修士論文

高信頼性とスケーラビリティを備えた
分散システムアーキテクチャに関する研究

1410302　藤澤宏明

主指導教員　鈴木　正人
審査委員主査　鈴木　正人
審査委員　青木　利晃
　　緒方　和博

北陸先端科学技術大学院大学
情報科学研究科

平成 29 年 8 月

概要

本論文の目的は、日々重要性の高まる非同期処理を行うシステムにおいて、より高い信頼性とスケーラビリティを兼ね備えたアーキテクチャを用いたサービスを実現するために、現行のシステムにおいて、キュー用 DB サーバとして RDB(Relational Data Base)型のデータベースを用いたアーキテクチャに対し、KVS(Key-Value Store)型のデータベースを用いたアーキテクチャの有効性を示すことである。

RDB 型は一つのデータベースを全てのアプリケーションサーバが共通して利用している。対して KVS 型は各アプリケーションサーバが、それぞれ別々に、各アプリケーションサーバと同一の HW 上に存在するデータベースを利用するアーキテクチャである。

RDB 型のデータベースを用いている分散アプリケーションにおいて、特に非同期でキューイング処理を行っている場合、以下のような問題が指摘されている。

- キュー用 DB サーバが SPOF(Single Point of Failure)であり、停止してしまうとシステム全体の停止に繋がる。
- 大量データの送受信時にキュー用 DB サーバがボトルネックとなり、システム全体の性能上限となってしまい、アプリケーションサーバを追加しても性能が向上しにくい。
- 特に中小規模のシステムにおいて、業務用 DB サーバと同一環境にキュー用 DB サーバが構築されることが多く、障害発生時やシステムの高負荷時に相互に影響が出てしまう。

KVS 型のアーキテクチャが RDB 型のアーキテクチャよりも有効であることを示すため、関連する技術として分散システム（複数のコンピュータが協調して動作を行い、複数個所で別々にデータを取り扱い、処理を行うシステム）の特徴と有効性を見極めるため、アーキテクチャについての調査を行った。調査では中央集約型システム（複数のコンピュータが協調した動作を行わず 1 か所で集中的にデータを取り扱い、処理を行うシステム）との対比を行い、分散システムの優位性を検討している。特に重要度の高いデータベースに着目して調査を行っている。

次に株価アラート配信システムを対象として、RDB 型と KVS 型の両方を実装し、実験を行うことで、提案するアーキテクチャによって改善した効果を確認している。

株価アラート配信システムとは、データの提供元（株価アラート生成システム）から受信するデータをユーザに配信するシステムである。このシステムは、株式市場に変動があった場合にユーザに取引を判断するためのデータを提供することを目的とし、事前に登録された情報を基に、株価アラート配信システムが各ユーザ別に異なるデータを作成する。重要な機能として、株価の変動にあわせてデータが連続して発生する場合、連続してデータを受信し、受信順にデータの配信を行う。性能要求として、20 秒以内にデータを作成し配信を行う必要がある。

このシステムをモデル化し、実装したアプリケーションを用いて、(1)性能、(2)障害発生時、(2)性能劣化について測定を行いアーキテクチャの有効性を確認している。

(1)正常時の性能：TPS, CPU 使用率

(2)故障発生時の状態：エラー検知、復旧方法

(3)データベース停止時の性能：TPS, CPU 使用率

送信データ数(配信対象人数)を 10, 50, 100, 150, 200 と変化させており、アプリケーションサーバ台数は 1 台～4 台に対して、TPS 及び CPU を測定している。
この結果として、以下の点で KVS 型の有効性を確認した

(1)KVS 型の方が全体的に RDB 型よりも TPS が高い数値となっており、データベースサーバの負荷が低減されていること。

(2)アプリケーションサーバの増加に伴い、RDB 型よりも KVS 型の方が線型に性能が向上しており、スケーラビリティが高いこと。

(3)障害検知後及びシステムの復旧後に異常停止や不必要なエラーが発生し続けることなく、使用可能であること。

(4)一部のデータベースが停止した状態で不必要にコンピュータリソースを使用することなく、システムを使用可能であること。

ただし測定結果において、一部予想と異なる点があったが、特に問題がないことを確認している。
さらに追加実験を行うことで、このアーキテクチャを利用する上での留意点を 3 種類に分類できた。

- 発生しなかった問題(障害発生後のメモリリーク、データベース停止時の性能劣化)
- 発生するが極小的な影響(データベース障害発生時の重複したエラー検知、メモリ使用量の増加)
- 発生するが頻度が低く、システムの特性上、問題とならない現象(負荷が低い場合、RDB 型の方が KVS 型よりも高い性能となる)

この他に、アプリケーションサーバに関しては RDB 型よりも KVS 型の方が高性能なサーバが必要となる。これはシステムの CPU ボトルネックがデータベースサーバからアプリケーションサーバに移動しているためである。

結果として、KVSを使用した新アーキテクチャは、従来のRDBを用いたアーキテクチャに対して、特にスケーラビリティに優れており、障害発生時にも問題なくシステムを継続して使用する事が可能であることが分かった。1 台のデータベースが停止しても大きく性能が落ちることなく、十分に実用可能であると考えられる。
ただし、従来のアーキテクチャとは異なる仕組みとなるため、システムのボトルネック、キュー用 DB が停止した時の処理負荷の見積もり及び DB サーバ高負荷時の運用を考慮してシステムの設計、及び運用を行う必要が出てくる。

以上の結果を踏まえ、別システムにKVS型アーキテクチャを適用することを考えた場合、キューDB及びアプリケーションサーバは株価アラート配信システムと同じ仕組みを利用可能である。
ただし、キュー用データの初期化及び可用性実現のための冗長構成については株価アラート配信システムと異なる仕組みが必要となる。

キューDB を用いる理由の一つとして、更新処理の発生量及び発生頻度が高いシステムにおいて有効なためである。逆に、更新量が少ないまたは頻度が少ない、更新量が多くても同一データに対する更新競合が少なくデータを参照する処理の方が多いシステムではキューDB は特に必要とならない。
このようなキューDB を用いることなく、KVS を適用可能なシステム例を下記に記載する。

- ソーシャルコミュニケーションシステム（チャット、Blog 等）
- 運用管理ツール（ログ収集、ユーザ/操作履歴管理等）

データモデルを工夫し、KVS に適したデータ構造とすることで利用しやすくする可能性はあるものの、KVS が適さないと考えられるシステム例を以下である。

- DWH：Data Ware House（店舗別売上データ分析、年間発注データ分析等）
- マスタデータ関連システム（BOM、MD 等）
- 業務支援システム（需要予測、自動発注等）

RDB が持つデータ処理関数を効果的に活用可能であり、データ操作時には様々な解析軸が必要となるため、KVS を活用する利点がない。

データベースに依存度が低いシステムの場合、データの送受信はあるものの、データベースの機能を活用した仕組みとなっていない可能性が高い。このため、RDB ではなく KVS を用いることができる可能性も高くなると考えられ、KVS を有効に活用できる可能性がある。

今後の課題として以下がある。

- データベース接続に関する耐障害性への影響調査
  プライマリまたはセカンダリとなるデータベースを明確に設定して接続先を固定して使用できるよう、JDBC ドライバの調査を行う必要がある。
- 障害復旧時の操作に関する調査
  データベースプロセスのみを再起動して復旧できることが望ましい。OS からの再起動を行う場合は影響範囲を考慮した運用が必要となる。
- システムの可用性
  実際の可用性を考えるならば、システムの全てのアプリケーションの実行や運用で用いる機能が全て実行可能であることを網羅的に確認する必要がある。
- アプリケーション機能と実装の違いが性能に与える影響
  システムからデータを配信する端末数及びユーザが事前に登録する配信データ種類の設定において影響が出てくると考えられる。

目次

図表一覧

第 1 章　はじめに

分散 DB を含む並行システムでは、利用するユーザの業務オペレーションと非同期に行われるデータの送受信技術の重要性は日々高まっている。特にデータの処理量や提供するサービスが年々増加しており、高い信頼性とスケーラビリティを実現するアーキテクチャが求められている。

現行の非同期処理を実現する仕組みは、データを処理するアプリケーションサーバをスケールさせることでデータ量に応じた分散処理を行っている。一方、非同期処理の要求に対する応答をユーザには基本的に返さない。つまり要求を受け付けた後は、システム内で障害が発生した場合でもその要求を失うことなく確実な実行を実現する必要がある。この機能を実現するために、現行のシステムでは非同期処理の要求をキューイングし、RDB(Relational Data Base)を用いてデータの一貫性や処理の正当性を保証している（キュー用 DB サーバ）。

この RDB を用いたキュー用 DB サーバは、全てのアプリケーションサーバで共通して同一の RDB を使用するように構築されることが多い。アプリケーションサーバは複数のサーバを冗長構成としてシステムを構築することで、一部のアプリケーションサーバに故障が発生してもユーザにサービスを提供し続けることができる。しかしキュー用 DB サーバは 1 か所でのみ稼働しており、もし停止してしまった場合は全てのアプリケーションサーバが使用できない状態となり、システム全体の停止につながる SPOF(単一障害点。Single Point of Failure)となっている。加えて、特にユーザ数の増加やデータ量の増大に伴い、大量に非同期で実行するアプリケーションの要求が発生する場合、各アプリケーションサーバからのキュー用 DB サーバに対する処理量も増加する。このためキュー用 DB サーバに負荷が集中してしまい、処理が遅延する状態が発生する。キュー用 DB サーバの処理が遅延してしまうと、システム全体が過剰に遅延する原因となってしまい、スループットが低下し、ユーザに満足のいく処理応答時間を提供できなくなってしまう。このときアプリケーションサーバを追加することでシステム全体の処理性能を向上させることもできず、システム全体の処理性能の上限がキュー用 DB サーバの性能上限となってしまうため、より高いスケーラビリティを実現することができない状態となってしまう。

この他に、特に中小規模のシステムにおける業務アプリケーションで使用する RDB とキュー用 DB との相互影響が問題となる。これは中小規模のシステム開発において、キュー用 DB サーバの構築及び保守に関する作業負担が大きくなってしまうため、構築や保守を簡易に行うことを目的とし、業務アプリケーションで使用する RDB と同一のハードウェア上にキュー用 DB サーバを構築することが多いためである。結果として障害発生時等には論理的な独立性が失われてシステム全体が停止してしまう問題が発生する。また、ユーザがシステムを利用するピークの時間帯では、相互に性能に影響があるためシステム全体のスループットが低下してしまう、という問題も発生する。

本論文の目的は RDB に代わり KVS(Key-Value Store)型のデータベースを用いることで、上記の問題解決を試みる点に独自性がある。具体的にはキュー用 DB サーバを SPOF とならないようにシステムを構築し、本来実行すべき業務処理との相互影響をなくし、システムの処理要求数に応じたスケーラビリティを実現するアーキテクチャでシステムを構築し運用することである。結果として、ユーザはサービスを常に同じように享受することができ、開発者は性能要件（非機能要件）に応じた柔軟なシステムを構築するこ

