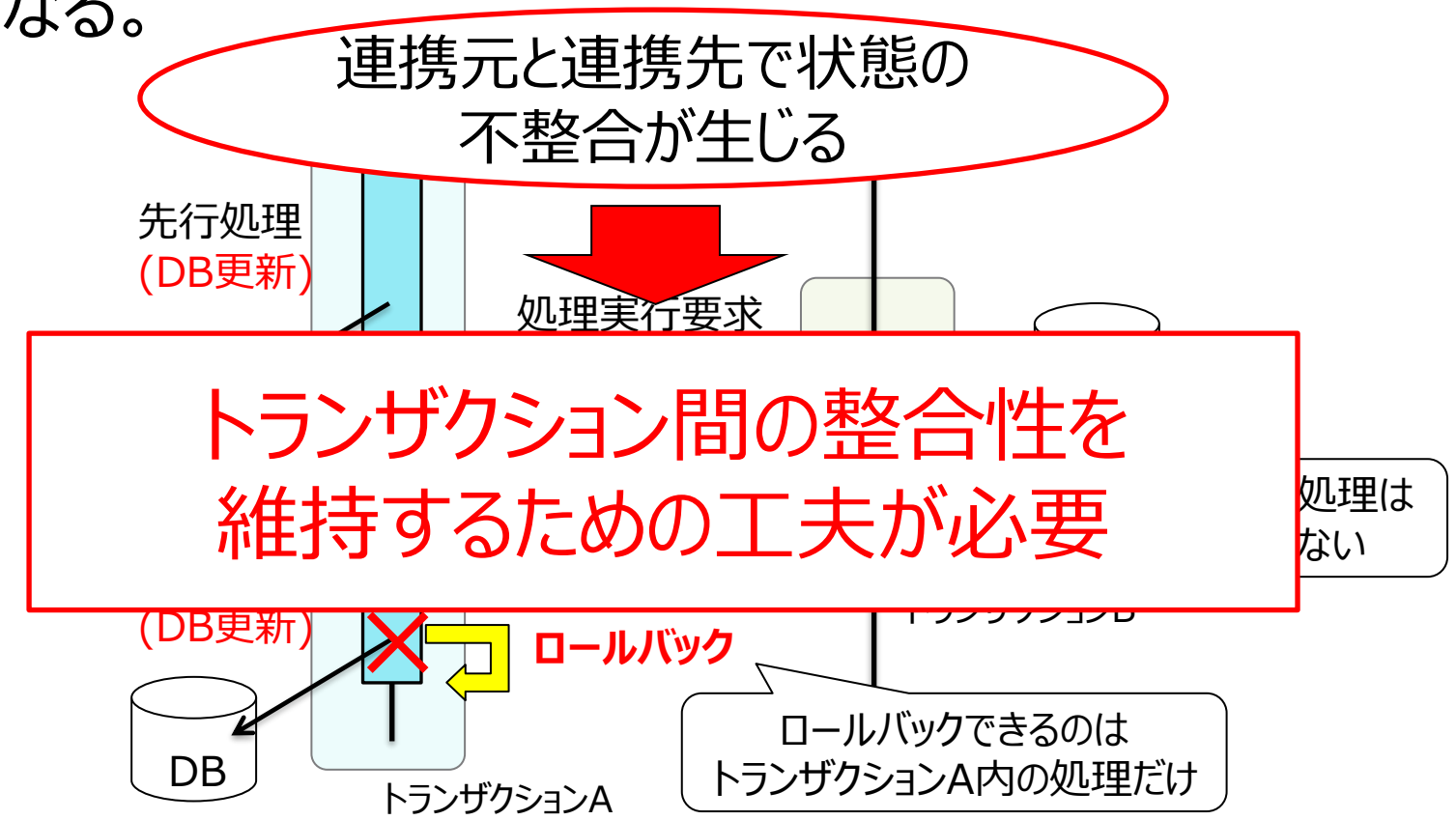
トランザクション間の整合性維持 – 取消(2/6)

連携元と連携先は別システムなので、当然トランザクションが異なる。



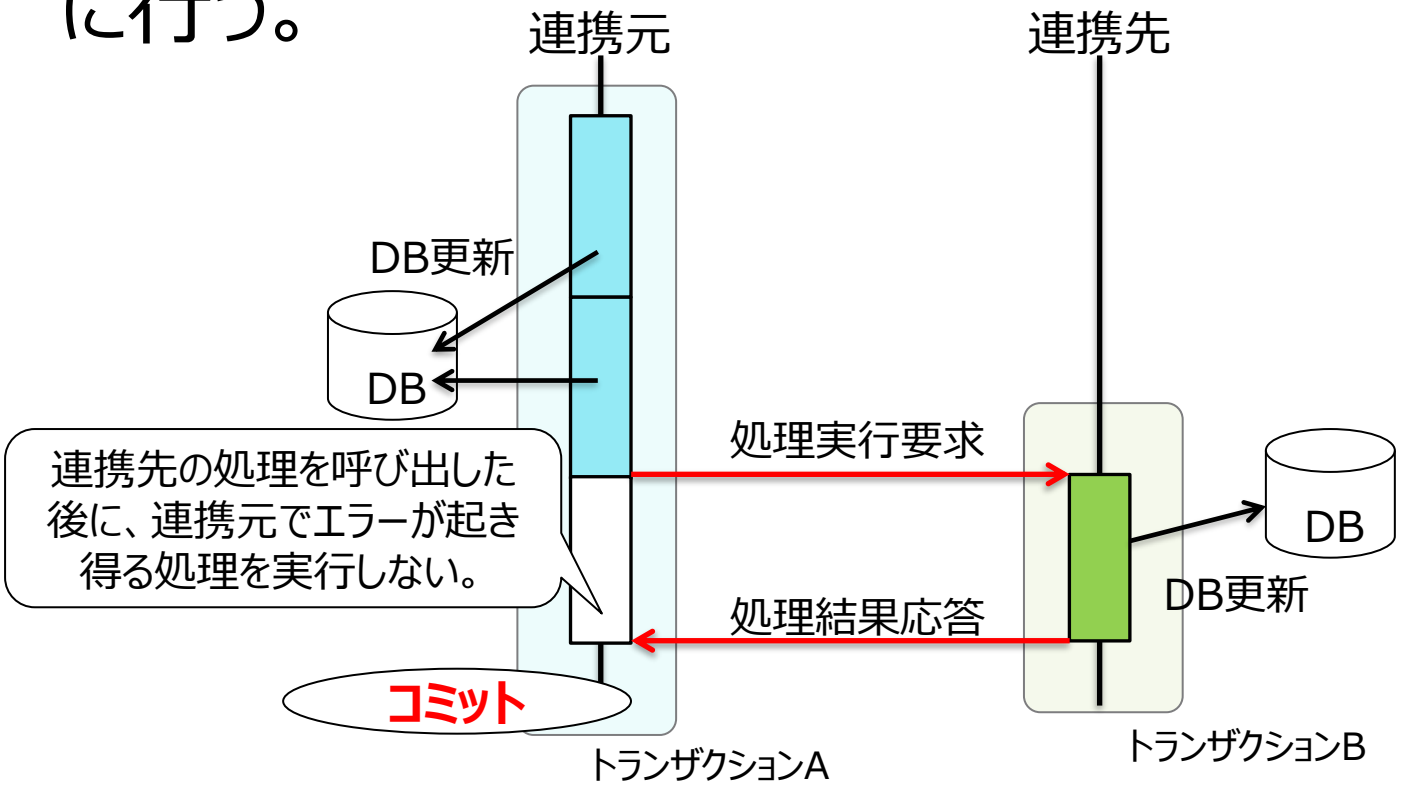
トランザクション間の整合性維持 – 取消(3/6)

連携元と連携先は別システムなので、当然トランザクションが異なる。



トランザクション間の整合性維持 – 取消(4/6)

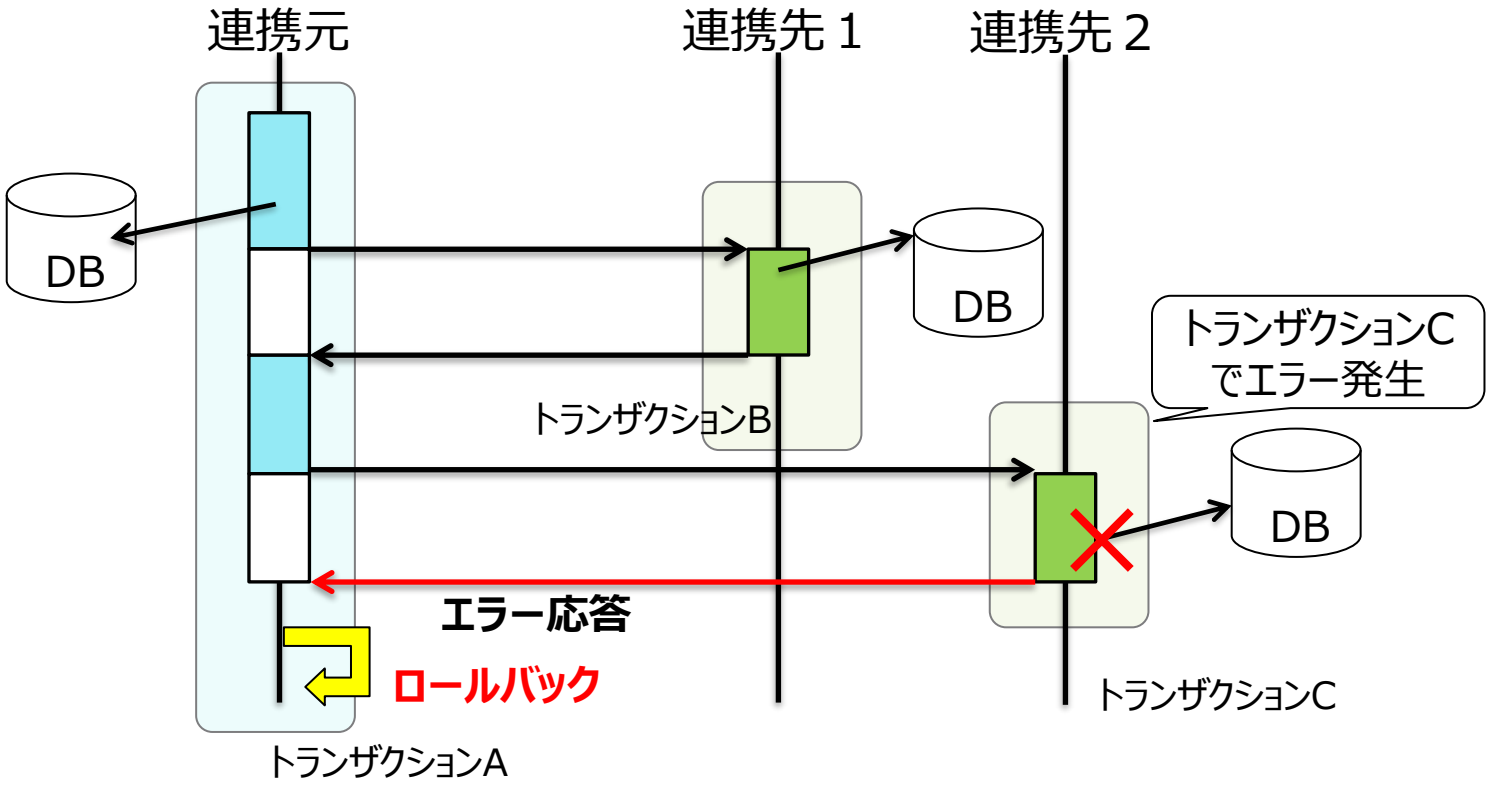
一つの方法として、レスポンスを利用してDB更新をする必要が無い限り、外部への更新要求は最後に行う。



トランザクション間の整合性維持 – 取消(5/6)

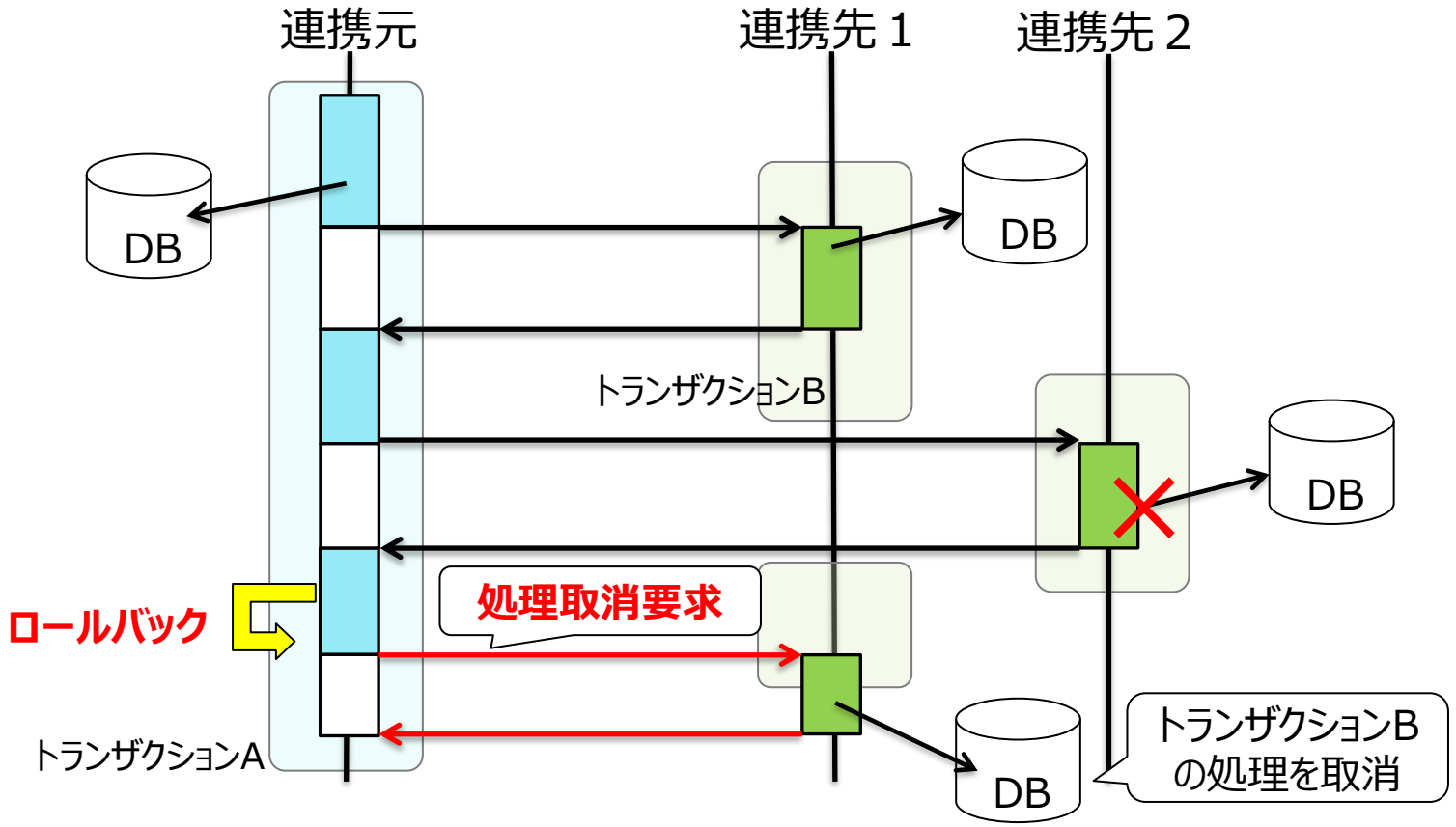
しかし、連携先が複数存在する場合この方法は使えない。

トランザクションA,Cはロールバックできるが、
トランザクションBがロールバックできない。



トランザクション間の整合性維持 – 取消(6/6)