とが可能になる。この目的を達成するため、Key-Value 型の DB を用いたアーキテクチャ(KVS 型)を実際に利用することが可能であり、RDB をキューDB として用いたアーキテクチャ(RDB 型)よりも有効であることを示すことが本論文の目的となる。
RDB 型も KVS 型も、非同期処理を行うための要求をキューイングし、処理に必要なデータを永続化するためにデータベースを利用している。RDB 型は一つのデータベースを全てのアプリケーションサーバが共通して利用する。対して、KVS 型は各アプリケーションサーバが、それぞれ別々に、各アプリケーションサーバと同一の HW 上に存在するデータベースを利用するアーキテクチャとすることで、以下の点が有効になると予想される。

- 一部のデータベースに障害が発生しても影響範囲が限定的あり、システム全体が停止することがなく、ユーザは継続してシステムを利用可能。
- システムのボトルネックが発生しにくく、アプリケーションのサーバ数に比例してシステム全体の処理性能が向上する。

本論文では「分散システム」と「中央集約システム」を下記のように定義している。

分散システム

- 複数のコンピュータが協調して動作を行う。
- 複数のコンピュータが別々にデータを取り扱うことが可能。
- 各コンピュータは論理的にも物理的にも異なる構成が可能。

中央集約型システム

- 複数のコンピュータが協調した動作を行わない。
- 1 か所でデータを取り扱い、集中して処理を行う。
- 各コンピュータを論理的にも物理的にも異なる構成とすることが困難。

この定義に基づくと、RDB 型は中央集約型システムであり、対して KVS 型は分散システムとして捉えることができる。上記の有効性は中央集約型システムに対する分散システムの有効性と考えることができる。
本論文の構成として、まず 2 章では分散システムの一般的な特徴について、中央集約システムと比較した有効性の調査を行っている。特にデータベースに着目して調査を行い、RDB と KVS の比較を行うことで KVS の有効性、及び複数の KVS について検討を行っている。
3 章では、本論文で提案している KVS を用いた新しいアーキテクチャのモデル化を行い、具体的に対象としているシステムに対して改善の見込めるポイントを明確にしている。さらに実際の有効性を確認するため実験を行っており、4 章において下記の内容についてまとめている。

- TPS の測定による処理性能の比較(RDB 型と KVS 型の比較)
- 意図的に障害を発生させた際のシステムの動作(障害検知後の動作)
- 一部のデータベースが停止している状態での性能(性能劣化測定)

5 章では実験の結果に対しての分析、6 章では実験で対象としたアプリケーションへの適用の他に、キュ

ーDB を持つ別システム、キューDB を持たないシステム、データベースへの依存度の低いシステムの利用に関する検討を行っている。最後に 7 章で全体を統括し、今後に向けた課題についても言及している。

第 2 章　関連技術調査

本章では、本論文で対象としている分散システムのアーキテクチャについて、新しい提案に繋げるため、一般的な特徴について調査を行った。特に提案の内容に密接に関連しているデータベースに着目し、調査した内容を記載する。

2. 1　分散システム

中央集約型システムに対する分散システムの優位性として下記があげられる。

- 機能
  - システム構成
- 非機能（耐障害性）
  - 無停止でのサービス提供が可能
- 非機能（性能劣化測定）
  - 障害発生時のユーザへの影響が限定的
  - 故障発生時のデータ復旧が容易
- 非機能（性能、スケーラビリティ）
  - 要求数の増加に対し、必要なスループットを提供可能

それぞれの優位性について、下記に記載する。

機能：システム構成

異なるサービスレベル（ユーザの利用時間帯、障害発生後の復旧までの時間、性能要求とコストとのトレードオフ等）の機能を提供したいと考えた場合、1か所で集中的に処理を行う中央集約型システムよりも、複数のコンピュータが協調して動作する分散システムの方が柔軟な構成でシステムを構築することができると考えられる（同時並行性が高いため）。中央集約型システムではシステム全体を最高レベルのサービスを実現するための構成に揃える必要性が出てくる可能性が高いが、分散システムでは、提供する機能やユーザが利用するサービスに応じて別々の構成にすることで対応することも可能である。また性能要件として求められる応答時間が提供する機能ごとに異なる場合、中央集約型システムではシステム全体の性能を考慮して一括してコンピュータリソースを用意し、常にシステム全体の構成を視野に作業を行う必要が出てくる。対して分散システムでは、提供する機能やユーザが利用するサービスごとに、別々に必要なコンピュータリソースを用意してシステムを構成することが容易であり、必要な機能のために必要なだけサーバを追加したり、システムを構成する一部に対して変更を行うことも容易に行うことができる。

非機能（耐障害性）：無停止でのサービス提供が可能

システムの一部（プラットフォーム及びアプリケーションが対象）をメンテナンスする場合、特に OS 再起動や HW の電源再投入が必要な場合、中央集約型システムでは 1 か所で集中的に処理を行って

いるため、一時的にシステムで提供している機能が停止する等、利用しているユーザに対する影響が大きくなると考えられる。分散システムではユーザに提供するサービスを停止することなく、24 時間 365 日、無停止でサービスを提供することが可能となるよう、(理想的な状態において)全てのコンピュータが停止しない限り、ユーザがシステムを利用し続けるシステムを構築することが可能と考えられる。同様にアプリケーションのバージョンアップを考えた場合、中央集約型システムではシステム全体を一度止める等、一括した変更作業が必要になると予想される。分散システムではサービスを利用中のユーザは継続してサービスを利用し続けつつ、新しくサービスの利用を始めたユーザから、新しいバージョンのアプリケーションを使用するといった変更作業にも柔軟に対応できると考えられる。

非機能(性能劣化測定):障害発生時のユーザへの影響が限定的

中央集約システムでは、1 か所でデータを取り扱うこともあり、システムを論理的に異なる構成の組み合わせとする前提となっておらず、例えば提供している機能やユーザが利用しているサービスごとにシステムを物理的に切り出すことが難しく、システムの一部で障害が発生した場合、システムを利用している全てのユーザに影響が出てしまう可能性が高くなると考えられる。

対して分散システムでは、部分的なシステム障害や NW 分断が発生しても障害の発生したサーバを利用している一部のユーザのみが影響を受けることになる(障害の発生していない、別のサーバを利用しているユーザは影響を受けない)。中央集約型システムと比較して、多くのサーバを用いてシステムを構築することになるため、初期構築や運用の難易度は上がる。しかし、透過的に機能を利用できる仕組みを実現することで、障害発生時に影響のあったユーザに対しても、同一のサービスを別サーバでも提供する仕組みを用意することで、システム全体を止めることなくサービスを提供することも可能となる。

非機能(性能劣化測定):故障発生時のデータ復旧が容易

中央集約型システムはデータの保持やバックアップも 1 か所で行うことになるため、データを含めた故障が発生した場合、データの復旧が難しい状態になると考えられる。回避策としては複数のデータバックアップを複数のデータセンターに配置するといった災害対策と同様の考え方が必要になってくる。

対して、分散システムでは正常にシステムが稼働している状態においてデータを分散して複数のコンピュータに配置して保持することが可能である。このため一部のコンピュータが故障したことに伴い、データの不整合や消失、アクセス不能な場合が発生したとしても、対象は一部のデータである。デフォルトで冗長的に別のコンピュータにデータが配置されていることから、別のコンピュータを利用してシステムを利用し続けることが可能となり、データを含むシステムの復旧を行うことも比較的に容易に実現できると考えられる。

非機能(性能、スケーラビリティ):要求数の増加に対し、必要なスループットを提供可能

中央集約型システムは 1 か所で処理を行うため、システムに対する要求数が増加しコンピュータリソースが不足する場合、必要なコンピュータリソースを柔軟に追加することができない。スケールアウトではなくスケールアップによる性能向上を図る必要が出てくる。複数種類の機能を提供する場合は、

最も処理負荷の大きい機能や最も処理要求数の多い時間帯に求められる応答時間を実現するためのサイジングを行い初期構築することになる。しかしシステム稼働後に予想以上の負荷となった場合やスパイクピーク時(相対的に短い時間幅で集中的に処理が行われる現象であり、システムが一時的に高負荷となるものの、その他の時間帯は緩やかな使用量)にあわせたサーバリソースの追加といった柔軟性に欠けている。一か所のコンピュータの性能限界がシステム全体の性能限界となり、ボトルネックが発生しやすいアーキテクチャである。

対して分散システムではスケールアウトによる性能向上を図ることが容易である。要求数が増加した場合にはサーバを追加することで容易にコンピュータリソースを追加することができる。特に理想的な状態においては追加したサーバ数に比例してシステム全体の処理性能も増加することになり、一部のコンピュータがボトルネックにならないような設計が可能である。例えば、システム稼働後に業務の繁忙期やスパイクピークの発生するタイミングにあわせて、一時的にサーバ台数を増加して処理を行い、通常の業務時間帯やユーザの利用頻度が少なく要求数の少ない期間はサーバ数を削減するといったスケーラビリティの高いシステムを構築が可能である。結果として、ユーザは常にストレスなくサービスを利用することができ、サービスの提供者側も不要なコストを割くことなくシステムを設計し、サービスを提供することが可能になる。

近年はネットワークが発展し、相対的に安価で高性能なコンピュータが普及してきていることを背景に、複数のコンピュータを組み合わせることで、分散して処理を行うシステムが発展してきた。分散して処理を行う場合、各コンピュータはネットワークを介して接続し、それぞれが処理を行うことになるが、接続の仕方は様々である。いくつかの例を図 2-1 に記載する。

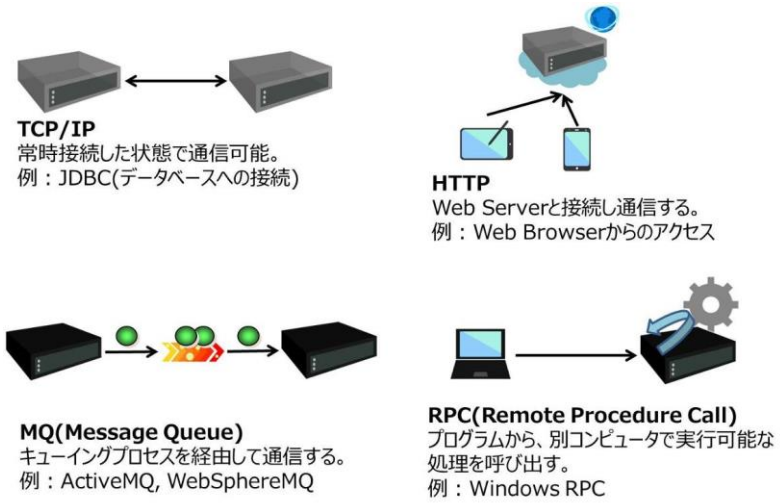


図 2-1: システムを構成する要素の接続例

TCP/IP

データベースに対して Java アプリケーションからアクセスする際に利用される JDBC(Java Database Connectivity)をはじめとし、高度なアプリケーション要求を可能にする接続であり、確実な電文送信を可能にする信頼性の高いプロトコル実装である。下記のような目的がある場合に用いられる。

- アプリケーション間で常に接続している状態を保つことで高速な応答を返す。
- 接続先システムで障害が発生した場合の障害検知、及び動的な接続先の変更。
- 接続時または処理要求に対する応答に対しタイムアウトを設定。
- 送受信される電文内のデータを直接アプリケーションで操作する。

HTTP

一般的にWeb系とも呼ばれる、Webブラウザ（例：IE, Edge, Chrome, FireFox等）を使用してデータの送受信を行うシステムに代表される。ユーザは使用している端末のWebブラウザを用いてデータの参照、または操作を行い、実際のデータはユーザの使用しているコンピュータ上ではなく、離れた別のサーバに存在している。この他にファイル形式あるいは電文として、コンピュータ間でデータの送受信をする場合に用いられる。基本的に接続している相互のコンピュータ間で同期を取って処理が行われ、動作しているアプリケーションに依存したデータフォーマットでデータの送受信が行われる。ここにREST/SOAPも分類される。

MQ（message queuing）

特に異なるコンピュータ上で、それぞれ異なるアプリケーションが動作しているシステムにおいて、アプリケーション間で必要なデータの送受信に用いられる。接続しているコンピュータは、それぞれ別のサービスレベル（それぞれ異なるシステム稼働時間、実行ユーザ、要求数、応答時間、提供サービス等）でシステムを構築することができる。データの送受信は、各アプリケーション間で同期をとることなく非同期で行われ、それぞれのアプリケーションの状態に応じてデータの送信、データの取得が行われる。

RPC

RPCとは、アドレス空間の異なるプログラム間で、requestとresponseを組み合わせた処理を行う仕組みである。代表例として、OLE, Exchange等のアプリケーションで利用されている。処理を実行する側では各アプリケーションを実行するために、それぞれ固有のインターフェースを用意している。処理を呼出す側は、このインターフェースを経由して処理要求を行う。呼出し側では処理結果をキャッシュして保持することで、同じ処理結果に対しては繰り返しアクセス可能にし、必要なデータ操作を行う場合にのみ通信を行うといった仕組みとすることも可能である。

また、アプリケーションの特性にあわせたアーキテクチャを選択し、様々なサービスが実現されている。下記に分散システムのアーキテクチャの代表的な例を記載する。

- Client-Server model
- Peer to Peer
- Grid computing
- Three-tier architecture

Client-Server model

インターネットを経由した動画配信サービスを例としてあげられる。サーバはデータを保持し、圧縮したデータを配信するのみであり、データを受信したクライアント側コンピュータは、サーバから動画データを受信して出力（デコード）する。リアルタイムなデータの配信だけでなく、ユーザが好みの時間に視聴する、繰り返し何度も視聴するといったタイムシフト（録画、録音）でのデータ配信も可能となっている。同様にクライアント側で画像データを受信した後は画像化を行い、音声データを受信した後は音声化するといったデータの送受信を行うことで、サーバ側の負荷を軽減している。

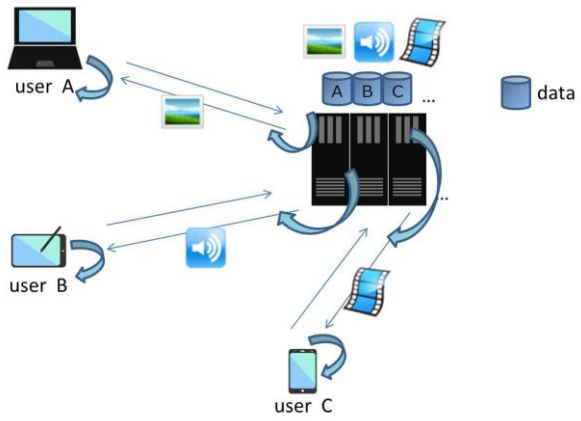


図 2-2: Client-Server model

Peer to Peer

少容量のデータを多数のコンピュータ間で通信する場合に最適な形態と考えられる。ファイル共有等に用いられる。特定のサーバだけがデータを持つことにはならないアーキテクチャであり、全てのコンピュータがデータを受信するだけではなく、保持しているデータの配信も行う。ユーザ数が増加すると、システム全体のコンピュータリソースも増加する。各コンピュータは peer と呼ばれ、コンピュータ数が増加すればするほど相互のコンピュータ間でデータ送受信を活発に行うことが可能になる。

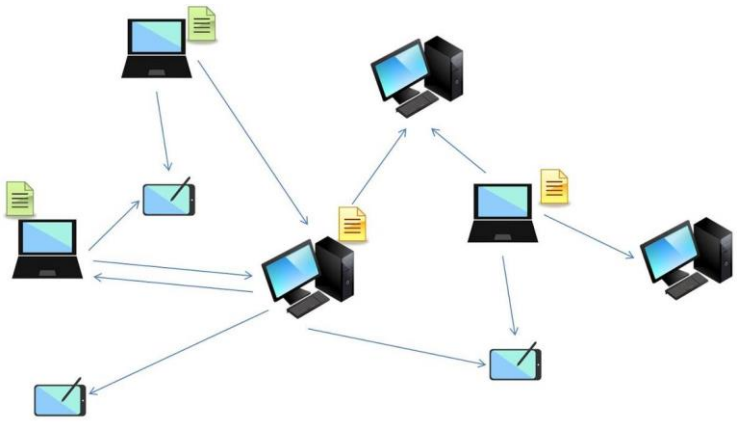


図 2-3: Peer to Peer

Grid computing

World Community Grid （OpenZika, Help Stop TB, FightAIDS@Home …etc）を例としてあげることができる。世界各地のユーザが有志でコンピュータリソースを提供し、それぞれのコンピュータが別々に計算処理を行う。協調した動作の必要がない、特定の計算処理に限定されており、計算用アプリケーションを各コンピュータにインストールすることで実行が可能となる。特定の地域に閉じることなく、全世界規模で計算処理が行われる。

図 2-4: Grid Computing

Three-tier architecture

システムのサーバ側を 3 層（UI, Application, Database）の仕組みに分割して構築するシステムである。各層毎にコンピュータリソースを増減することで、処理要求数や処理種類に応じたアプリケーションの配置やコンピュータリソースの適正化が可能になる。また、各層毎にメンテナンスを行う運用を行うシステムを構築することが可能となる。

さらに 3 層→N 層の仕組み（Multitier architecture）に分割することで、システムを構成する各要素の相互影響を極小化し、各要素間を疎結合な状態としたシステムを構築することも可能である。

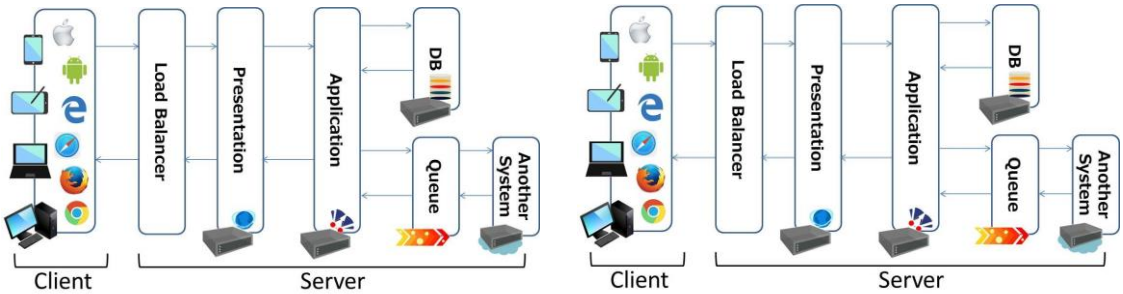


図 2-5: tier architecture
左:3 層　右:N 層

2. 2 分散データベース

企業向けシステムにおいて、データの重要度は非常に高いものであり、業務と密接に関わり合いが高いため、データが消失してしまうことは絶対に防止しなくてはならない。このためデータを冗長化し、分散して配置するアーキテクチャが必要になってくる。同時にシステムの停止は業務の停止または圧倒的な作業効率の低下となってしまう等、非常に影響が大きいことから、システムに障害が発生しても使用し続けられるよう、可用性も求められることになる。

一方、一般的にデータベースはシステムのボトルネックになりやすい。これは全ての業務用アプリケーションでデータの参照や更新を行うためである。例を下記に記載する。

- 商品の受発注データの作成および参照
- 請求処理、日次や月次での集計処理
- 物流データ(在庫、倉庫)との連携
- 売上データの管理、POS システムとの連携

業務システムは必ずデータベースにアクセスして処理を行い、日々データ量は増大していくことになる。また、特に様々な処理が競合した場合、データに不整合が起きないようデータを管理し操作する処理を実現するには開発コストが大きい。データの一貫性を保つためには、集中的に一つのデータベースで処理が行われることが多く、相対的にデータベースでの処理時間がシステム内で大きくなり、ボトルネックとなる傾向がみられる。このような背景から、システムに対する要求数やデータ量が増加した場合の性能向上も求められる。

これらの要求に対応するための、代表的な分散データベースの例と特徴を下記に記載する。

- DB scale 型:DB の機能によるデータ配置
  - Master - Slave
  - common storage
- HA(High Availability) 型:クラスタ用 SW を利用したデータ配置
- KVS(Key Value Store) 型:キーと値のペア型でのデータ配置
  - Oracle NoSQL
  - HBASE
  - Cassandra

DB scale 型(Master - Slave)

MySQL に代表される。このデータベースは Master 及び Slave の 2 種類のノードで構成され、データの更新は Master のみで行われる。Slave は更新の発生したデータを Master から取得することでデータの同期を取る。Master ノードが停止した場合、Slave の一つが Master となり、処理を継続する。DB にアクセスするアプリケーションが、どのノードに接続するかを決定する。データの更新を行う場合は Master に接続し、データの参照のみを行う場合は Slave へ接続するといったアプリケーションで行う処理特性に応じた接続先の選択を行う。