連携先が複数存在する場合、連携先での処理をロールバックさせるためのAPIなどが必要になる。

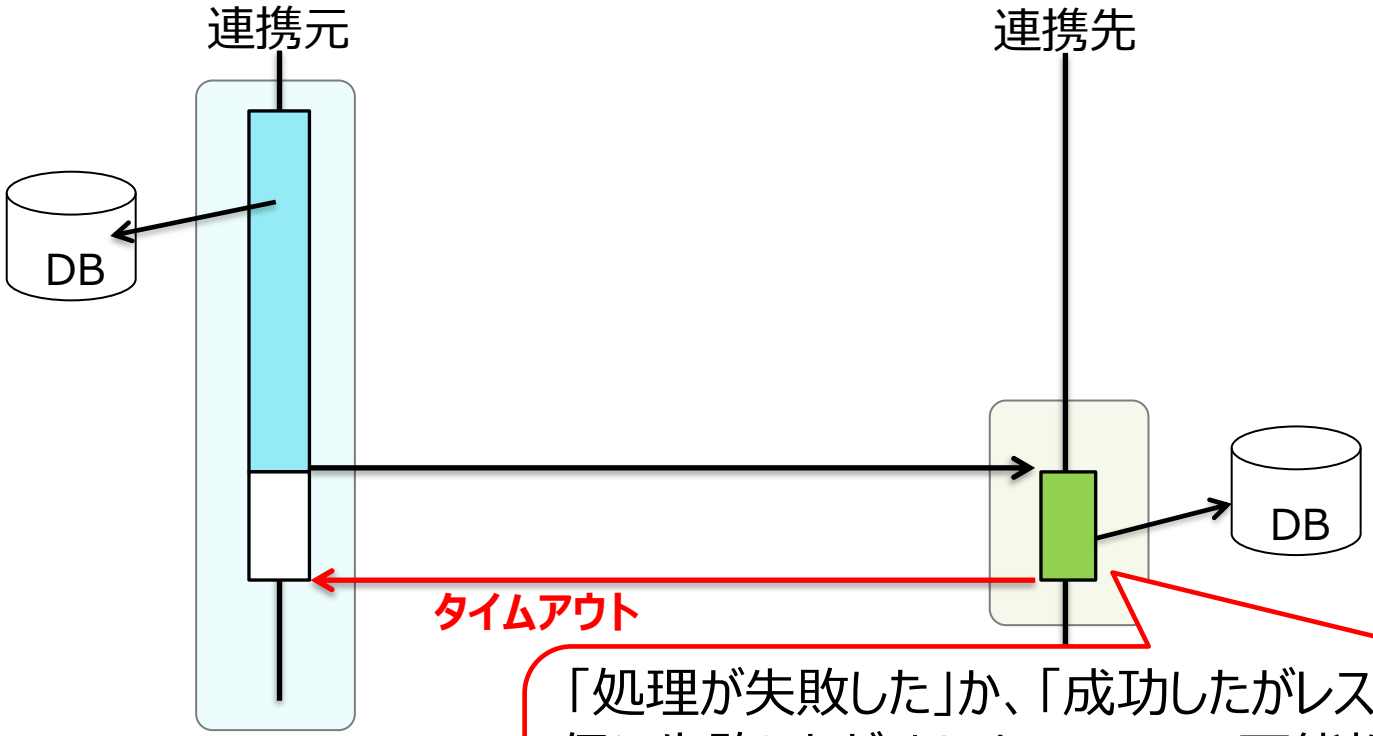


リモート呼び出しで考慮すべき点

- リトライ間隔
- トランザクション間の整合性維持
  - 取消
  - 再送

トランザクション間の整合性維持 – 再送(1/4)

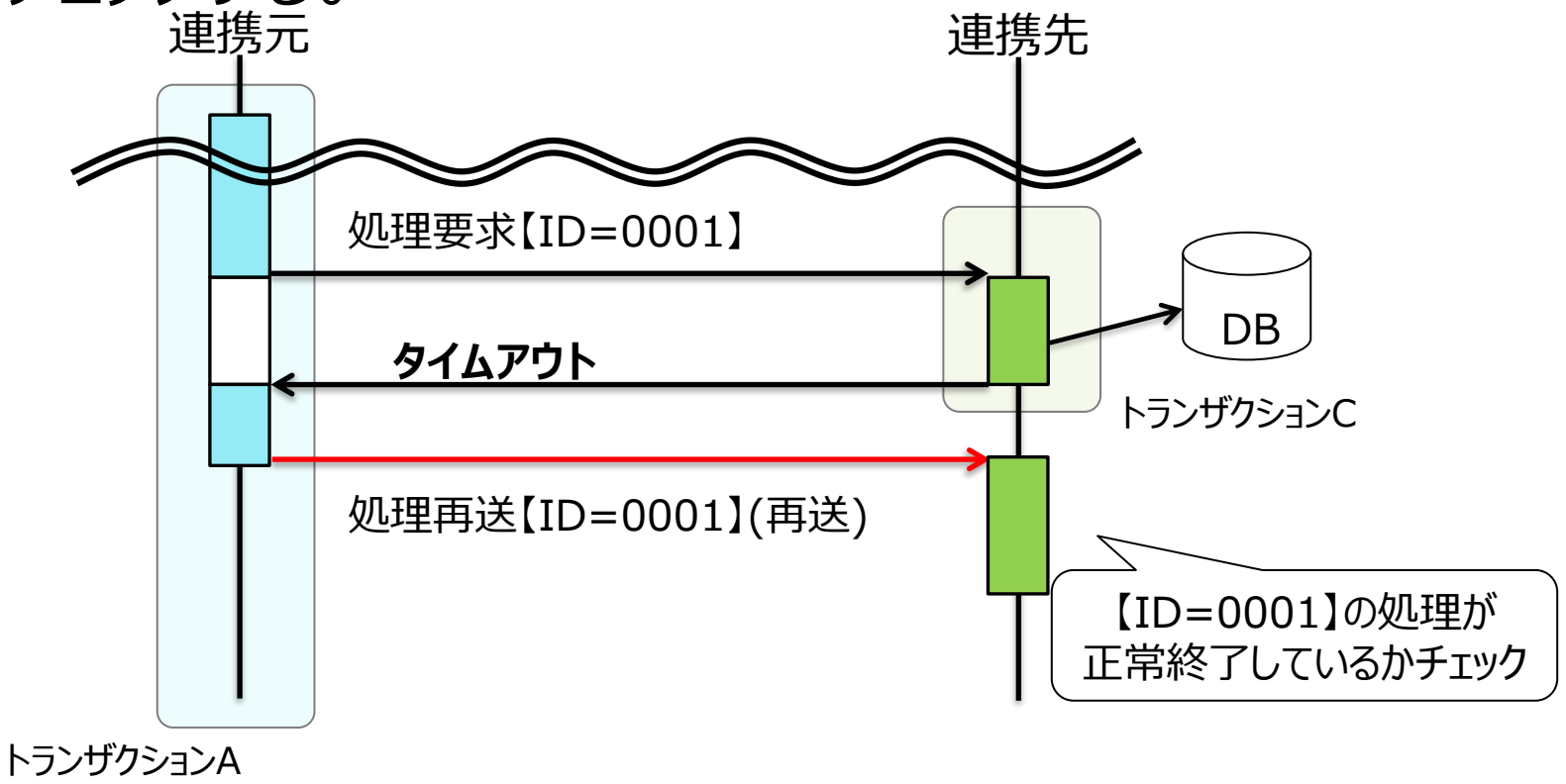
レスポンスのタイムアウト時、誤った制御を行うと
連携先で二重に処理が実行される恐れがある。



「処理が失敗した」か、「成功したがレスポンス送信に失敗しただけ」、という２つの可能性がある。
リトライ時、単純に再処理してしまうと、成功していた場合、二重処理になってしまう。

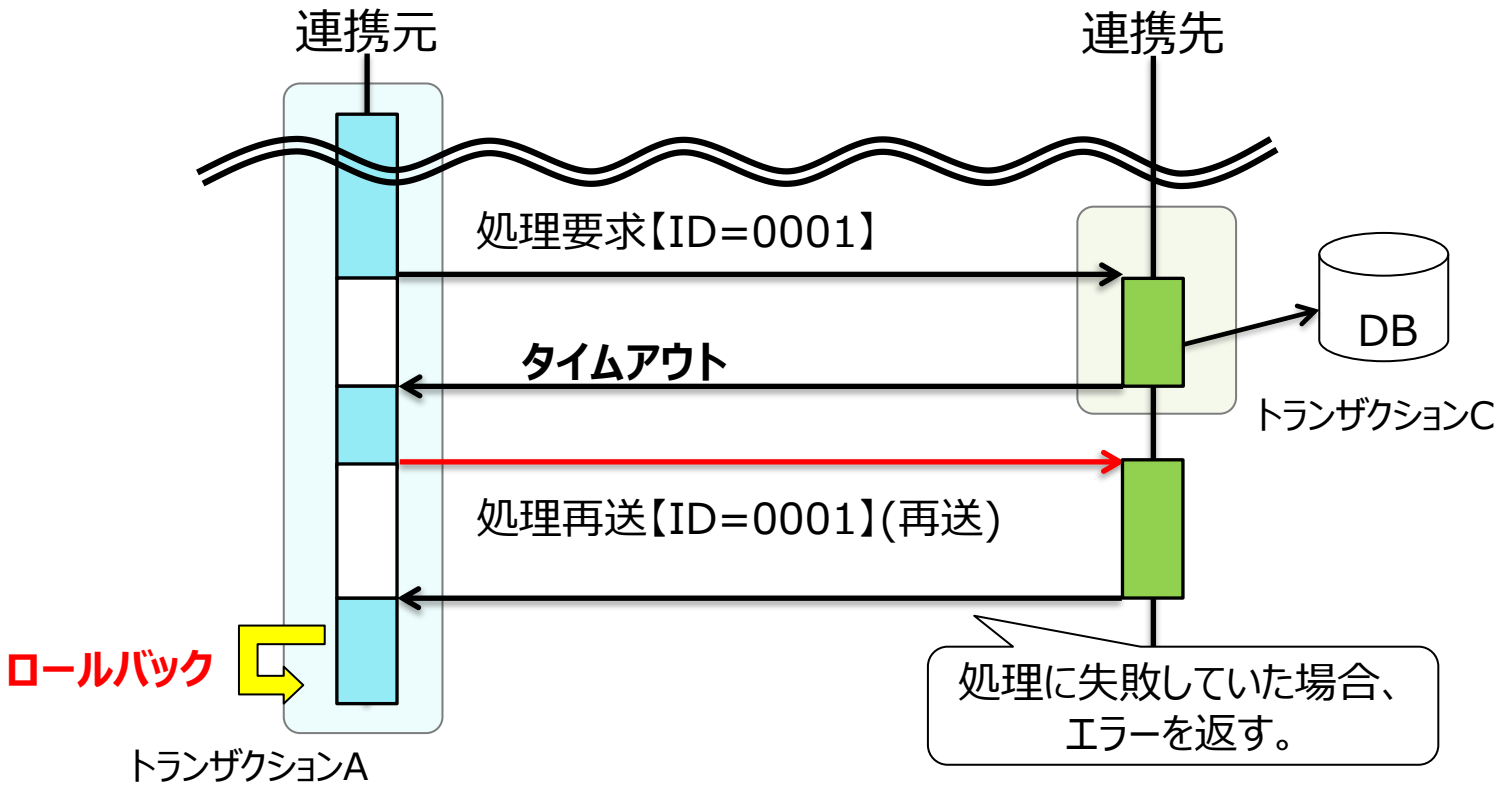
トランザクション間の整合性維持 – 再送(2/4)

タイムアウトした要求と同じ識別子を付与して再送する。
連携先では、送信された識別子の処理が正常終了しているかチェックする。



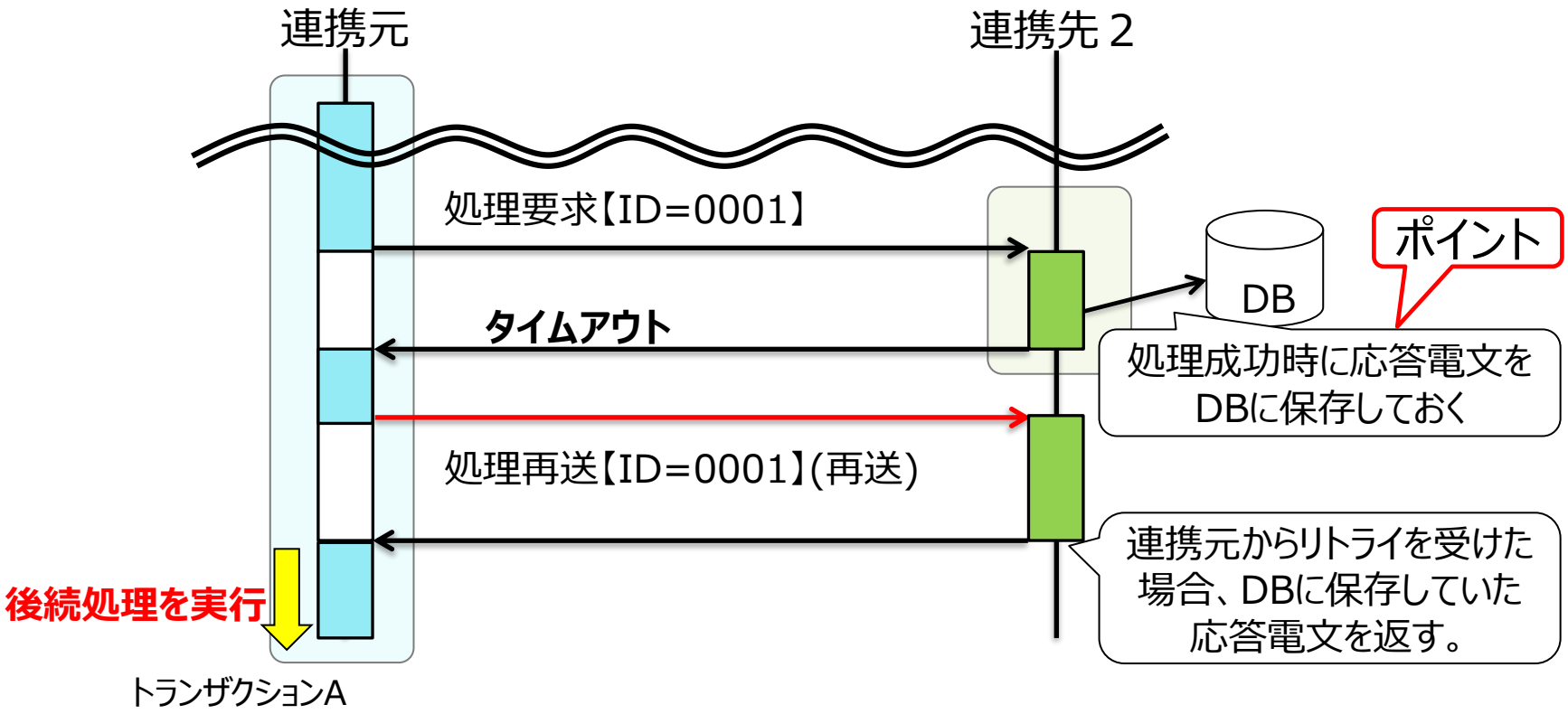
トランザクション間の整合性維持 – 再送(3/4)