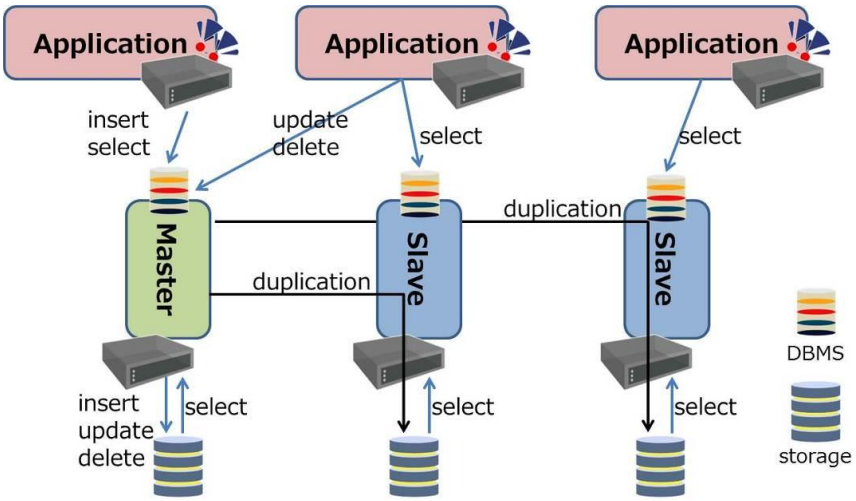


図 2-6: DB scale 型 (Master-Slave)

DB scale 型(common storage)

Oracle, DB2 に代表される。各ノードの役割は均等であり、メモリ上のデータはノード間で常に同期をとる。データの保存先となるストレージは全ノードで共通のため、いずれかのノードで障害が発生した場合、他のノードで継続して処理が可能となる。DB にアクセスするアプリケーションが、どのノードに接続するかを決定する。どのノードが管理するデータにアクセスするか、アプリケーションで取り扱うデータの種類によって選択を行う。

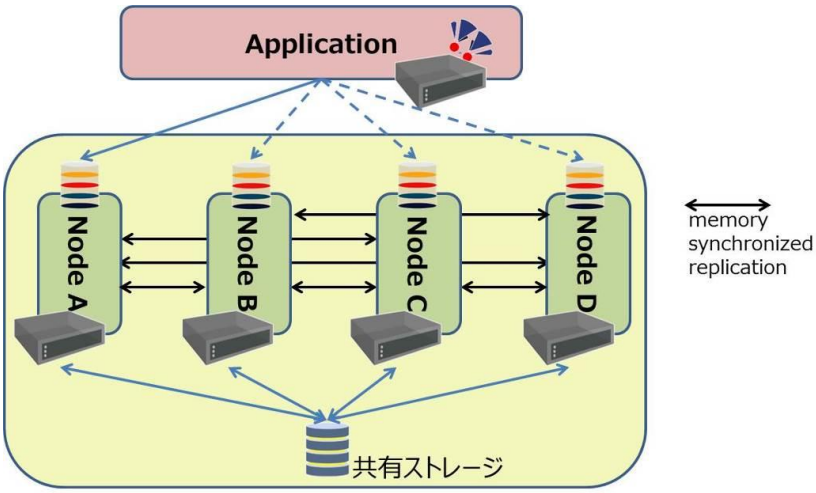


図 2-7: DB scale 型 (Common Storage)

HA(高可用性)型

データベースとは別に、クラスタリング用ソフトウェアを用いることでデータベースの冗長化を実現する。アプリケーションからの接続先は、クラスタリング用ソフトが提供する固定した一つの接続先となる。冗長化している各ノードは、ストレージを共有しており、クラスタリング用ソフトウェアが DBMS の死活監視を行うことで、障害発生時は復旧までの時間が短く切替が可能となる。必要最小限の冗長構成(2 台)

が一般的で、中小規模のシステムでの利用が多い。監視やメンテナンス及び構築コストが低く、性能やスケーラビリティが求められない場合に用いられる。

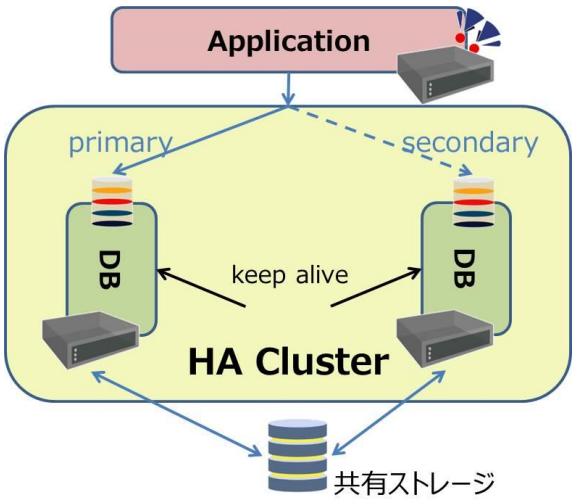


図 2-8: HA 型

以上、DB scale 型及び HA 型は RDB(Relational Data Base)を用いることが一般的である。次に KVS(Key Value Store)を用いた分散データベースについて記載する。

KVS 型 (Oracle NoSQL)

Master ノードと Replica ノードの 2 種類から構成され、全てのノードは同じデータを持つ。アプリケーションはいずれかのノードにアクセスして処理を行うが、更新処理は Master ノードでのみ行っている。参照処理は各ノードがアプリケーションに応答を返す動作となっている。これは複数のユーザが同一のデータに対して更新を行う場合、データの状態が異なる結果とならないよう、データの一貫性を優先するアーキテクチャとしているためと考えられる。このため、参照処理は負荷が分散されるものの、データの更新を行う場合は常に Master ノードのデータが先に更新され、後から Replica ノードのデータが更新されるため、Master と Replica 間でデータの不整合が発生する時間帯が存在する。

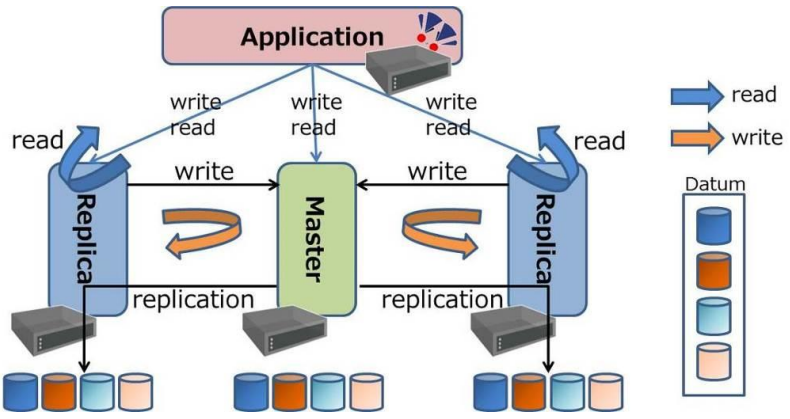


図 2-9: Oracle NoSQL

KVS 型 (HBase)

各ノード(Region)はデータの参照及び更新を同じように行うが、どのデータを取り扱うかはマスタによって割り当てられる。このためアプリケーションは、どのノードにアクセスして処理を行えばよいか先に確認してから実際の処理を行う(必ず Zookeeper にアクセスし対象のノードを把握した後、対象のノードに対して実際の処理を行う)。データの保存先は HDFS(Hadoop Distributed File System)により分割されて保存されており、HDFS を利用した他のアプリケーション(Hadoop, Pig, Hive 等)と連携しやすい。一方、システムの信頼性を高めるためには Zookeeper, Master それぞれを冗長化することも必要となってくる。

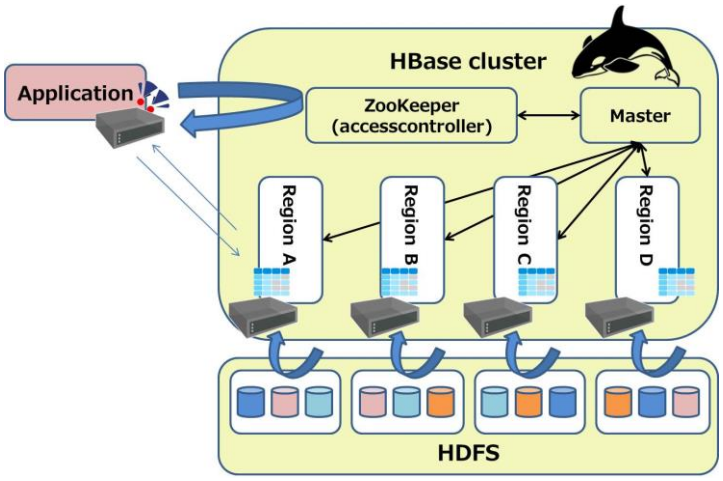


図 2-10: HBase

KVS 型 (Cassandra)

各ノードの役割は均等であり、アプリケーションはいずれかのノードにアクセスして処理を行う。データは各ノードに分散して配置される(データのキー情報をハッシュ化し、複数のノードに冗長的に分散される)。データの更新時は更新したデータの情報が非同期で伝播するため、各ノード間でデータの整合性が失われる時間帯が大きくなるものの、特定のボトルネックとなるポイントがなく、スケーラビリティが向上するアーキテクチャとなっている。

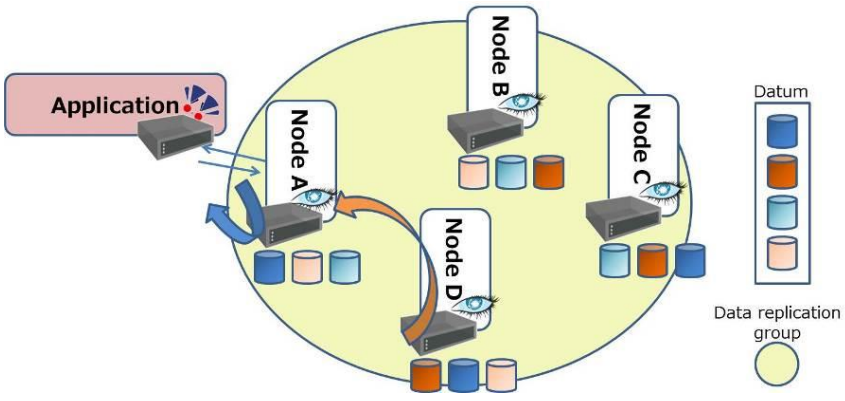


図 2-11: Cassandra

次に、RDB と KVS の比較を表 2-1 に記載する。RDB は中央集約型であり、障害発生時の影響が大きく、性能ボトルネックとなりやすいアーキテクチャとなっているのに対し、KVS は分散してデータを配置することで、信頼性を向上させ、要求数やデータ量に応じたスケーラビリティを実現可能なアーキテクチャとなっている。

表 2-1 RDB と KVS の比較

point	RDB	KVS
data format	テーブル形式 各値を個別に定義可能	キーとペアになる値 値は大きく一括りに取り扱われ、詳細定義は各KVSで異なる
data access	SQLを用いたデータ加工、集計、条件検索に優れる	キーアクセスが基本 データの格納に優れる
policy	データを一元管理	データを分散して管理

最後に KVS 内での比較を表 2-2 に記載する。Oracle NoSQL はデータ更新が中央集約型となっており、マスタノードのみでデータの更新が可能となっているためスケーラビリティの向上が難しく、HBase はデータ配置を管理するマスタ及びアプリケーションからのアクセスを振り分ける zookeeer が中央集約型となっており、SPOF となっている。対して Cassandra は、直接ノードにアクセスして処理を行うことが可能であり、SPOF がなく、スケーラビリティに優れている。他の KVS に対して、より効果的なアーキテクチャを実現できると考えられるため、Cassandra を採用して、KVS 型のアーキテクチャの開発を行った。

表 2-2 KVS 型の比較

point	Oracle No SQL	HBase	Cassandra
data access	各ノードに直接アクセス	マスタ管理 Zookeeper経由	各ノードに直接アクセス
data update	マスタノードのみ	各ノードで更新	各ノードで更新
storage	各ノードのローカル	共通HDFS	各ノードのローカル

第 3 章　アーキテクチャ

本章では対象としているシステムのアーキテクチャについての提案を行い、その特徴を把握し、実験で測定する点を明確にするためのモデル化を行う。

システムを利用してサービスを提供する場合の達成目標を下記にあげる。

- システムに障害が発生した場合、影響範囲を最小限にとどめ、ユーザがシステムを利用し続けることを可能にする構成
- 新しい機能の提供、システムの不具合を修正する場合、システムを停止してユーザがサービスを利用できなくなる時間の最小化
- システムに対する要求数の増加やシステムで扱うデータ量の増加に伴う、サービスを利用するユーザに対する応答時間の最小化

システムを構築する場合、機能要求や非機能要求に応えるため、上記を踏まえた上でシステムのアーキテクチャを決定する必要がある。
本論文では、下記をアーキテクチャの対象としている。

- サーバ構成
- ソフトウェア構成
- データの冗長化

システムを利用するユーザ側の処理や接続方式は含めておらず、ネットワーク環境や通信特性も対象外としている。
本論文で対象としているアーキテクチャについて、RDB をキュー用データベースとして使用しているシステム（RDB 型アーキテクチャ）と KVS をキュー用データベースとして使用する新しいアーキテクチャ（KVS 型アーキテクチャ）について、それぞれ記載する。

3. 1　RDB 型アーキテクチャの特徴と問題点

一般に非同期処理を実現するアーキテクチャでは、データを処理するアプリケーションサーバをスケールさせることでデータ量に応じた分散処理を行っている。このとき確実な実行を実現するため、非同期で実行するアプリケーション用に RDB を用いたキューイングを中央集約型で構築し(キュー用 DB サーバ)、データの一貫性や処理の正当性を保証している点が特徴としてあげられる。このようなアーキテクチャを本論文では RDB 型と呼ぶ。

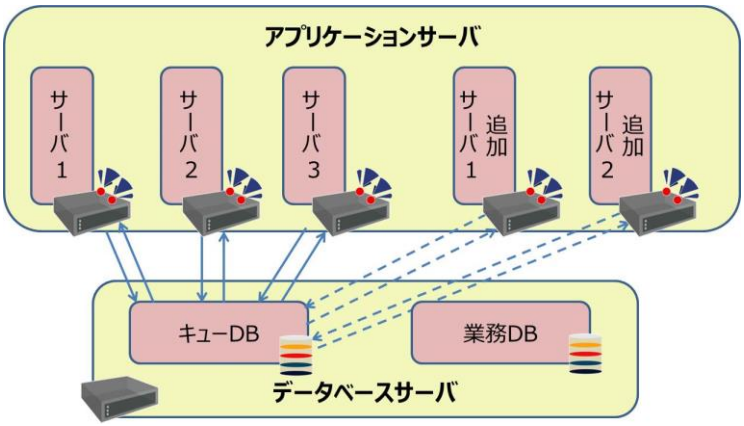


図 3-1: RDB 型アーキテクチャ

RDB 型アーキテクチャでは下記の点が問題点となっている。

1. キュー用 DB サーバが停止するとシステム全体が停止し、SPOF(単一障害点。Single Point of Failure)となる。
2. 特に大量のデータ送受信を行う場合、キュー用 DB への負荷が集中し、システム全体の性能が低下する。結果として、キュー用 DB サーバの性能がシステム性能の上限となり、アプリケーションサーバを追加しても性能が向上しない。
3. 同一のハードウェアを利用してキュー用 DB サーバを含むシステムを構築した場合、利用ピーク時の性能や、障害発生時に利用不能になる等の影響が双方に発生する。

3. 2 KVS 型アーキテクチャ

RDB 型における問題点を解決するため、キュー用 DB サーバとして KVS 型のデータベースを使用したアーキテクチャを提案する。RDB 型のアーキテクチャとは異なり、中央集約型とせず、アプリケーションサーバ上と同一環境に構築することで、分散してデータを利用する点が特徴となる。

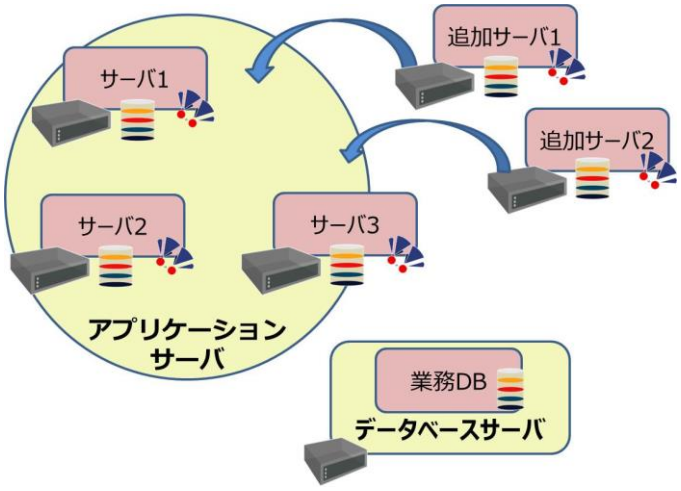


図 3-2: KVS 型アーキテクチャ

RDB 型の問題点に対して、それぞれ以下の点で改善が期待できる。

1. アプリケーションサーバと同じ粒度で分散して処理を行うため、SPOF が発生しない。
2. 特に大量のデータ送受信を行う場合でもキュー用 DB サーバへ局所的な負荷集中が発生することがなく、理想的にはアプリケーションサーバ数に比例して性能が向上する。
3. 業務アプリケーション用の RDB 同一ハードウェア上に構築されないため、性能や障害発生時に双方に影響が発生しない。

3. 3 実験対象システムの定義と構成

本論文では、株価アラート配信システムを対象として、RDB 型と KVS 型の両方を実装し、実験を行うことで、提案するアーキテクチャによって改善した効果を確認する。

株価アラート配信システムとは、データの提供元（株価アラート生成システム）から受信するデータをユーザに配信するシステムである。システムのユーザは株の取引を行っており、株式市場の動向を注視している。あらかじめユーザが取引対象としている株価の動向に対し、株式市場に変動があった場合に取引を判断するためのデータを提供することを目的としており、PUSH 型で各ユーザにそれぞれデータを配信するサービスを提供している。最終的にはモバイル端末にデータが届くが、モバイル端末にデータを届けるサービス（MDS: Message Delivery System）は別システムを利用しており、株価アラート配信システムの対象はユーザに届けられるメッセージを作成し MDS に受け渡すまでである。ユーザは事前に、どのような情報を配信してほしいか登録しており、事前に登録された情報を基に、株価アラート配信システムが各ユーザ別に異なるデータを作成する。

機能要求は下記に記載する。

- モバイル端末にデータを配信すること。
- 各ユーザは株価変動時に欲しいデータをあらかじめ登録し、市場における株価の上下価格閾値を超えた場合、株価配信アラートシステムがデータの提供元（REUTERS）からデータを受信し、対象のデータを登録していたユーザにメッセージを作成して配信する。
- 株価の変動にあわせてデータが連続して発生する場合、連続してデータを受信し、受信順にデータの配信を行う。

性能要求を下記に記載する。

- 最大同時生成メッセージ数：3 万件
- メッセージ配信端末数：5 万台
- 許容配信時間：20 秒以内（ユーザ端末までの配信ではなく、システムが配信対象のデータを受信し、ユーザに配信するメッセージを作成して送信処理を行うまでの時間。図 3-3 に示す①→⑤が対象）

この性能要件は、外部要因として突発的に処理負荷が高騰するイベントが発生した場合でも満たす必要がある。発生するタイミングとしては下記となる。

- 株式取引開始時刻（9:00）

- 昼休み明け取引開始時刻(12:30)
- ビッグニュース発生時(不定期)

いずれも平日、株式市場で取引が行われる時間帯が対象となる。

設計目標としては、一般的な指標として、上記のような処理負荷が高騰するイベントが発生した場合でも性能劣化(配信時間の遅延)を 30%程度に抑えることが求められる。また性能劣化が許容される時間幅としては 1 時間を超えないことが求められる。

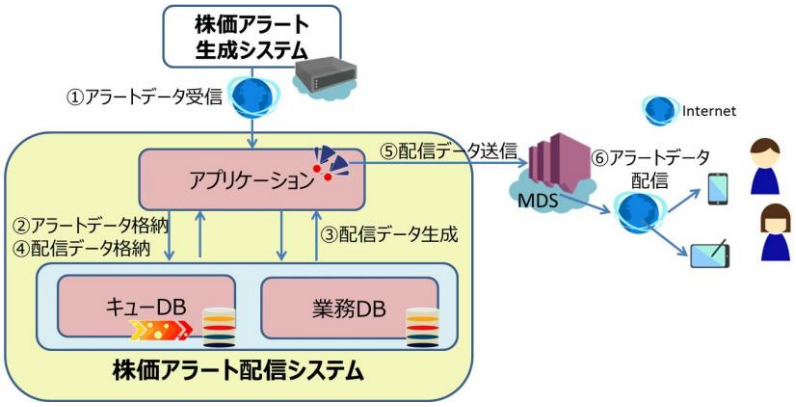


図 3-3: 株価アラート配信システム構成

図 3-3 において、データを配信する動作シーケンスは以下となる。

① システムが株価アラート生成システム(ロイター)からデータ受信
② システムがキュー用 DB サーバに受信データ格納
③ システムがユーザに配信するメッセージ作成
④ システムがキュー用 DB サーバに配信するメッセージを格納
⑤ システムが MDS(Mobile data Delivery Service)に送信
⑥ MDS が各ユーザにアラートデータを配信

システムに対する処理要求数が増加した場合、下記の数が増加する。

- ①の受信数
- ⑤の配信数

①の受信数は株価市場の変動が大きい場合、⑤の配信数は①の増加に加えて、システムを利用するユーザ数の増加、ユーザ 1 人当たりに配信するデータ数の増加するケースが考えられる。