連携先が処理に**失敗**していた場合は、エラーを返す。



トランザクション間の整合性維持 – 再送(4/4)

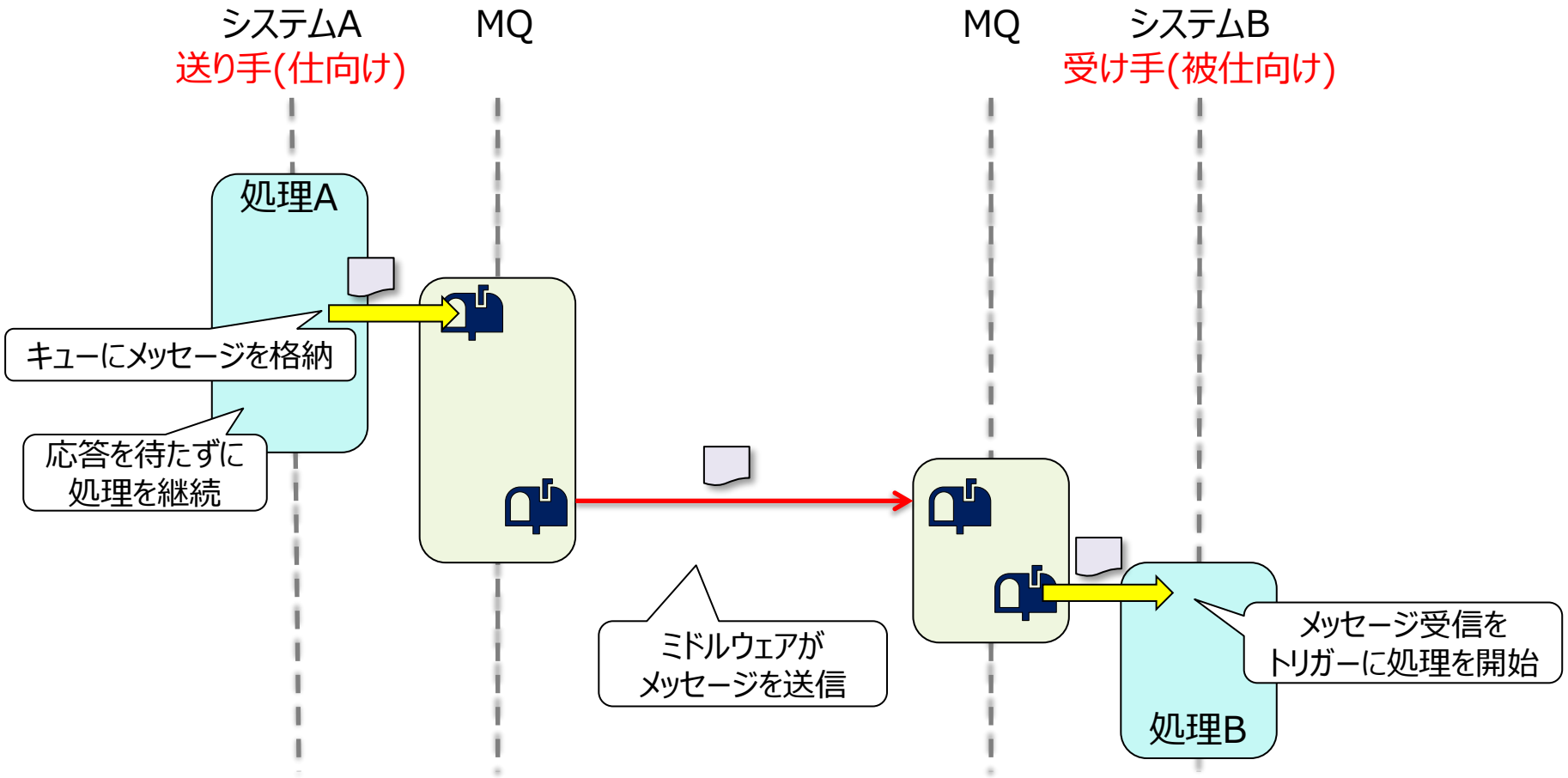
連携先が処理に**成功**していた場合は、処理は実行せず、DBに保存しておいた応答電文を返す。



キューによる連携

キューによる連携とは

メッセージ・キューイングによる非同期通信（相手の応答を待たずに処理を継続する）を行う方式



キューによる連携の特徴

相手システムが停止していても、自システムのアプリケーションは処理を継続できる。
（非同期通信の場合）

⇒可用性が高い

メッセージキューイングミドルウェア

メッセージング・キューイングの機能を提供するミドルウェアのこと。
以下のようなソフトウェアが存在する。

- IBM WebSphere MQ
- Oracle WebLogic (JMS)
- Pivotal Rabbit MQ
- Apache ActiveMQ
- RedHat（JBoss） HornetQ (JMS)

…

【補足】キュー連携による同期応答通信(1/2)

メッセージ・キューイングの真価が発揮されるのは非同期通信だが、実際には同期応答通信（相手の応答を待って、処理を継続する）で利用されることも多い。

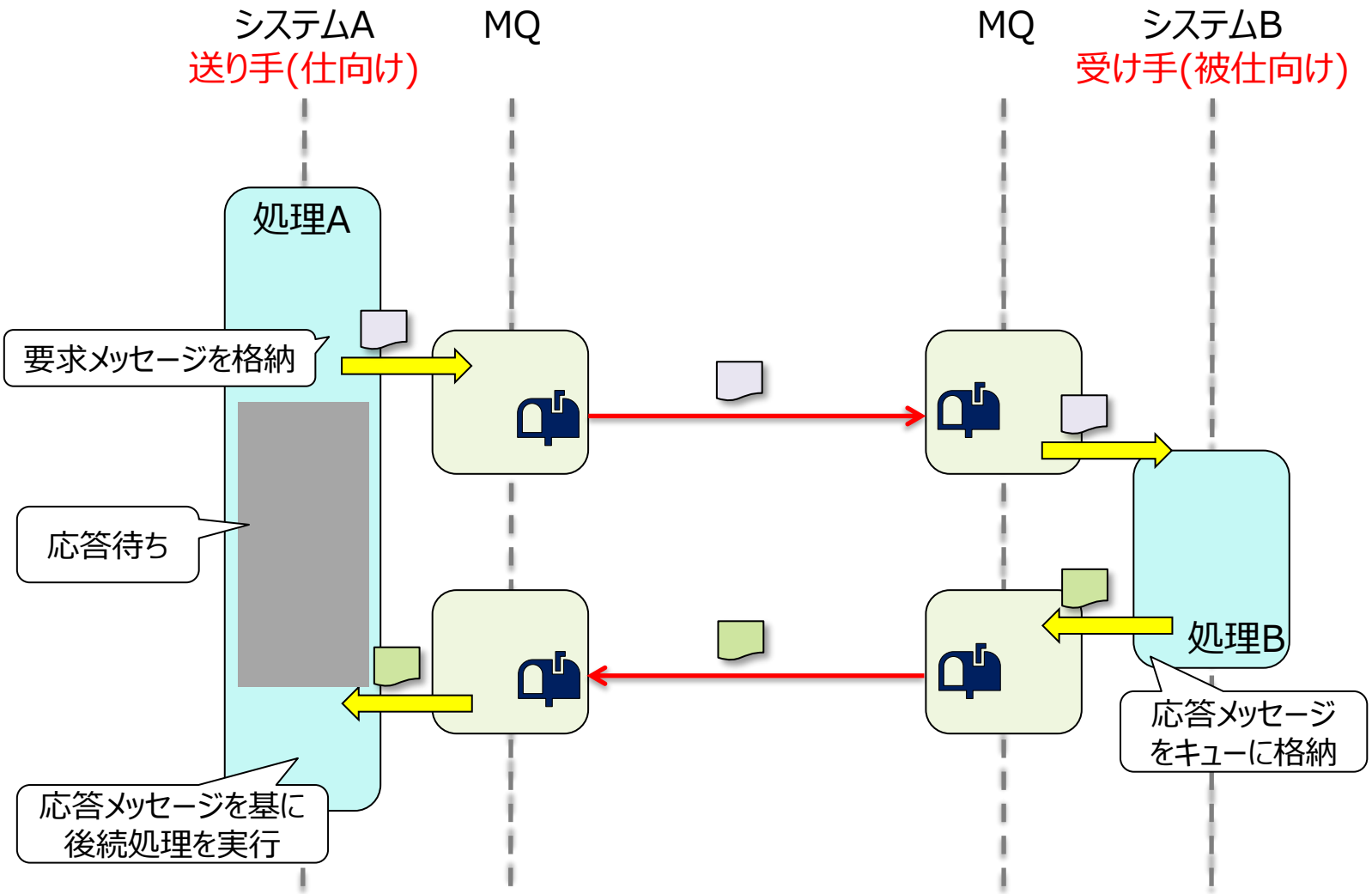
しかし同期通信にしてしまうとメリットが失われる。

- キュー連携ならば非同期通信が実現できること。
- 同期通信ならリモート呼び出しの方が簡単、低コストで実現できる。

⇨ 高価なミドルウェアを購入するのは割に合わない

【補足】キュー連携による同期応答通信(2/2)

同期応答通信の場合の処理フロー

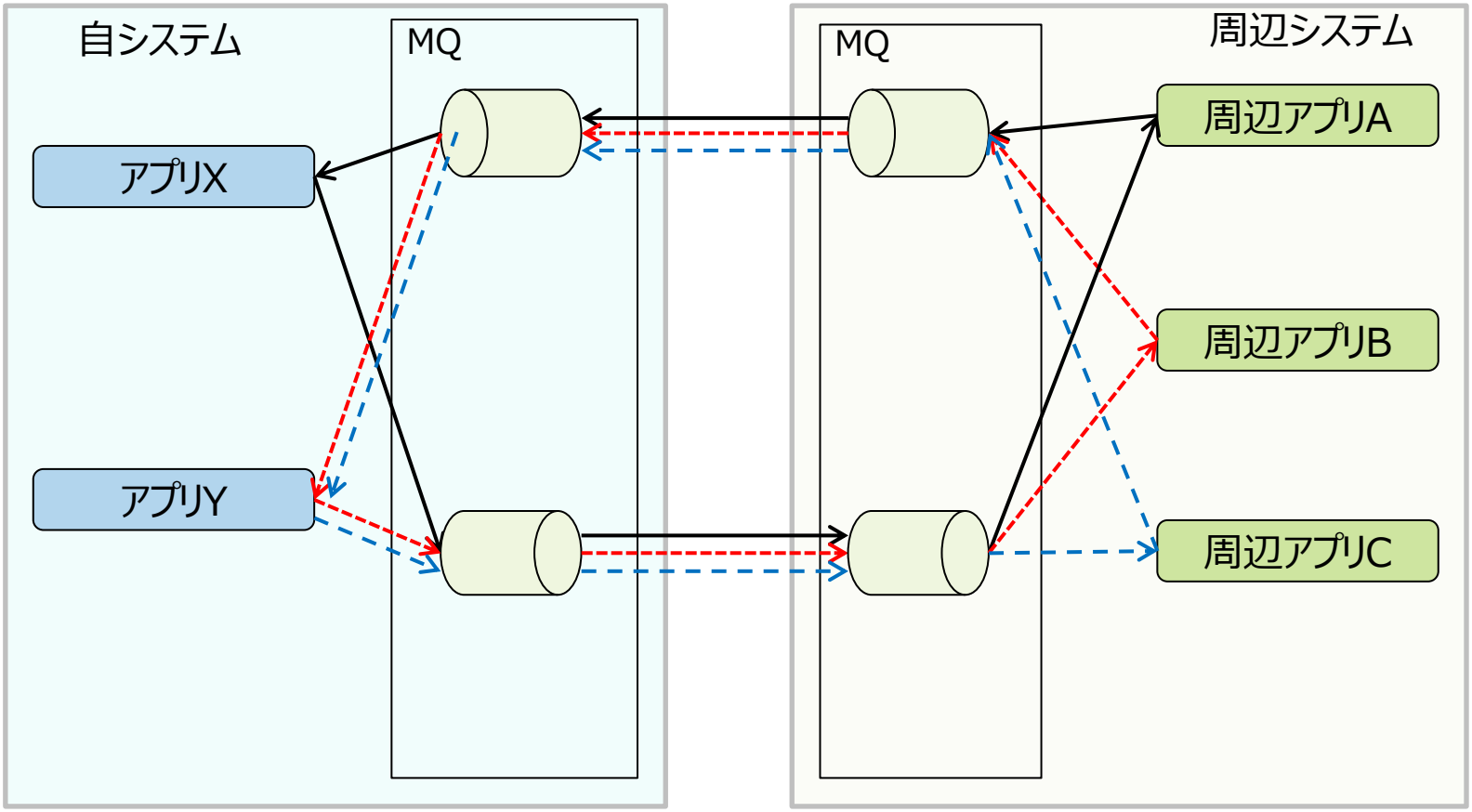


キューによる連携で考慮すべき点

- キューの構成
- エラー時の対処
- 同期応答通信での考慮点

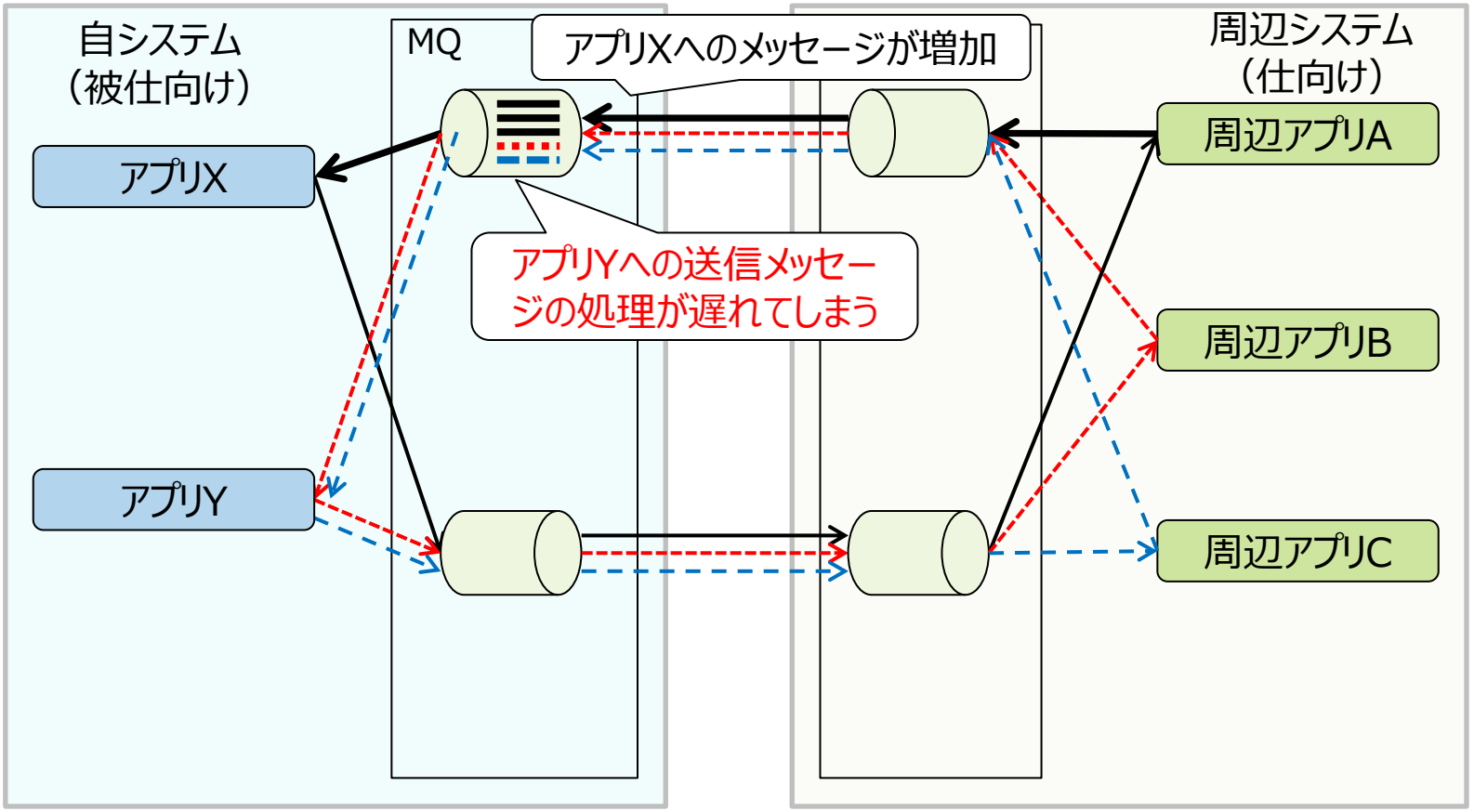
キューの構成(1/6)