キューイングしている送受しているデータ例を図 3-4 に記載する。

```
P G N T 5 0 20170316210248770.34.MSV_003.150.65.106.64.39548.1489665768768 jaist.FacJaist.dummy
Receive 150.65.106.64.39548.1489665768768 150.65.106.64 S106-064.JAIST.AC.JP 20170316 210248 N
0   -1 A A    180  N 0  0   jp.ac.jaist.bl.parameter.PrmDummyReceive flowId=20170316210248770.3
4.MSV_003.150.65.106.64.39548.1489665768768flow,q.tt=0,qput.time=1489665768770
<?xml version="1.0" encoding="UTF-8"?><p:data p:version="1.1"><root><request><alertMessage>"JAIST
ALERT MESSAGE"</alertMessage><alertNum>1</alertNum></request><result /></root></p:data>
```

図 3-4: キューイングしているデータ

表内のデータは視認性を高めるため、表内で改行を行っている。実際は 1 行一続きのデータである。処理要求一つにつき、上記のデータが一つキューに格納され、アプリケーションサーバで 1 トランザクションとして処理が行われる。

データの形式は、大きくヘッダー部とデータ部に分かれており、それぞれ役割が異なっている。

ヘッダー部（黄色部）

- 各値は white space（¥u001d）で区切られている。
- 全てのアプリケーションで共通して使用する。
- 各アプリケーションに共通して実施される処理で使用するシステム的な値。
- システム内で自動的にセットされる。

データ部（緑色部）

- XML 形式で表現される。
- 各アプリケーションで異なる。
- 最終的にユーザに届くデータの実体。
- アプリケーションのロジックに依存して作成される。

キュー用 DB に格納されるデータ及び、アプリケーションに対する処理要求は全て上記のデータ形式となっている。

このシステムにおいてデータの送受信の際にキューイングを用いている理由は下記となる。

- 受信したデータを確定し、データの提供元と疎結合な状態で株価アラート配信システムの稼働が可能。
- データの配信処理でエラーとなった場合、データの提供元から再受信する必要なく、配信処理のみ再実行が可能。

キューからデータを取得した後の処理は複数のスレッドが並列で処理を行うため、キューからデータを取り出す順番はデータの登録順になっているものの、データの取得後は先行して取得したデータの配信処理が遅延した場合、後から取得したデータの配信処理が追い越す可能性がある。要件として配信するメッセージの到着順序を保証する必要はないため、このシステム的に発生しうる現象は特に問題とならない。

3. 4 モデル化

システムとしてボトルネックになるポイント(処理時間の影響)を明確にし、スケーラビリティが向上する見込みがあることを検討するため、本論文で対象としているシステムの論理的なモデル化を行っている。モデル化に当たり、ユーザに配信する情報のステータス管理機能、ユーザが設定する複数の配信条件といった機能は対象外としている。モデル化は、株価配信アラートシステムがデータの提供元からデータを受信し、ユーザに配信するメインとなる機能を対象としている。このモデルに対し、システムの性能に対して、配信対象人数を変化することの影響と有効性が確認できるようにしている。このため、モデルでのシステムの動作は下記のようになる。

対象となる機能

- 事前に登録される画一的な情報を基に、各ユーザに対し別々にメッセージを配信する。
- 株価の変動に対し、登録していた閾値を超えた場合、通知を行う。

前提条件

- ユーザ毎に取引対象が異なるため配信するメッセージ内容は異なるが、それぞれのメッセージサイズには大差がなく、性能に影響がない。
- 最初に株価の変動監視対象となるデータを受信する。その後、配信用に各ユーザ毎にモバイル端末に届くことになるメッセージを作成する。
- メッセージの内容は作成後から配信されるまで、何らかの要因によって更新されたり、失われることはない。
- メッセージの内容は、システム内、及びシステム外からの影響を受けることなく、相互に干渉することもない。

このモデルを図 3-4 に示す。

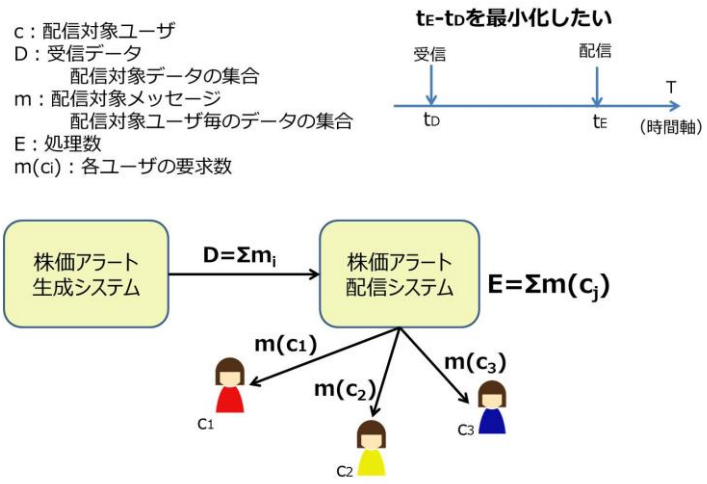


図 3-4: 株価アラートシステムで配信されるメッセージ数の関係

各値の説明は以下の通り。

- 配信対象ユーザを c，システムを利用しているユーザをそれぞれ ci
- 配信対象ユーザごとに作成するメッセージを m
- m の内容は c により異なるため、それぞれのメッセージを mi
- 株価アラートシステムが受信するデータの集合を D，D は m の集合となる。
- 各ユーザに配信するメッセージを m(c)，m(c)は m の集合となる。

株価アラートシステムの性能目標は、配信するメッセージの作成処理（m を配信対象人数ごとに作成する処理を m_{ci} とし、全メッセージの作成は $m(c) = \Sigma m_{ci}$ となる）の処理時間を最小化することである。株価アラートシステムからデータを受信する頻度の増加やデータベースに格納されるデータ量の増加の他、配信対象人数が増加しても、株価アラートシステム内での処理時間を最小にし、スループットが最大となることが望ましい。

特にキューイングしているデータを永続化するために使用しているデータベース（キューDB）に対する動作は、図 3-5 にモデルとして表現している。

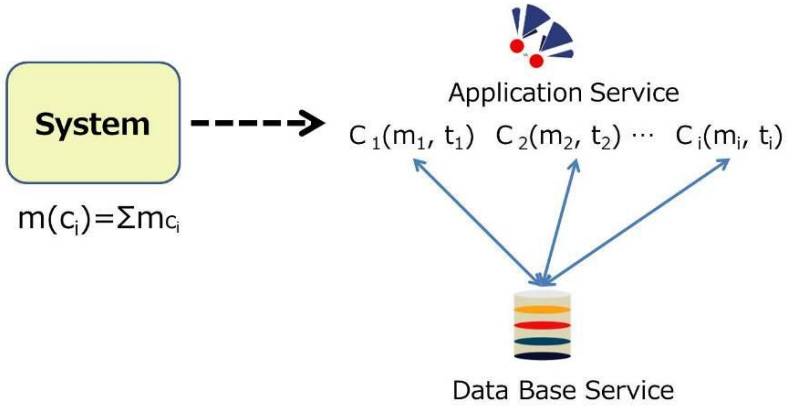


図 3-5: キューDB へのアクセスと処理時間の関係

ユーザ毎に配信対象となるメッセージを作成するための処理時間を t，C をアプリケーションサーバで行う処理とすると、C = C(m, t)となる。

アプリケーションから見た場合、データの永続化を実際に行うサービスが RDB か KVS かによって動作が異なることはなく、同じ処理が実行される。キューDB との応答時間（アプリケーションサーバからデータを渡し、キューDB から結果を取得するまでの時間）を小さくすることで、システムの処理性能を向上させることを目的としてアーキテクチャの有効性を考えると、キューDB を実現している要素を変更することで性能が向上する可能性がある。

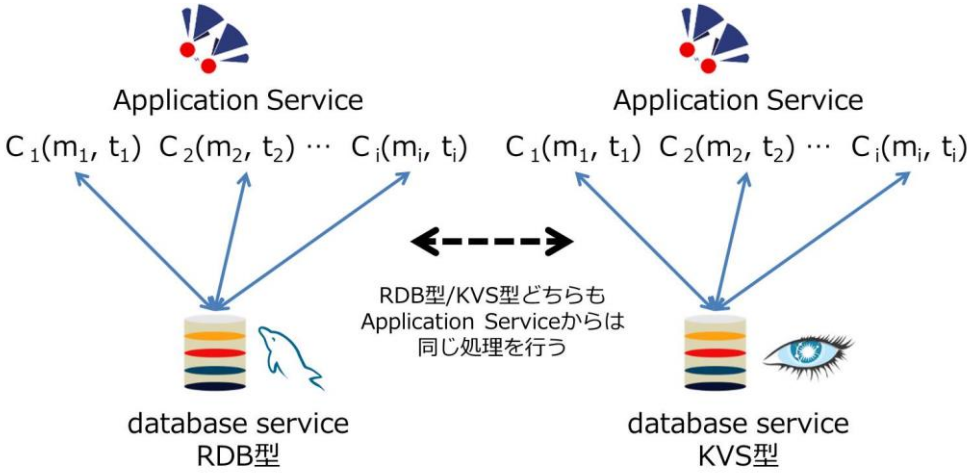


図 3-6: データストアサービス層における RDB 型と KVS 型における相違点

図 3-6 に示した通り、キューDB が RDB 型であっても KVS 型であっても、システムの動作は変わらない。

システムに対する要求数の変化に伴うシステムの状態遷移図は図 3-7 のようになる。

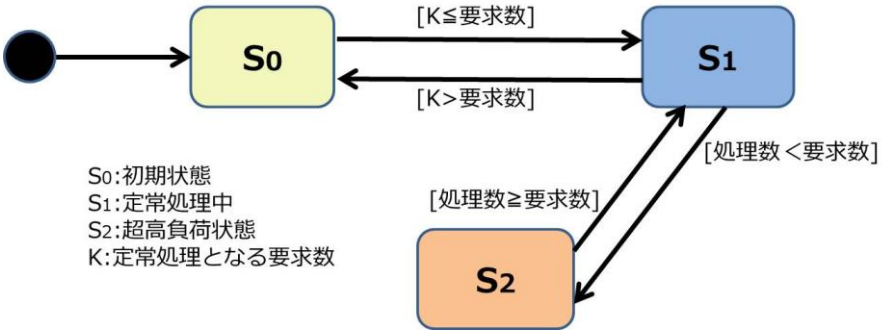


図 3-7: 処理負荷の変化による状態遷移図

システムの起動時は S0 の状態である。システムに対する要求数が一定の数に達するとシステムが定常的に処理を行う状態 S1 に遷移する。さらにシステムの処理性能を上回る要求数が発生した場合、処理を待つキューが大量に発生し、S2 の状態に遷移する。時間の経過とともに処理を待つキューが少なくなるため、S2 から S1 の状態に戻り、最終的にはシステムに対する処理要求数が 0 になるため、S1 から S0 に状態が戻ることになる。システム構築時には S2 の状態になることを避けることが目標となる。S2 の状態になると、ユーザに対するデータを配信するまでの時間が過剰に大きくなる、CPU の枯渇やメモリ不足といったシステム障害が発生する可能性が高くなる。