各アプリで共通のキューを使用する場合、2つ課題が発生する。



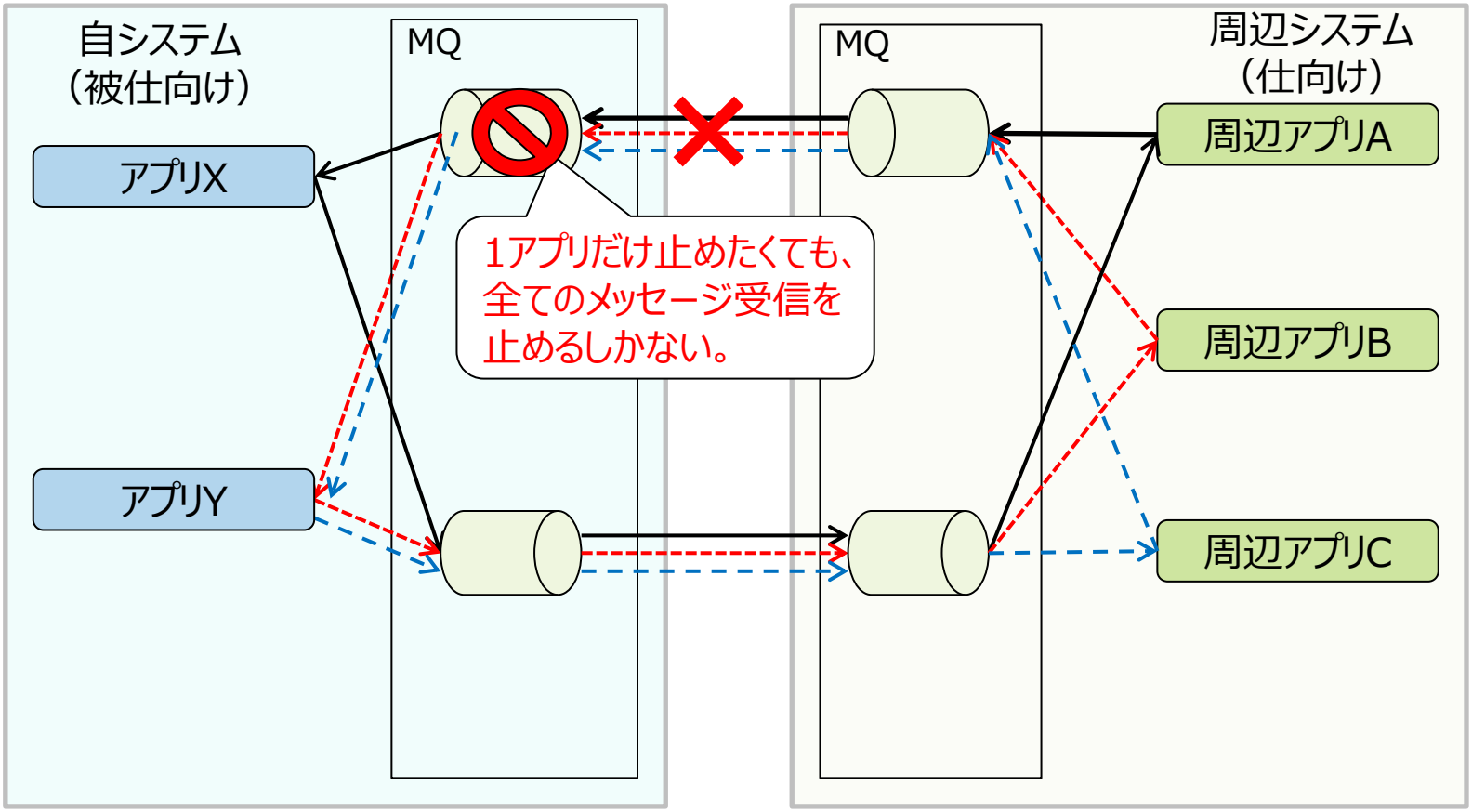
キューの構成(2/6)

1つ目： １アプリへの負荷増加が、全体の
スループットに影響を与えてしまう。



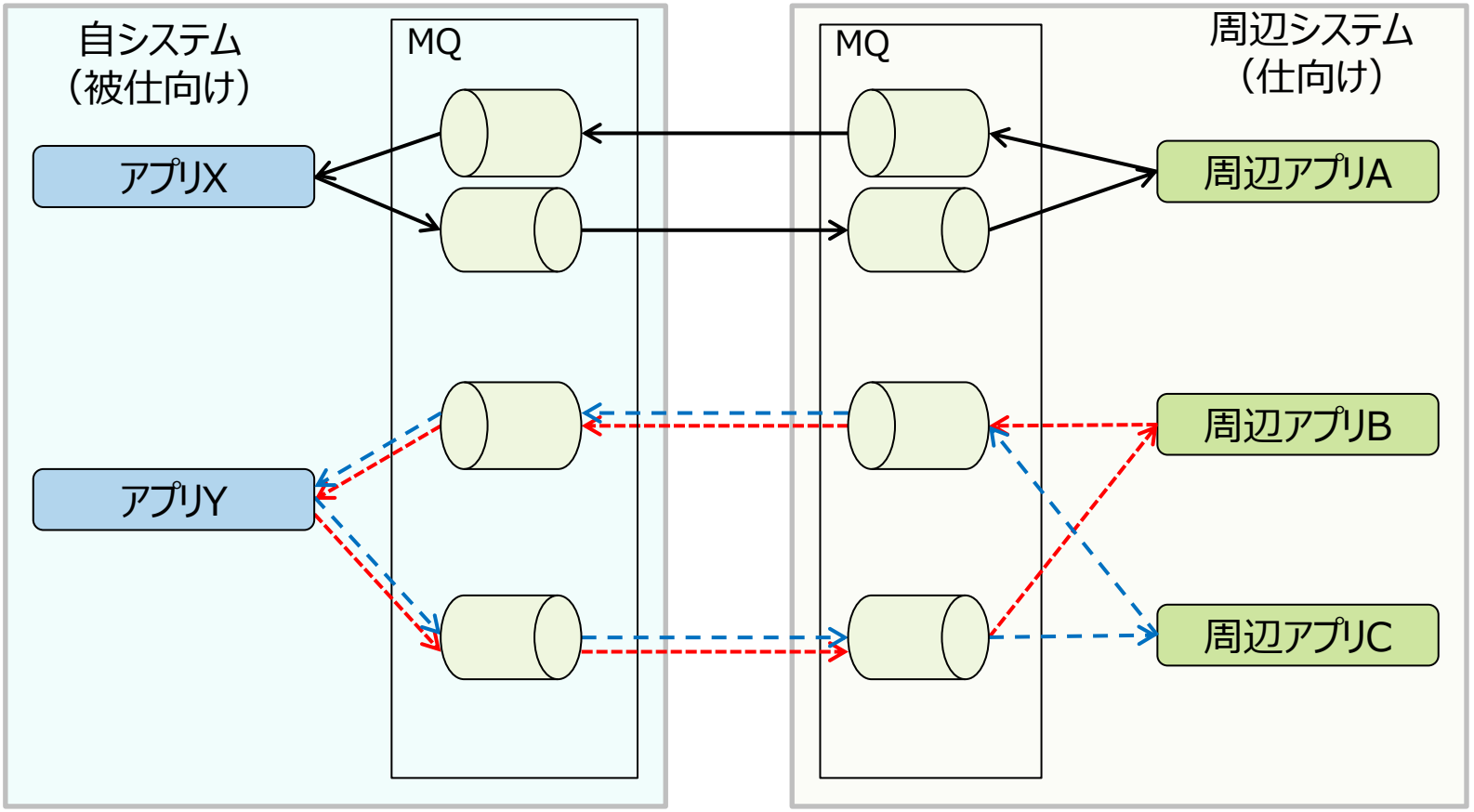
キューの構成(3/6)

2つ目： 1アプリだけを閉局するためにメッセージ
受信を止めるなどの柔軟な対応ができない。



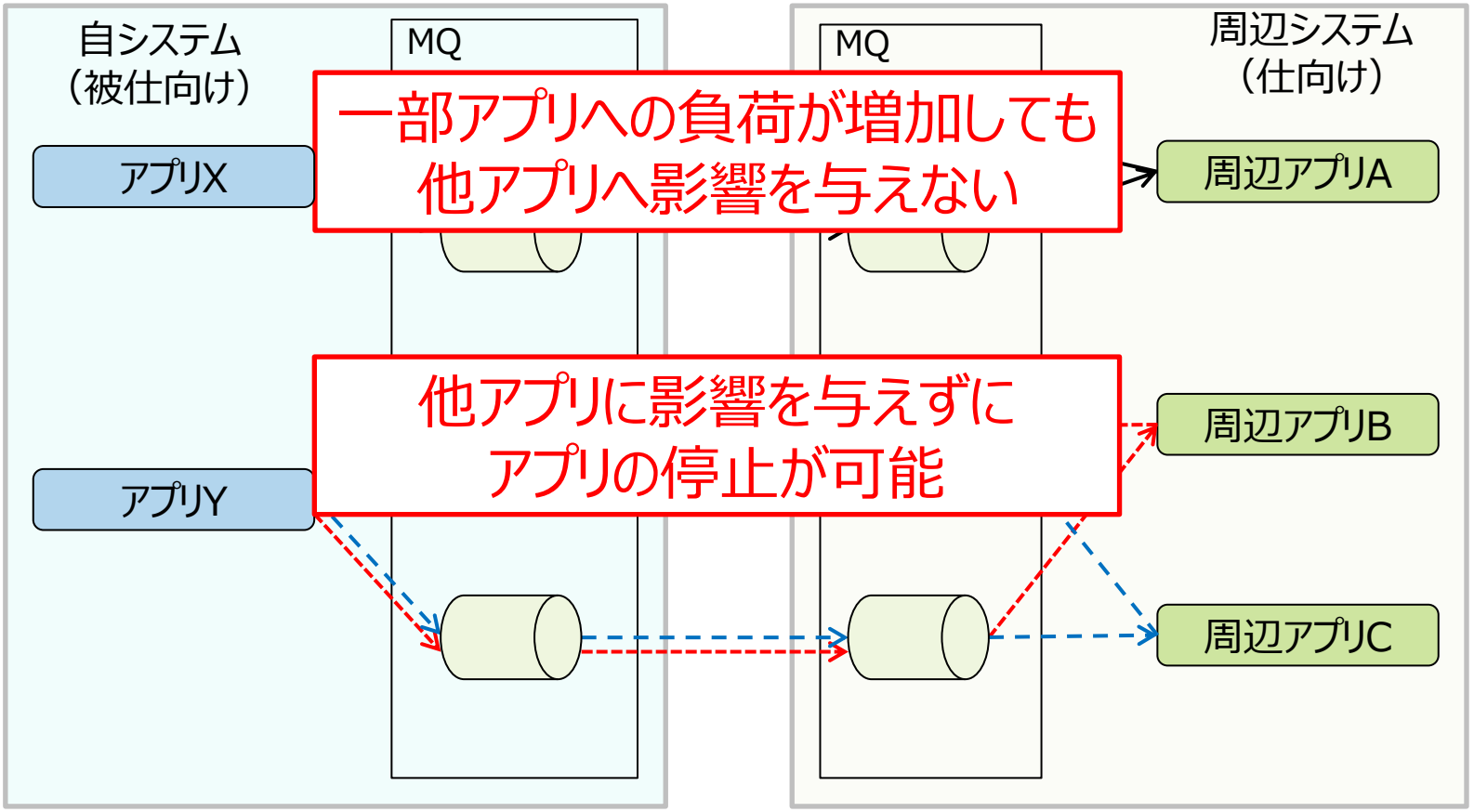
キューの構成(4/6)

対策：アプリ単位でキューを分割。



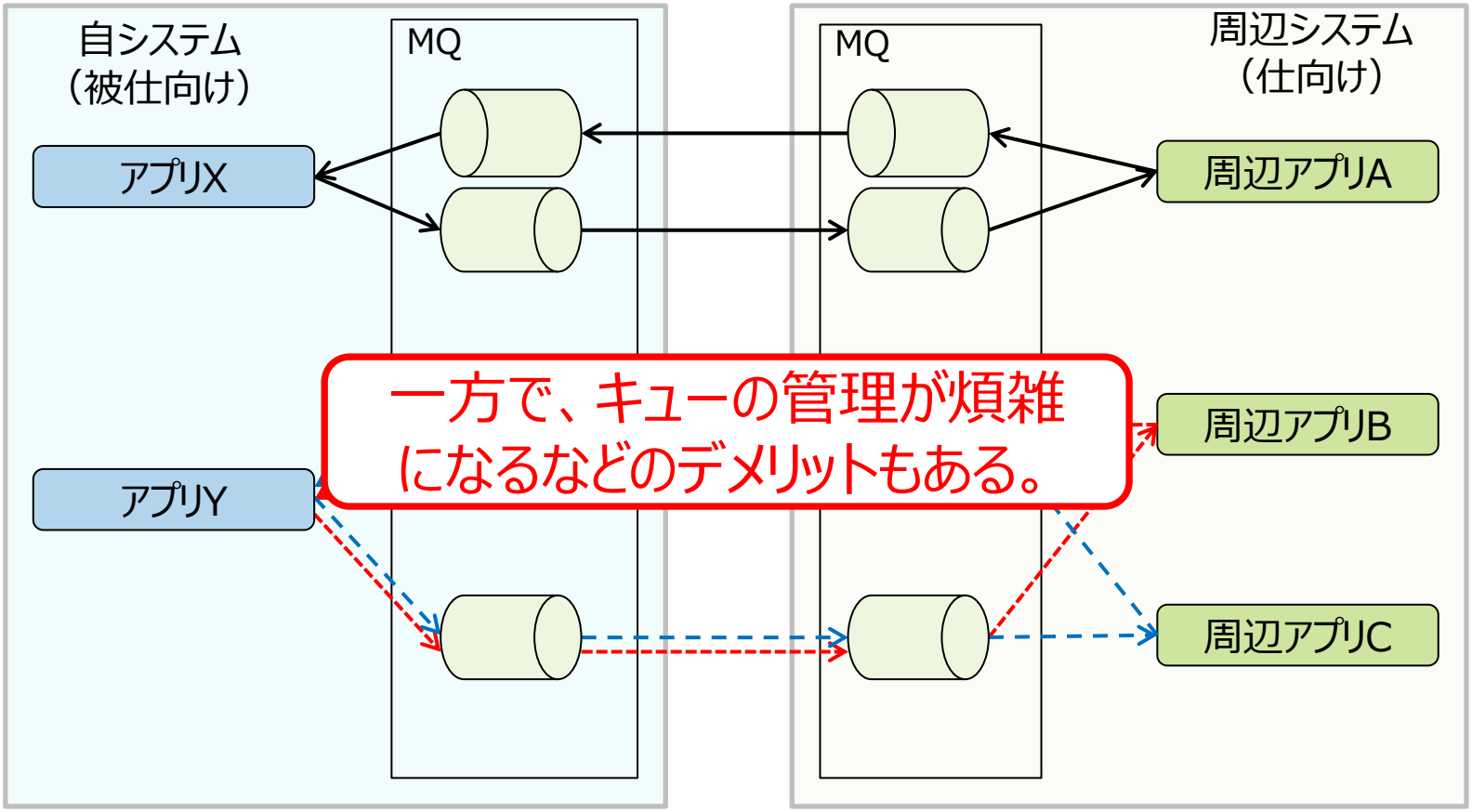
キューの構成(5/6)

対策：アプリ単位でキューを分割。



キューの構成(6/6)

対策：アプリ単位でキューを分割。

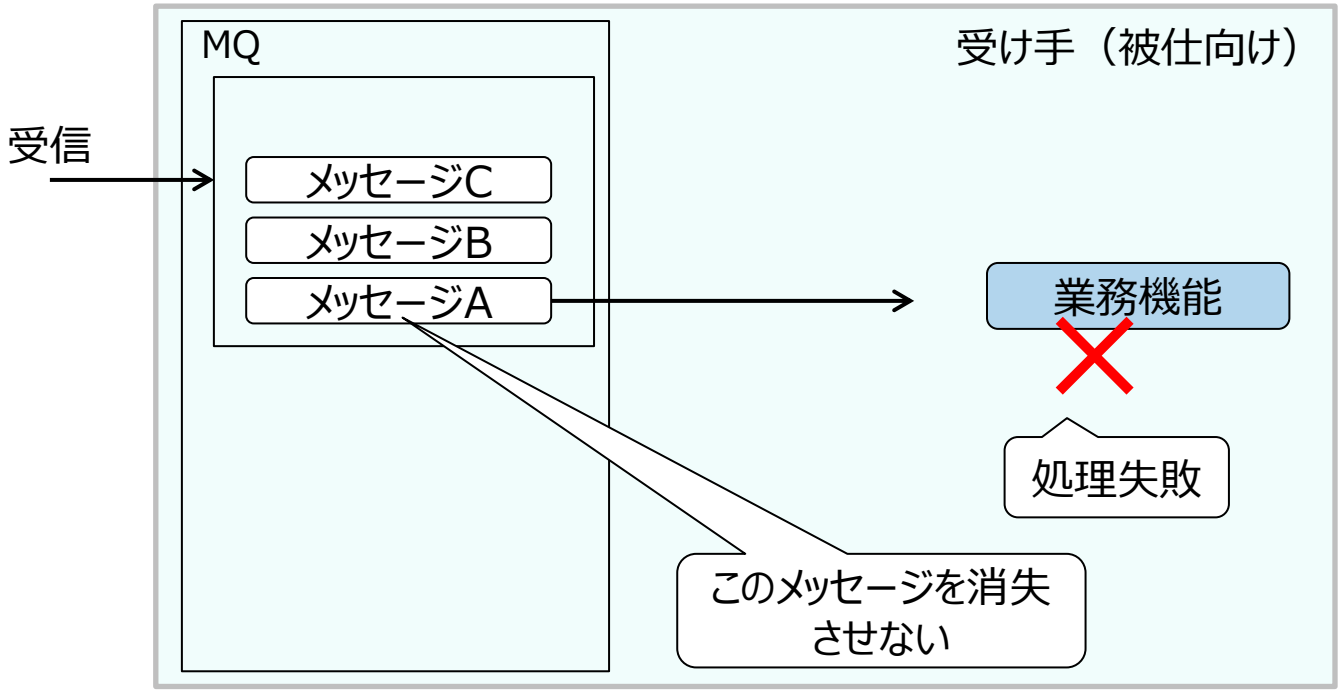


メッセージ・キューイングで考慮すべき点

- キューの構成
- エラー時の対処
- 同期応答通信での考慮点

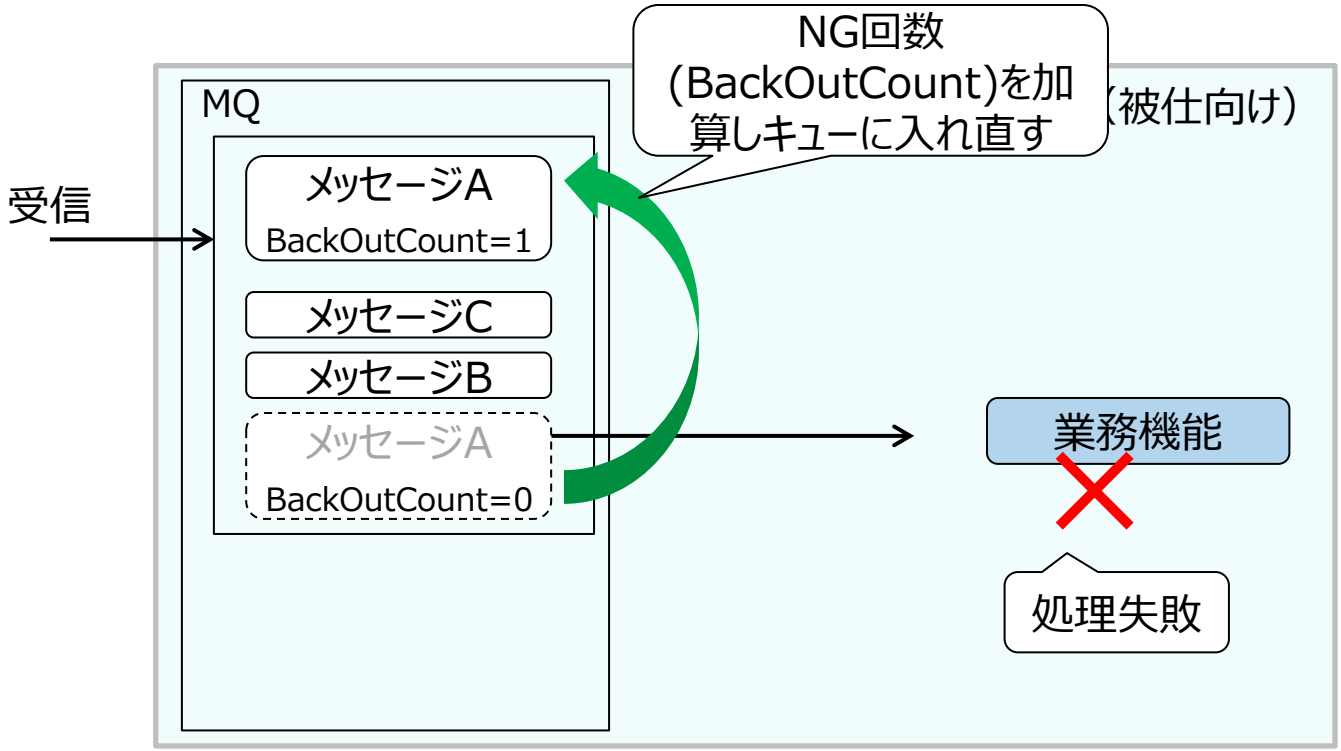
エラー時の処理(1/3)

送信されたメッセージと同じものを再送してもらうことは、相手側でそのメッセージの特定が難しいなどの理由から通常困難。
このため、取込処理に失敗した場合、リカバリや障害調査の為、そのメッセージが消失しないようにする必要がある。



エラー時の処理(2/3)

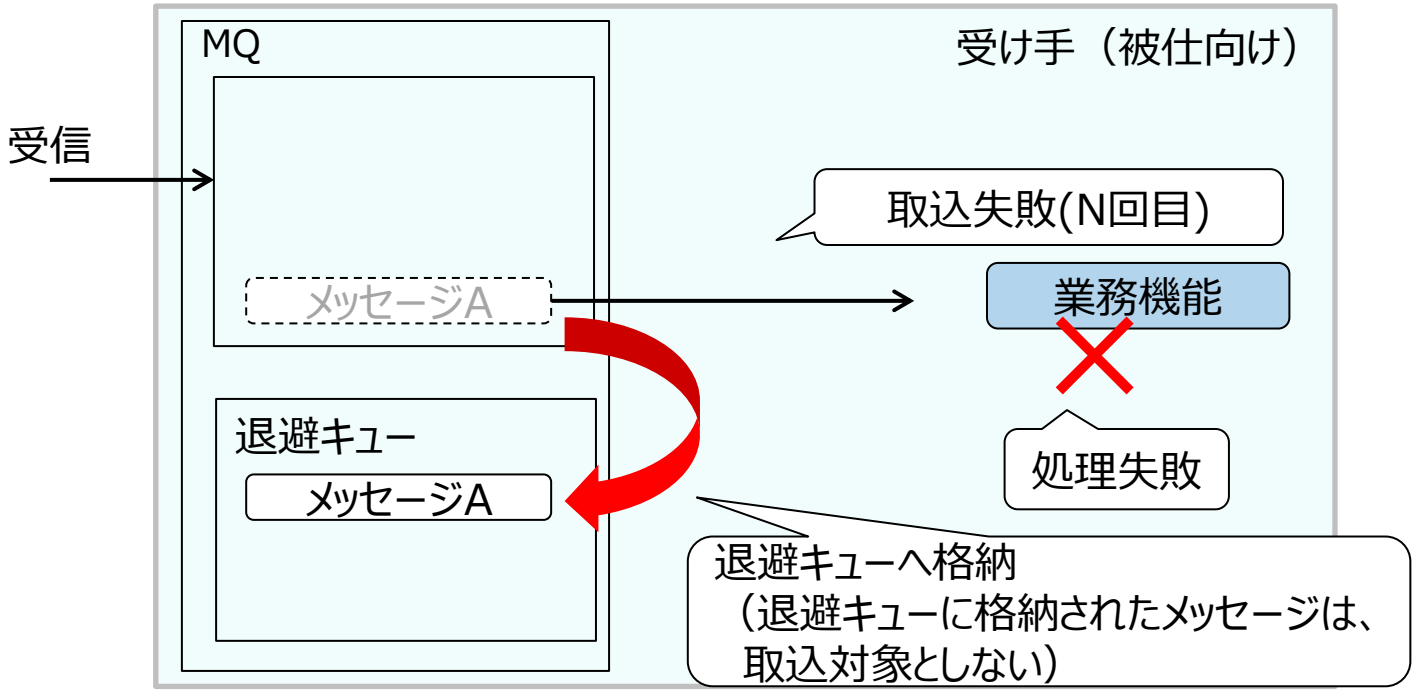
まず、取込処理に失敗したメッセージをキューに戻し再送信させる。



エラー時の処理(3/3)

一定回数以上取込処理が失敗した場合は、
メッセージを退避キューへ格納し保持する。

⇒障害調査やリカバリが可能となる。



メッセージ・キューイングで考慮すべき点

- キューの構成
- エラー時の対処
- 同期応答通信での考慮点

  同期応答通信の場合は、リモート呼び出しと同様の考慮が必要になる。
  - リトライ間隔
  - トランザクション間の整合性維持

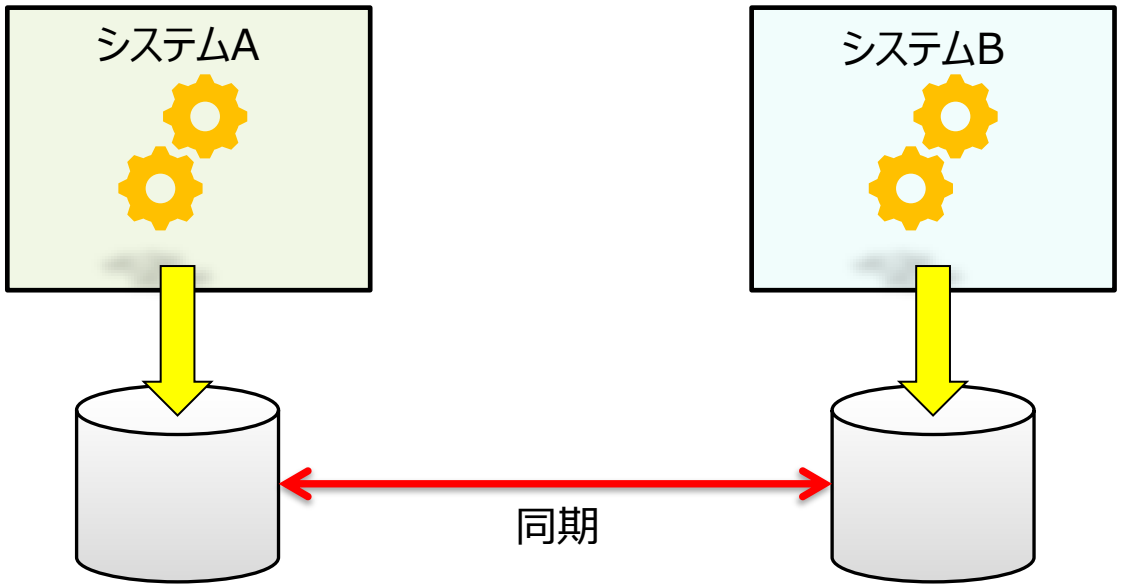
キューによる連携の使いどころ

- 非同期処理を実現したい場合。
- 連携先がメッセージキューイングミドルウェアを利用している場合（キューによる連携しか方法がない場合）。

システム間連携の理論（CAP定理）

CAP定理(1/6)

- 複数のシステムが共通のデータを持つとどのようなトレードオフが生じるかを表す定理。
- 何を捨て、何を取るべきかを整理しやすくなる。



CAP定理(2/6)