KVS 型も RDB 型も同じようにシステムの状態が遷移することになるが、RDB 型の持っているキューDB がボトルネックとなる問題が解消された場合、下記の効果が期待できる。

- KVS 型の方が S1 から S2 に遷移しにくくなる。
- S2 の状態が発生したとしても、KVS 型の方が S1 に戻るまでの時間が短くなる。

モデル化の最後にアプリケーションサーバ内で行うデータベースに対する処理フローを図 3-8 に示す。アプリケーションサーバとキューDB とのデータ送受信、及び業務用 DB とのデータ送受信に関して図式化しており、RDB 型でも KVS 型でも全く同じ処理フローを行う。業務用 DB に対してアプリケーションが実行する SQL でのデータ送受信は含めておらず、データベースに対するトランザクション単位で共通して行われる処理のみを対象としている。

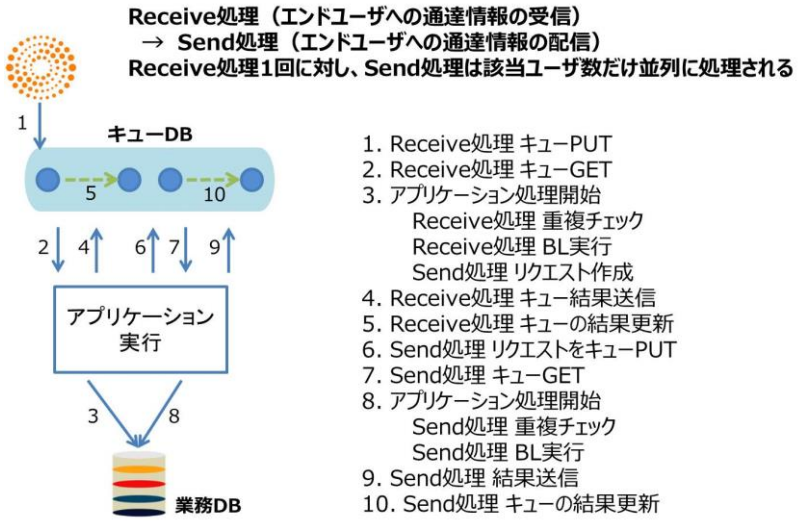


図 3-8: アプリケーションサーバとキューDB 間の処理シーケンス詳細

1→10 までの処理時間が t となる。この処理時間に対する影響は以下となる。

- アラート発生数の増加により、処理要求数が増加（1→10 に影響）
- 配信対象となるユーザ数の増加により、処理要求数が増加（3→10 に影響）

また RDB 型は 1→10 の全ての処理が物理的に同一のデータベースに実行されるのに対し、KVS 型では 3, 8 は業務用 DB, 1, 2, 4, 5, 6, 7, 9 はキューDB と物理的に異なるデータベースに処理が行われることになる。

性能測定では、1（アラート発生数）を一定とし、2（配信対象人数）を変更することで、システムの処理負荷の増加による影響と性能を確認する。
耐障害性実験では、1→7 及び 9→10 の処理で障害を検知する可能性がある。それぞれの処理で障害を検知し、検知した後の動作に問題がないか確認する。
性能劣化測定では、キュー用 DB のデータベースが縮退している状態で性能測定を行い、処理時間への影響と HW リソース（CPU，メモリ）の使用状況を確認する。

第 4 章　実装と実験

本章では KVS 型アーキテクチャと RDB 型アーキテクチャそれぞれで株価アラートシステムを実装したアプリケーションで実験を行い、結果について検証を行う。

4. 1　実験環境

実験は複数の物理サーバ上に VMWare を用いて構築された仮想サーバ環境（JAIST Cloud）上で実施した。実験に使用した HW 構成及び SW 構成は表 4-1 の通り。

表 4-1 HW/SW 構成

項目	アプリケーションサーバ	データベースサーバ	実験データ取得用サーバ
台数	4	1	1
CPU	Intel(R) Xeon(R) CPU X5690 @ 3.47GHz		
仮想CPUコア数	8	16	16
仮想メモリ	12	24	24
仮想HDD	500	500	500
OS	CentOS Linux release 7.2.1511 (Core)		
その他	JDK 1.8.0_111 Cassandra 3.0.11.1564	MySQL 5.7.17	JDK 1.8.0_111

プラットフォームの選択理由として、DB は MySQL，OS は Red Hat Linux の利用経験があるため、いずれも同系統の最新バージョン、かつ無償で使用できるものを選択している。これは構築時及び実験中の不要なトラブルを回避するためである。

実験用に構築した環境のネットワーク性能は実験の結果に影響がなく、システムが高負荷な状態でも信頼できるデータを取得できることを確認するため、各サーバ間のネットワーク通信速度に問題がないことを確認した。各サーバに OS をインストールした後に通信遅延測定プログラム（iperf https://iperf.fr/）を用いて測定を行った。結果を表 4-2 に示す。

表 4-2 通信速度の確認

標準的

```
[root@s106-064 ~]# iperf -c 150.65.106.60 -p 4000 -i 1
------------------------------------------------------------
[  3] local 150.65.106.64 port 17209 connected
with 150.65.106.60 port 4000
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec   113 MBytes   951 Mbits/sec
[  3]  1.0- 2.0 sec   112 MBytes   940 Mbits/sec
[  3]  2.0- 3.0 sec   113 MBytes   946 Mbits/sec
[  3]  3.0- 4.0 sec   112 MBytes   937 Mbits/sec
[  3]  4.0- 5.0 sec   112 MBytes   937 Mbits/sec
[  3]  5.0- 6.0 sec   112 MBytes   940 Mbits/sec
[  3]  6.0- 7.0 sec   113 MBytes   947 Mbits/sec
[  3]  7.0- 8.0 sec   112 MBytes   937 Mbits/sec
[  3]  8.0- 9.0 sec   112 MBytes   940 Mbits/sec
[  3]  9.0-10.0 sec   111 MBytes   932 Mbits/sec
[  3]  0.0-10.0 sec  1.10 GBytes   940 Mbits/sec
```

最速

```
[root@s106-062 ~]# iperf -c 150.65.106.64 -p 4000 -i 1
------------------------------------------------------------
[  3] local 150.65.106.62 port 40457 connected
with 150.65.106.64 port 4000
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec   722 MBytes  6.06 Gbits/sec
[  3]  1.0- 2.0 sec   554 MBytes  4.65 Gbits/sec
[  3]  2.0- 3.0 sec   530 MBytes  4.44 Gbits/sec
[  3]  3.0- 4.0 sec   631 MBytes  5.29 Gbits/sec
[  3]  4.0- 5.0 sec   547 MBytes  4.59 Gbits/sec
[  3]  5.0- 6.0 sec   538 MBytes  4.51 Gbits/sec
[  3]  6.0- 7.0 sec   562 MBytes  4.72 Gbits/sec
[  3]  7.0- 8.0 sec   530 MBytes  4.45 Gbits/sec
[  3]  8.0- 9.0 sec   547 MBytes  4.59 Gbits/sec
[  3]  9.0-10.0 sec   510 MBytes  4.28 Gbits/sec
[  3]  0.0-10.0 sec  5.54 GBytes  4.76 Gbits/sec
```

1 秒毎の通信速度(TCP)を計測しており、1GbE(1G bit per sec on Ethernet, IEEE 802.3ab)相当の処理性能が出ることを確認している。

一部の VM 間では、表 4-2 最速のように、他と比較して 4 倍以上と異常に高速な数値を得た。これは同一の物理 HW 上に異なる VM として構築された結果と考えることができるが、全体として、実験環境としては十分な性能が保障されており、各サーバ間の通信速度の遅延による障害検知の遅延やシステム全体の性能が劣化することがなく、構築した実験環境において十分に信頼できるデータを取得できることを確認できた。

4.2 実験の目的

RDB 型のアーキテクチャに対して KVS 型のアーキテクチャの有効性を確認するため、下記の実験を行う。

1. 性能
2. 耐障害性
3. 性能劣化測定

それぞれの実験の目的は以下の通り。

1. 正常時の性能測定
   機能要求を満たしたうえで、ユーザに送信するメッセージが生成された後、十分に短い時間(数秒以内)でのデータ配信が完了する。
   また、要求数(ユーザ数×メッセージ配信数)の増加に対する 1 台のアプリケーションサーバで実現可能なスループット、及びアプリケーションサーバの増加に対するシステム全体の処理性能の増加傾向を把握することで、必要なシステム構成の設計が可能となる。
   このため、以下の 2 つのパラメータを変化させて測定を行う。
   - 配信対象人数
   - アプリケーションサーバ数

以下の値を測定する。

- TPS（Transaction Per Seconds）
- 各サーバの CPU 使用率

TPS はシステム全体で処理する性能を把握する。また CPU 使用率は処理のボトルネック箇所を把握する。

2. 耐障害性実験

一般的にシステムの一部で障害が発生した場合においても、ユーザがサービスを利用可能な状態とするため、冗長性が必要となる。KVS 型のアーキテクチャにおいて、特にデータの永続化サービスとして利用している KVS 型データベース（Cassandra を利用）で障害が発生しても、ユーザがサービス利用可能な状態であることが必要となる。
この要求を満たすことを確認するため、使用しているデータベースに対して意図的に障害を発生させ、以下の状態を確認する。

- 障害を検出時の状態
- 障害検出後の状態
- 障害検出後にシステムを復旧した後の状態

それぞれの状態において、確認する項目は以下の通りである。

- アプリケーションから DB アクセス時に不必要なエラー（接続不能、接続時のタイムアウト、アプリケーションサーバのメモリ上の値と DB とのデータ不整合、クエリー実行不能、データロック状態に依る処理継続不能等）を発生し続けないか
- CPU 枯渇、またはメモリ不足により HW リソースが使用できない状態にならないか

また、システムの障害前と同様の性能が出力できるようにする復旧手順を確認する（上記のような問題が発生することなく、システムに障害が発生した前と同様の性能が出力されることの確認を行う）。

3. 性能劣化測定

システムの一部で障害が発生した場合、正常な状態と比較して性能劣化が予想される。ユーザにとってサービス利用可能なレベル（30%程度の性能劣化、新しい処理要求のタイムアウトが発生しないこと）であるかを確認する。この実験では、「2. 耐障害性実験」と同様に意図的に一部のデータベースを使用できない状態とし、この状態で「1. 正常時の性能測定」と同様に処理性能を測定し比較を行う。

4. 3 測定

4. 2 で説明した各実験において測定する対象を下記に記載する。

1. 正常時の性能測定

アプリケーションサーバ 1 台、2 台、3 台、4 台のケースそれぞれに対して、下記のデータを測定する。

- TPS（最大値）
- TPS（平均）
- CPU 使用率（アプリケーションサーバの最大）
- CPU 使用率（DB サーバの最大）

送信データ数（配信対象人数）を 10，50，100，150，200 と変化させ、RDB 型のアーキテクチャと KVS 型のアーキテクチャそれぞれに対して測定し、比較を行う。