下記の3つの要素の頭文字を取って
「CAP定理（きゃっぷていり）」

- Consistency（一貫性）
- Availability（可用性）
- Partition-tolerance（分断耐性）

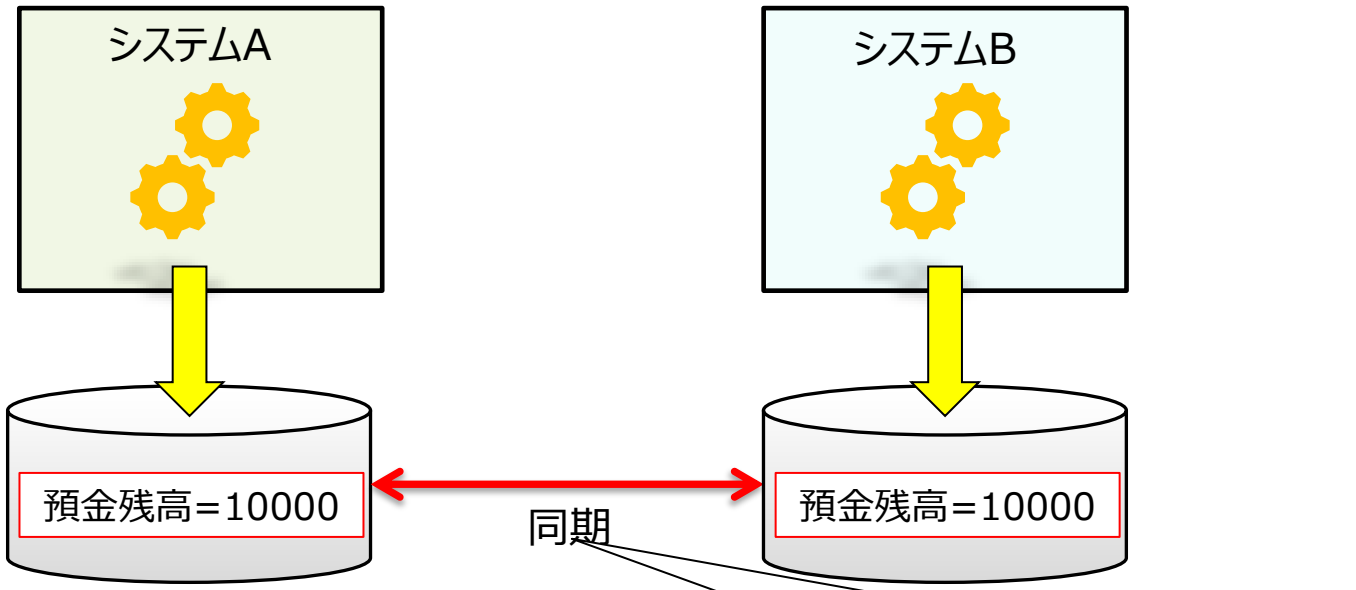
複数のシステムが共通のデータを分散して持つとき、
「**一貫性、可用性、分断耐性の3つを同時に全て100%にすることはできない**」という定理。

各要素の詳細は次ページ以降で説明。

CAP定理(3/6)

Consistency
一貫性

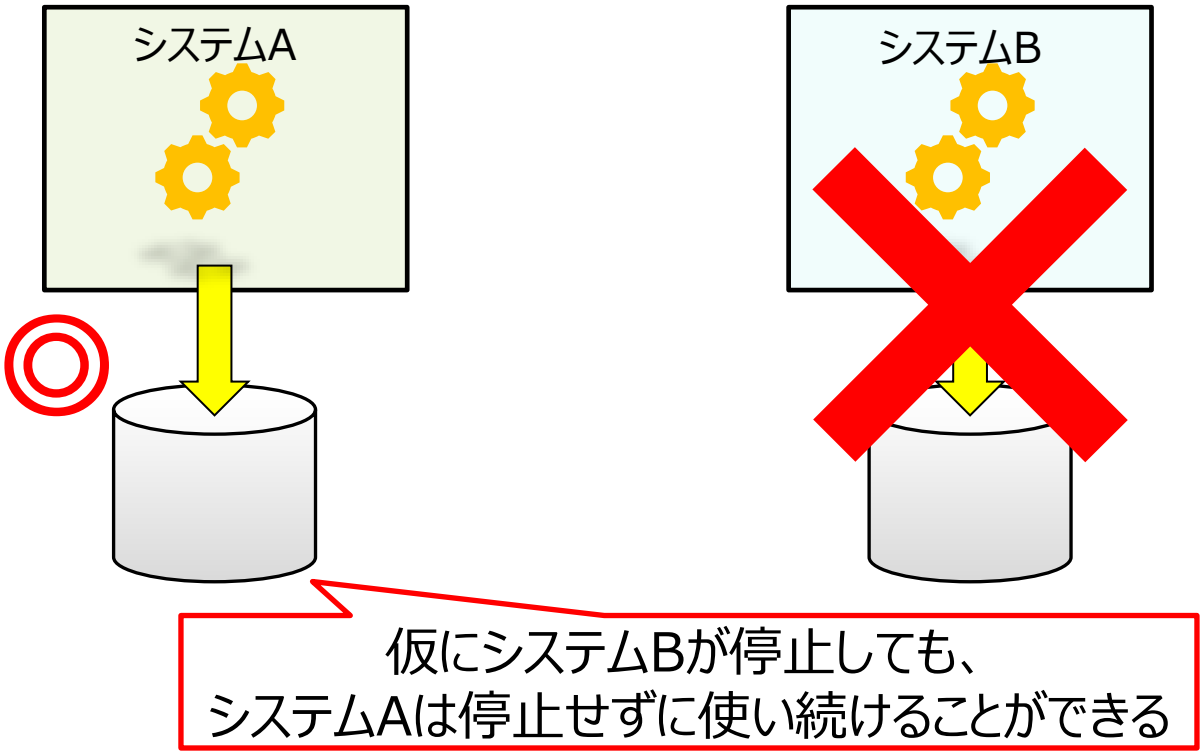
各システムが、同じ値を参照して処理できること。



CAP定理(4/6)

Availability
可用性

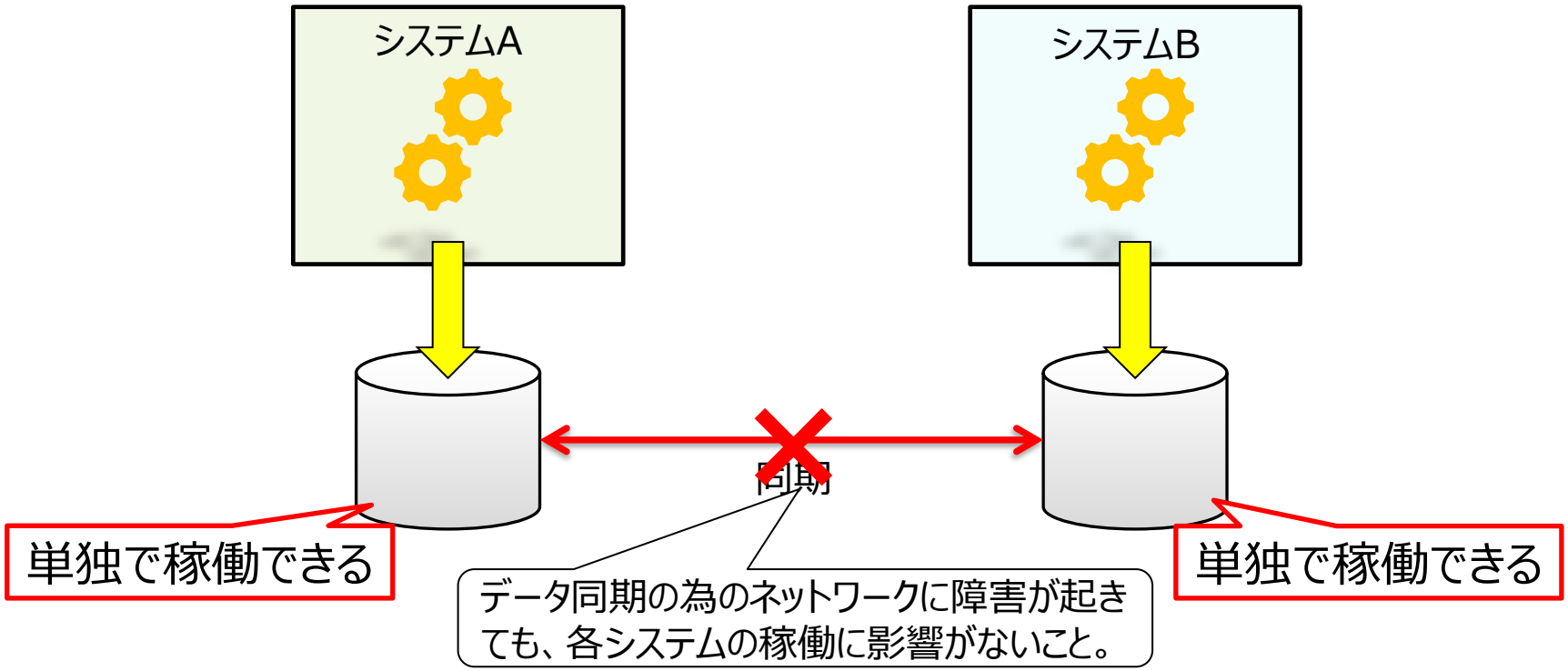
各システムが他システムの影響を受けず、稼働し続けられること。



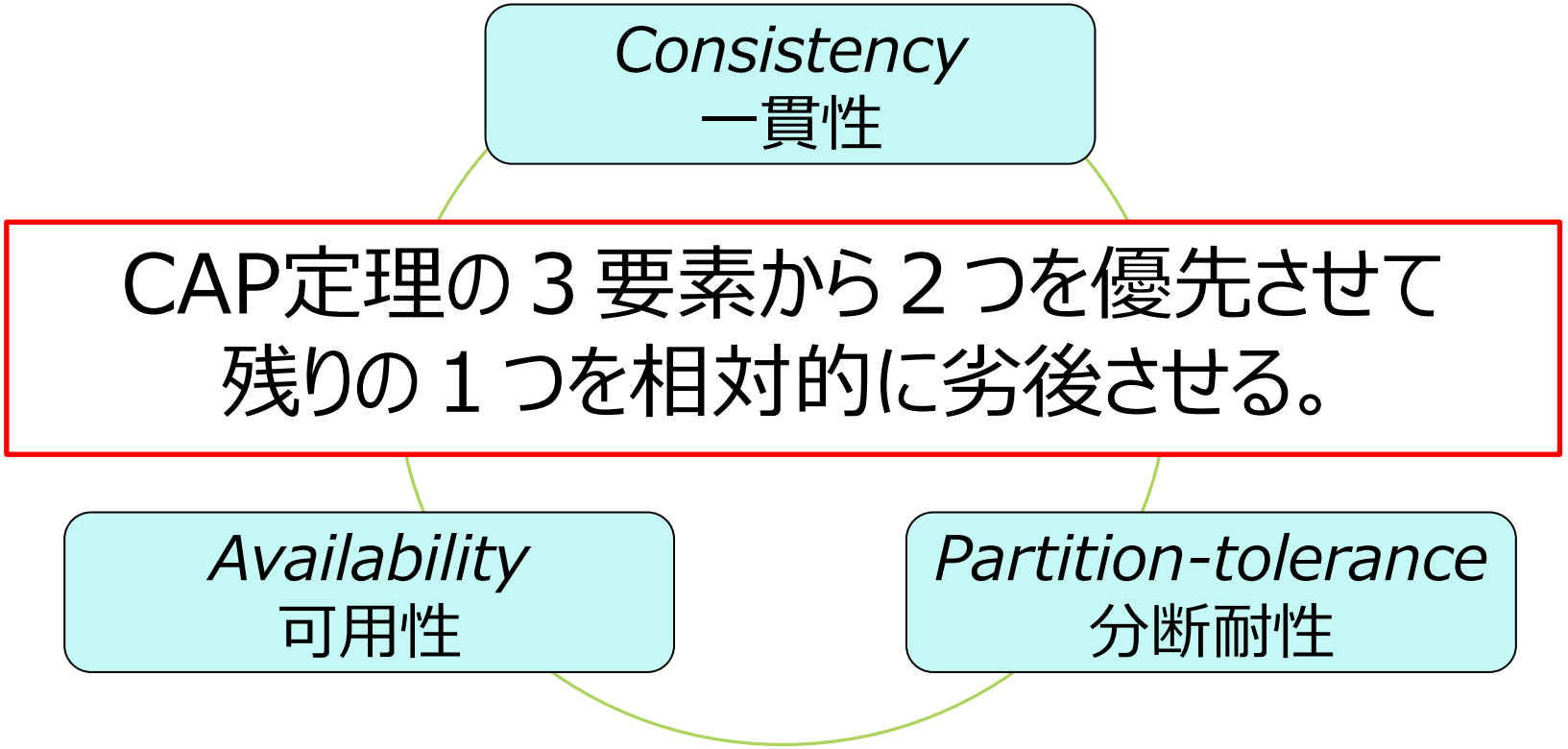
CAP定理(5/6)

Partition-tolerance
分断耐性

ネットワーク分断が起きても各システムの稼働が止まらないこと。



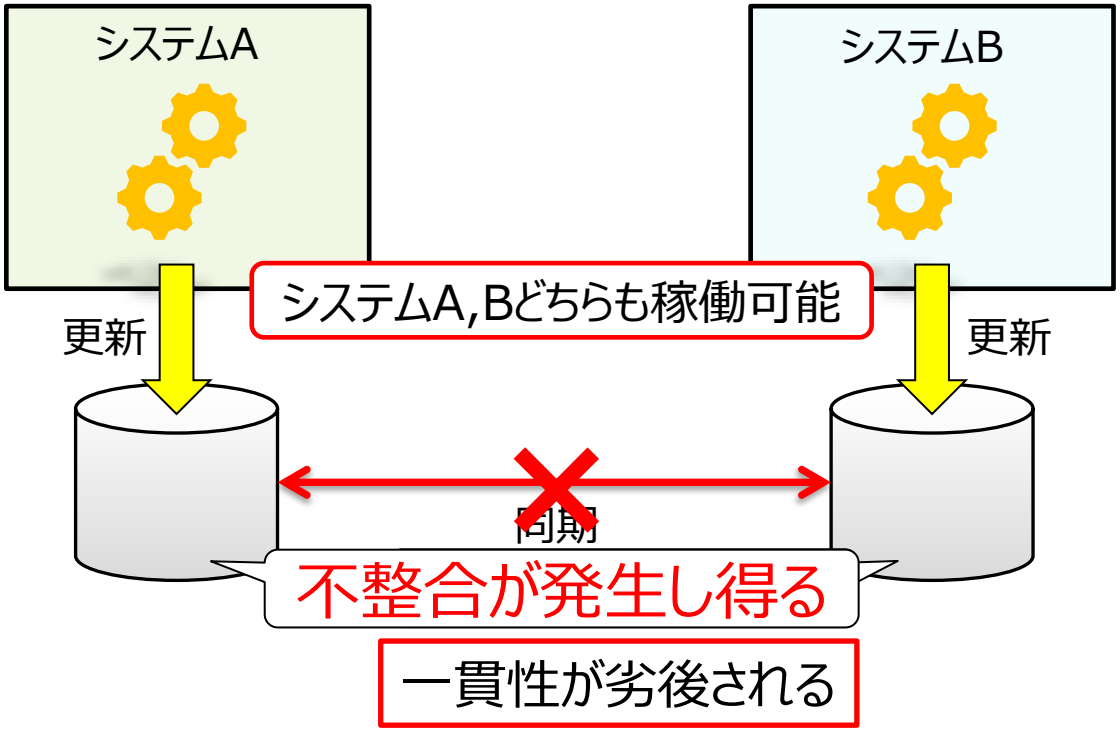
CAP定理(6/6)



- 相対的に劣後させる要素は何か、それがどのように劣後するのかの整理や検討をする際に利用する。
- 一貫性、可用性、分断耐性は全て同時に保持することはできない、ということを、理論的に説明できる。

分断耐性＋可用性を優先

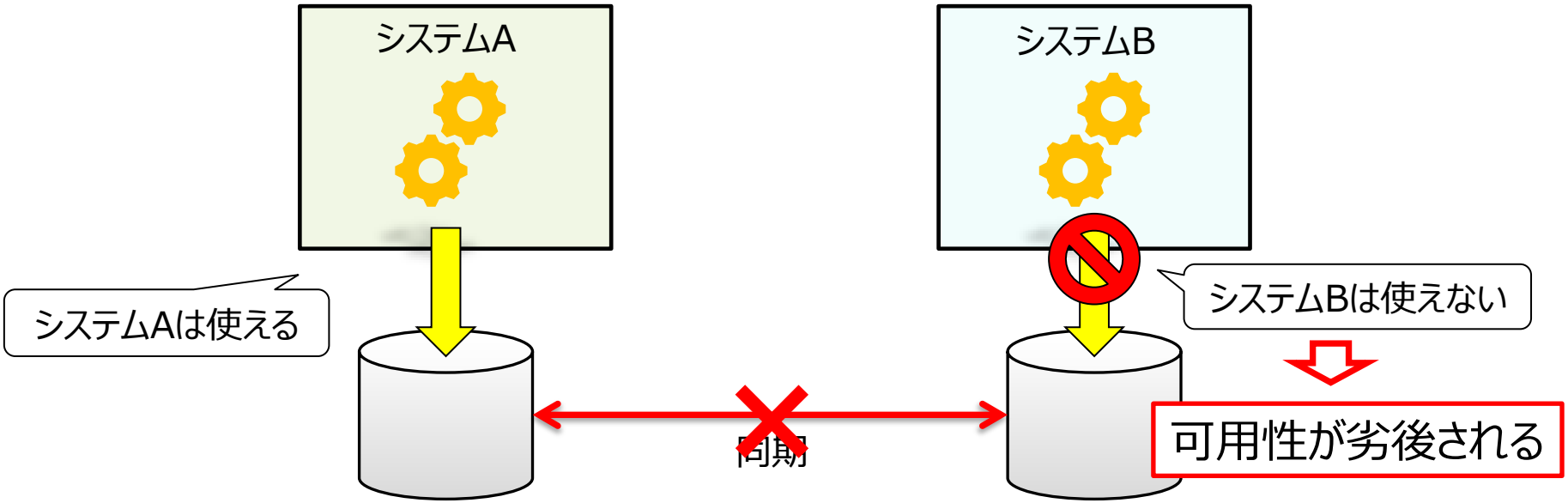
ネットワーク分断時でも各システムが稼働可能だが、データの不整合が生じる（一貫性が劣後する）可能性がある。



分断耐性＋一貫性を優先 (1/2)

実現手法 1

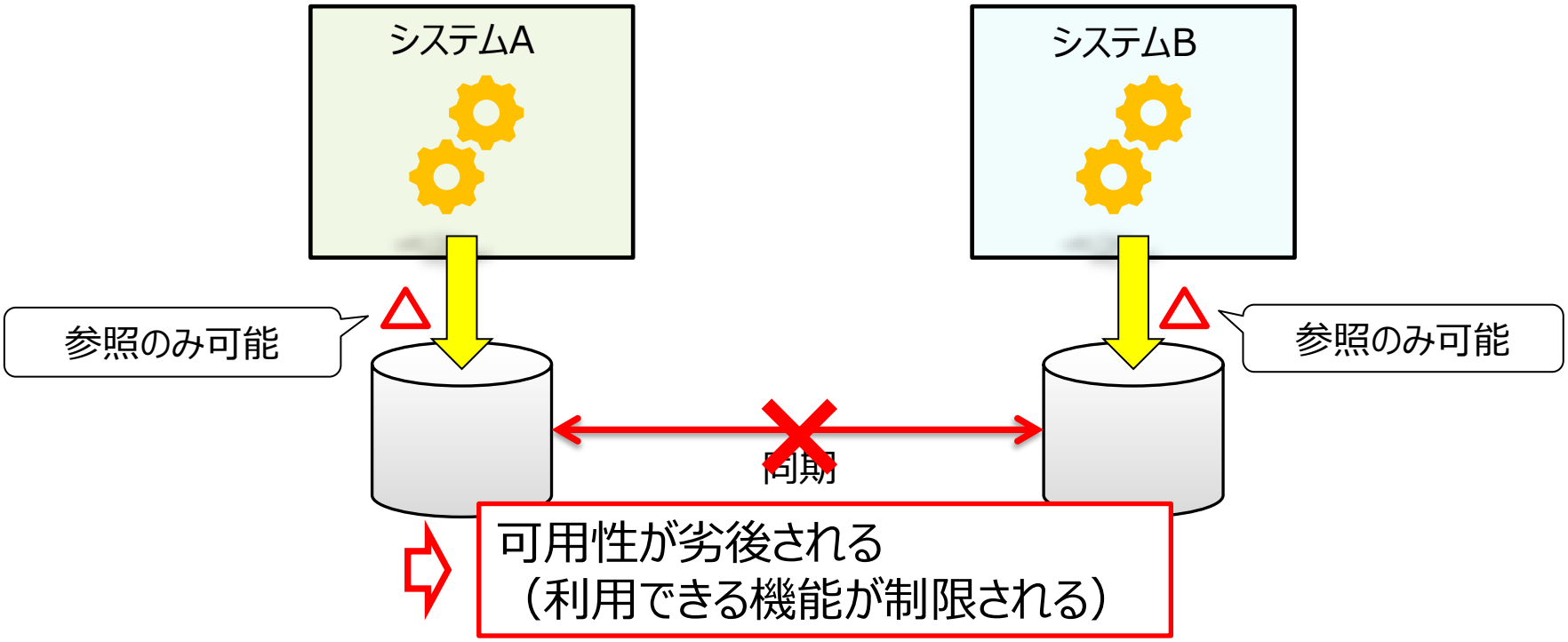
ネットワーク分断時、システムBを停止することにより
データの一貫性を保証する。



分断耐性＋一貫性を優先 (2/2)

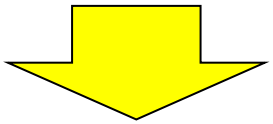
実現手法２

ネットワーク分断時、DB更新を伴う処理のみを
停止することにより、データの一貫性を保証する。



可用性＋一貫性を優先(1/2)

可用性と一貫性のトレードオフが生じる原因は、
データが分散しているから。

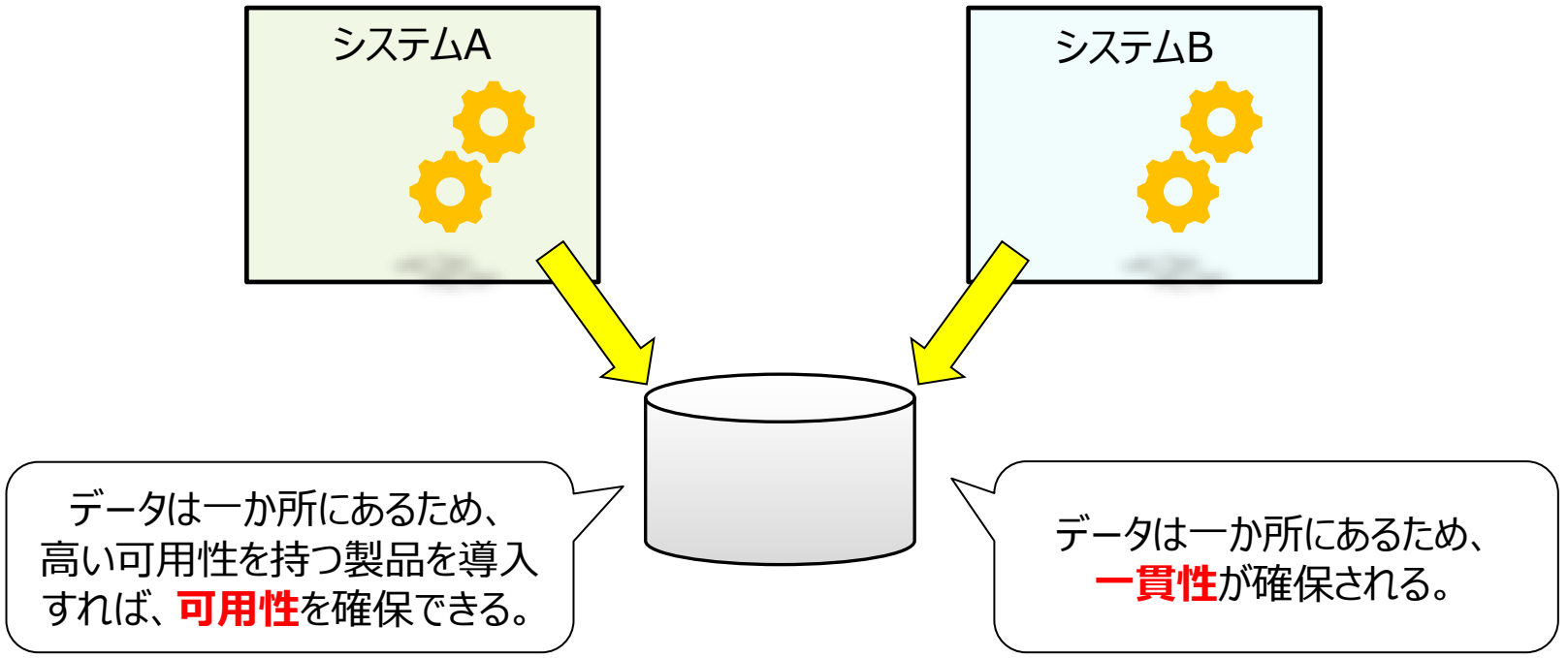


可用性と一貫性を追求するならば、データの分散を
あきらめる必要がある。

データは各システムごとに持たせたいけれど、可用性も一貫性も追求したい、という要求は実現できない。

可用性＋一貫性を優先(2/2)

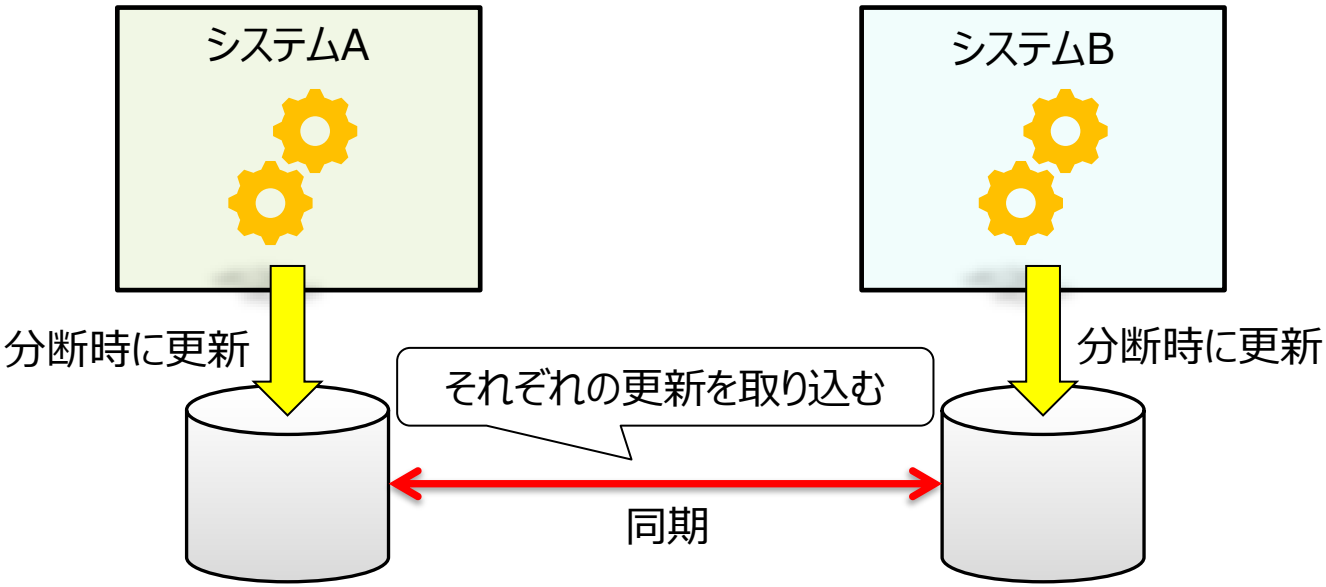
データを保存するシステムに高い可用性を持つ製品を導入し、可用性を確保する。



あくまでもトレードオフ(1/2)

例）分断耐性＋可用性を優先した場合…

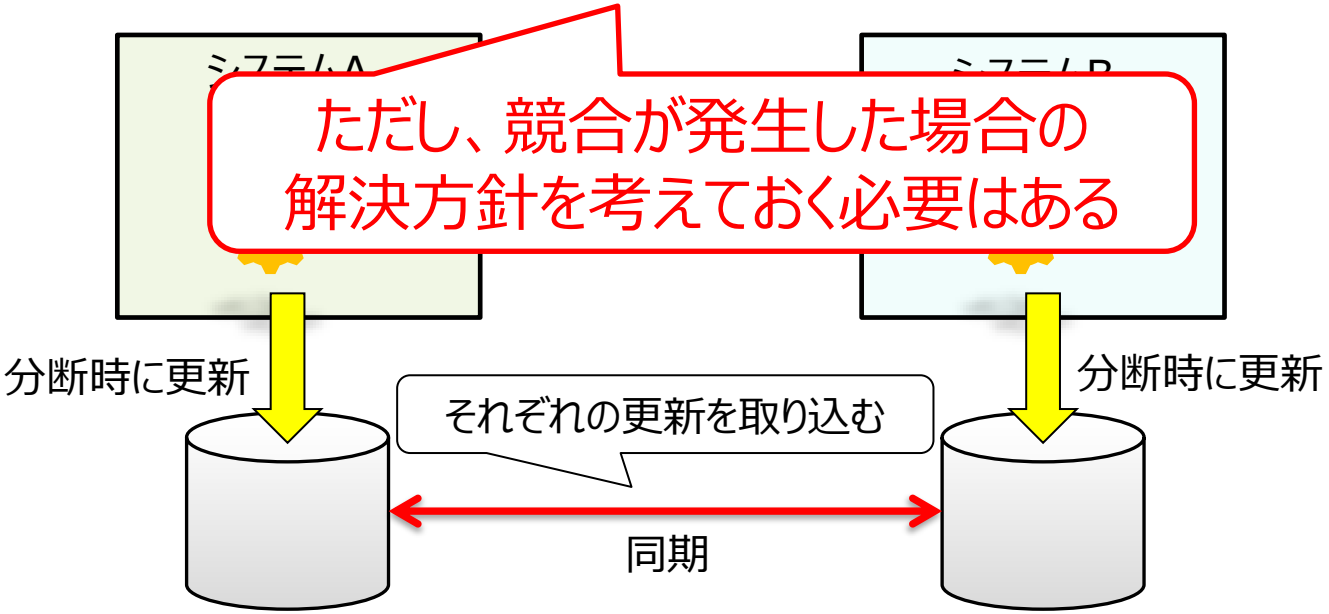
- 分断時は各システムでの更新が共有されないが、ネットワーク回復後にそれぞれの更新を取り込むことで結果的に一貫性は維持される（一貫性をまったく捨ててしまうわけではない）。
  これを、タイミングは後になるが、結果的に整合性は取られるという意味で、「結果整合性」という。



あくまでもトレードオフ(2/2)

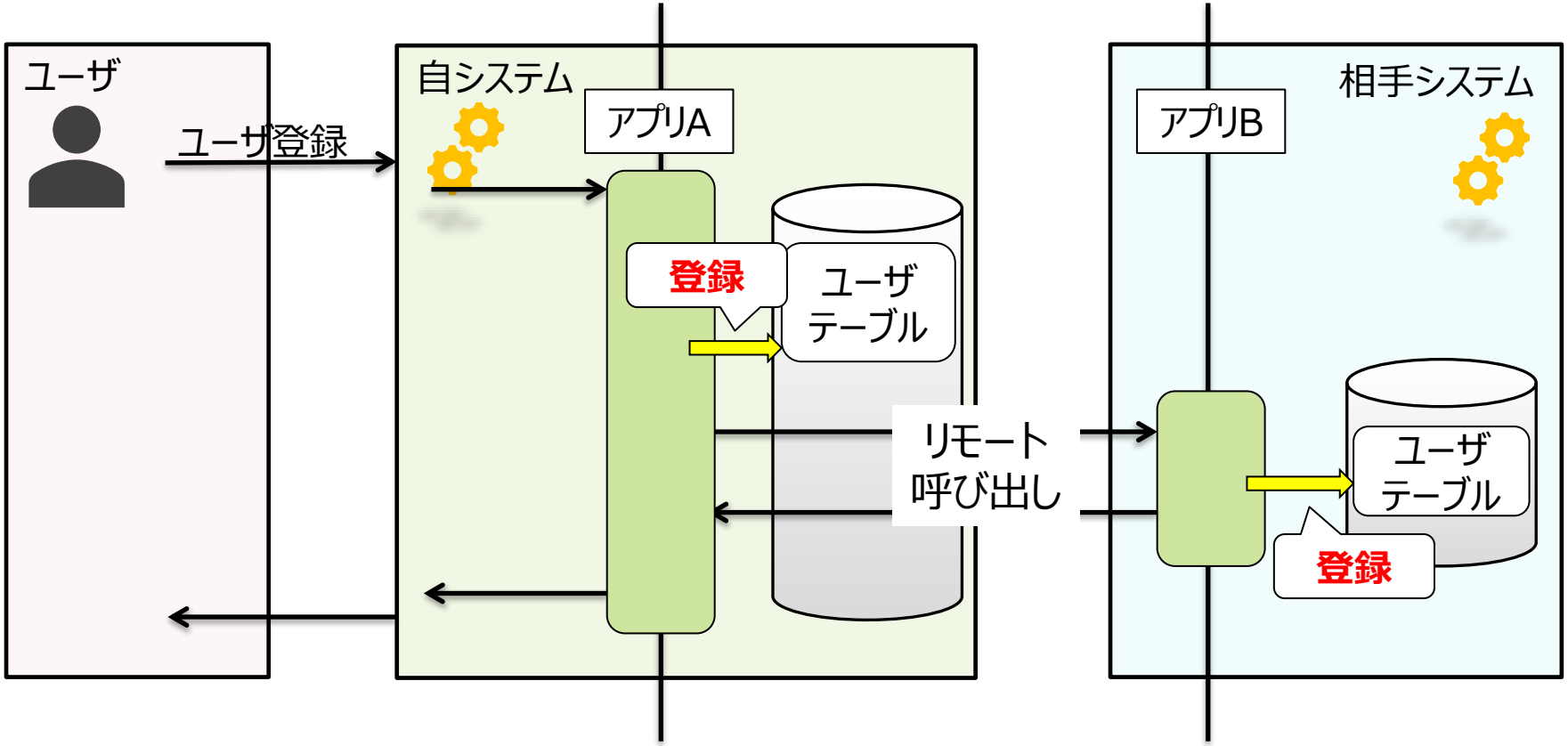
例）分断耐性＋可用性を優先した場合…

- 分断時は各システムでの更新が共有されないが、ネットワーク回復後にそれぞれの更新を取り込むことで結果的に一貫性は維持される（一貫性をまったく捨ててしまうわけではない）。
  これを、タイミングは後になるが、結果的に整合性は取られるという意味で、「結果整合性」という。



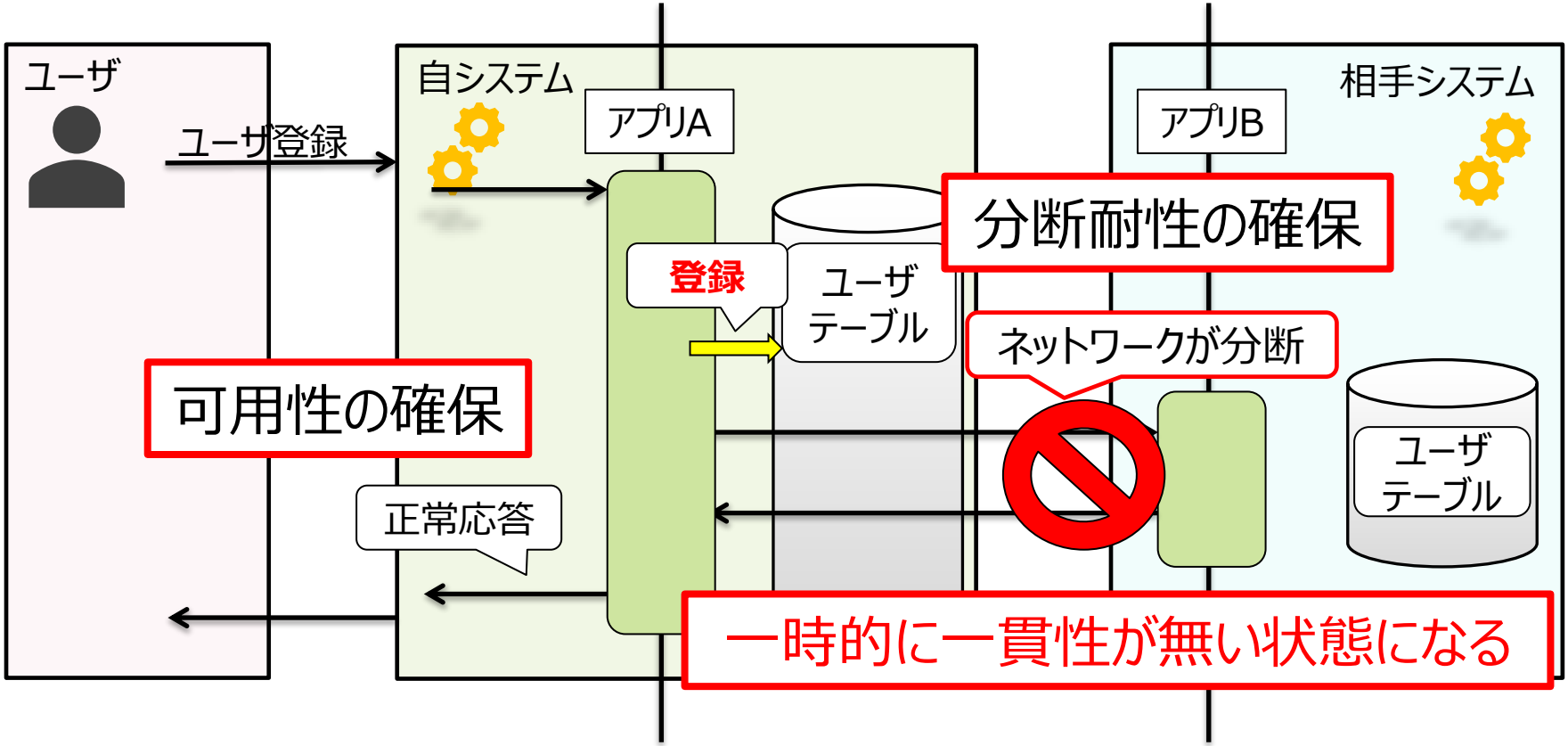
事例 – 分断耐性＋可用性(1/3)

ユーザ情報を自システムと相手システムで同期するシステムで、分断耐性を可用性を優先した事例。



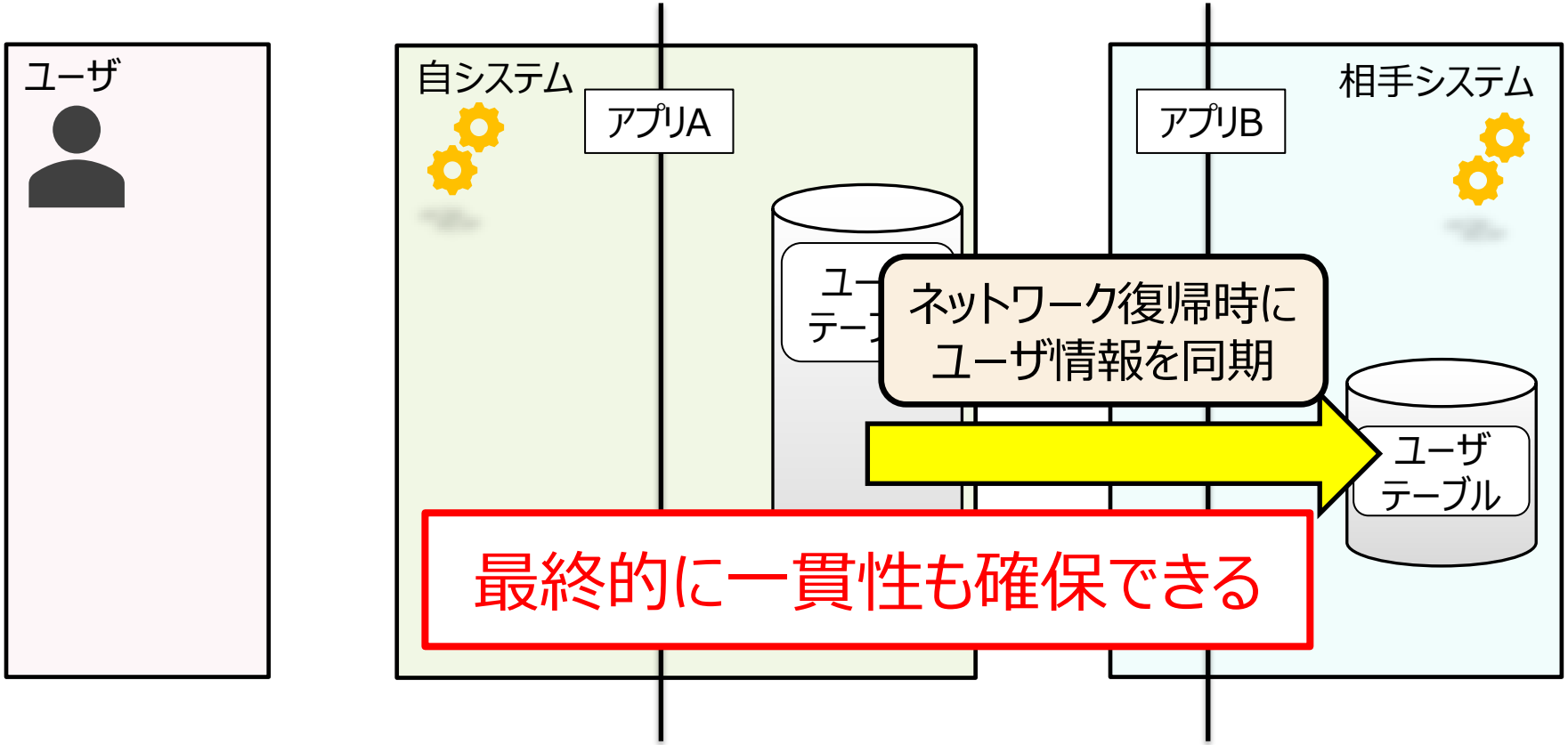
事例 – 分断耐性＋可用性(2/3)

通信エラーで同期できなくても、ユーザに正常応答を返す（分断耐性と可用性の確保）。



事例 – 分断耐性＋可用性(3/3)

ネットワーク復帰時にユーザ情報の同期を行うことで、一貫性を維持する（結果整合性）。



本講座で説明したこと

- システム間連携とは
- 方式のパターン
  - データベース共有
  - ファイル連携
  - リモート呼び出し
  - キューによる連携
- システム間連携の理論（CAP定理）

ITで、社会の願い叶えよう。

TIS INTEC
Group

付録

RESTに関する参考書籍

まずはこちらから

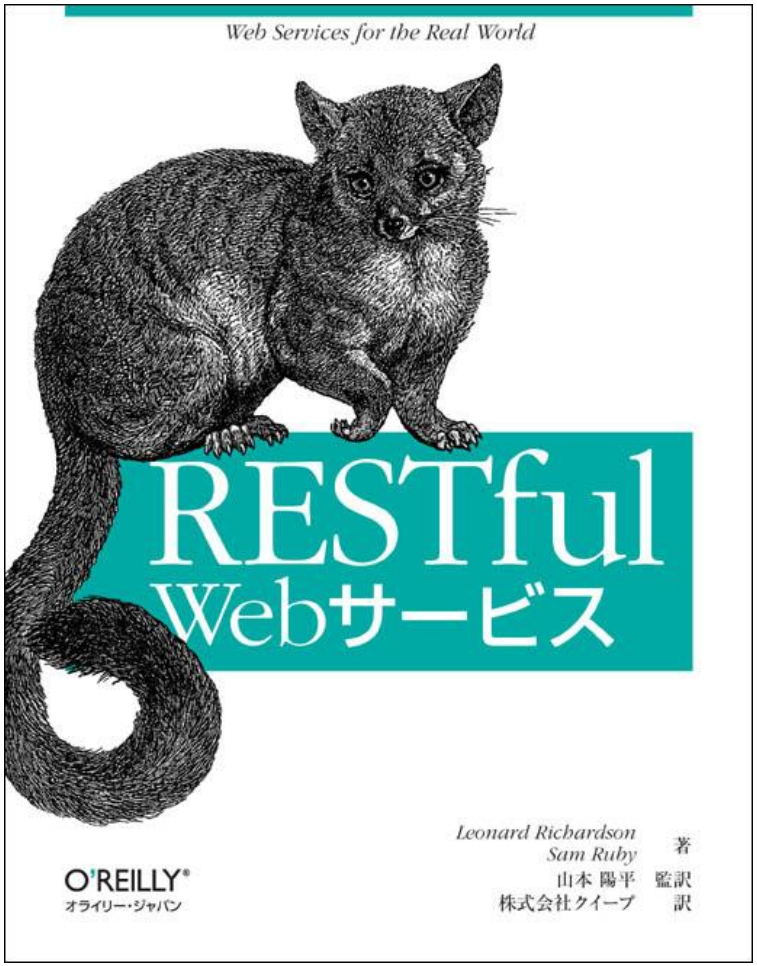
Webを支える技術
　　技術評論社
　　山本陽平　著
　　ISBN 978-4-7741-4204-3

https://gihyo.jp/book/2010/978-4-7741-4204-3
2021年11月25日17時の最新情報を取得

RESTに関する参考書籍

POSTとPUTの使い分けなど、より詳細な内容はこちら

RESTful Webサービス
オライリー・ジャパン
Leonard Richardson, Sam Ruby 著
山本 陽平 監訳
株式会社クイープ 訳
ISBN978-4-87311-353-1



https://www.oreilly.co.jp/books/9784873113531
2021年11月25日17時の最新情報を取得