2. 耐障害性実験

KVS 型アーキテクチャにおいて、アプリケーションサーバ 2 台の構成に対して実験を行う。
1 台のデータベースに対し、停止コマンド（kill -SIGKILL）を実行して強制終了を行う。
発生させるタイミングは下記の 2 通り。

- 処理要求が存在しない状態
- アプリケーションの処理実行中

データベースは 2 台で冗長構成を取っており、障害発生したデータベースに接続していたアプリケーションは、障害の発生していない、もう一方のデータベースに接続が移動する（F/O: Fail Over）。強制停止を実施した後、エラー検知することを確認する。

また、データを測定するタイミングは以下の通り

- 強制終了前（正常に 2 つのデータベースが動作している状態）
- 強制終了後（停止コマンドを実行して、1 つのデータベースが停止している状態）
- 復旧した後（停止していたデータベースを再稼働させ、正常な状態に戻した状態）

測定対象は以下の通り。

- TPS（最大値）
- CPU 使用率（アプリケーションサーバ F/O 移動元の最大）
- CPU 使用率（アプリケーションサーバ F/O 移動先の最大）
- CPU 使用率（DB サーバの最大）
- メモリ使用量（アプリケーションサーバ F/O 移動元）
- メモリ使用量（アプリケーションサーバ F/O 移動先）

配信対象人数は、最も影響が大きい（最も負荷の高い）と考えられる、200 とし、処理性能の劣化や CPU とメモリの使用状況に変化がないことを確認する。
復旧手順は、正常時のアプリケーション起動と同様の手順で行う。もし復旧後に正常な動作が行えなかった場合、障害検知後から復旧までに何らかの問題がシステムに発生したと考えられる。

3. 性能劣化測定

KVS 型アーキテクチャにおいて、アプリケーションサーバ 2 台、3 台、4 台の構成で、1 台のデータベースが停止した状態で、性能劣化がどの程度発生するかを確認するため、下記のデータを測定する。

- TPS（最大値）
- TPS（平均）
- CPU 使用率（アプリケーションサーバの最大）
- CPU 使用率（DB サーバの最大）
- メモリ使用量（ヒープサイズの空き容量の割合）

TPS に関しては、正常に全てのデータベースが動作しているときの値（「1. 正常時の性能測定」で取得した値）と比較を行い、各パラメータ変更時における差の増減を確認する。

また CPU 使用率、及びメモリ使用量を測定する対象は以下の通り。

- データベースの接続の移動元となるアプリケーションサーバ（移動元）
- データベースの接続の移動先となるアプリケーションサーバ（移動先）
- データベースサーバの停止に影響がないアプリケーションサーバ（移動無）

4. 4 結果

実験を実施し、下記に記載する結果を得ることができた。正常時の性能測定では、実験結果の特徴的な例として、最も処理負荷の高くなるケース(アプリケーションサーバ 4 台、配信対象人数 200)を掲載している。その他の結果は付録 A1_正常時の性能測定データ.pdf を参照。同様に、耐障害性実験の全データとグラフは、付録 A2_耐障害性実験データ.pdf に記載している。性能劣化測定で得られた全データとグラフは、付録 A3_性能劣化測定データ.pdf にそれぞれに記載している。

1. 正常時の性能測定

TPS(最大値)

図 4-1 に TPS(最大値)の値を示す。

TPS の数値は大きい方がシステムで処理できる要求数が多く、性能が高いと考えられるが、全体的に RDB 型のアーキテクチャよりも KVS 型のアーキテクチャの方が大きい値となっている。

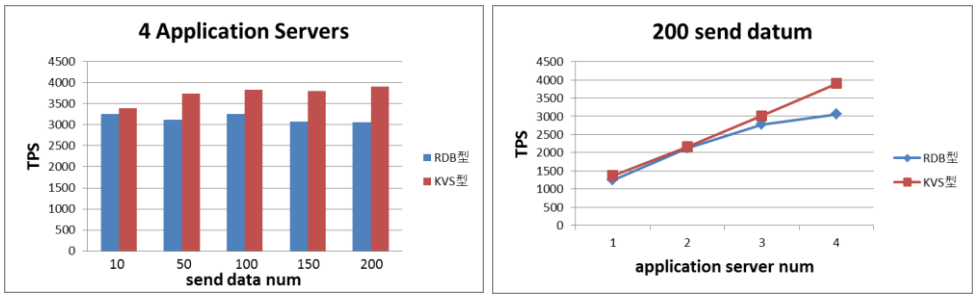


図 4-1: TPS 比較(最大値)
左:データ数変更 右:サーバ数変更

スケーラビリティを見た場合、RDB 型のアーキテクチャはアプリケーションサーバ数の増加に対し、TPS の増加が頭打ちになる傾向がみられる。対して KVS 型のアーキテクチャの増加傾向は線型であり、アプリケーションサーバ数に比例して TPS が増加している。

TPS(平均)

図 4-2 に TPS(平均)の値を示す。

アプリケーションサーバが 1 台または 2 台の場合、傾向として配信対象人数が少なく処理負荷が低い実験においては RDB 型のアーキテクチャの方が KVS 型のアーキテクチャより大きい数値(10→20%程度)となっているケースもあるが、アプリケーションサーバが 3 台または 4 台のケースの場合、処理負荷が高い実験においては KVS 型の方が RDB 型よりも大きい数値となっている。

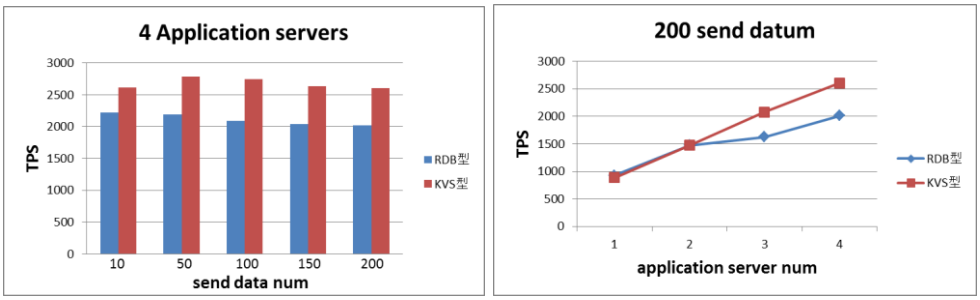


図 4-2: TPS 比較(平均)
左:データ数変更　右:サーバ数変更

スケーラビリティを見た場合、TPS(最大値)と同様に RDB 型ではアプリケーションサーバ数の増加に対し頭打ちの傾向になっているのに対し、KVS 型ではアプリケーションサーバ数に比例して TPS が増加していることが分かる。

CPU 使用率(アプリケーションサーバの最大)

図 4-3 にアプリケーションサーバの CPU 使用率を示す。

アプリケーションサーバの CPU 使用率は 90%を超過しない限り、大きな値となっている方が好ましい(多くの処理を行ってボトルネックなくシステム全体の TPS が向上していると考えられるため)。90%を超えると CPU リソースが枯渇し、アプリケーションは正常な動作が行えなくなってしまう可能性がある。

全体的に RDB 型のアーキテクチャよりも KVS 型のアーキテクチャの方が大きい値となっており、3 倍程度の値となっている。処理負荷が高くなるほど RDB 型のアーキテクチャは CPU 使用率が減少傾向となるのに対し、KVS 型のアーキテクチャは一定以上の CPU を使用し、処理負荷が増加に応じて CPU を使用していることが分かる。

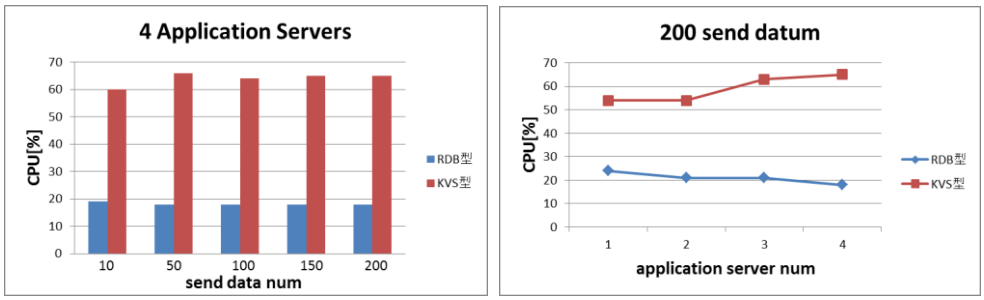


図 4-3: CPU 比較(アプリケーションサーバ)
左:データ数変更　右:サーバ数変更

CPU 使用率(DB サーバの最大)

図 4-4 に DB サーバの CPU 使用率を示す。

DB サーバの CPU 使用率は小さな値となっている方が好ましい。これは DB サーバがボトルネックになることなく、アプリケーションサーバの増加に対し、システム全体の TPS が向上することに繋がるからである。

全体的に RDB 型のアーキテクチャよりも KVS 型のアーキテクチャの方が小さい値となっており、CPU 使用率の差で 20%程度の余裕がある。

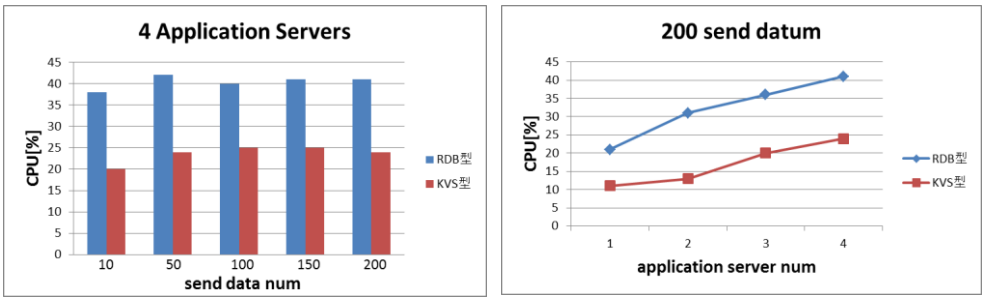


図 4-4: CPU 比較(DB サーバ)
左:データ数変更　右:サーバ数変更

2. 耐障害性実験

強制停止コマンドを実行して強制終了の実施

- 処理要求が存在しない状態
  停止直後、及び停止後の一定時間に至るまで特にエラーは発生しなかった。また、疎通確認を行ったところ、特にエラーなく実行可能であった。

- アプリケーションの処理実行中
  下記のエラーが発生した。重複を排除し、一連のエラーの起因となっている内容を抜粋して図 4-5 に記載している。全体のエラーログは付録 B を参照。

```
1004004 キューGET エラー(DEFAULT) @DBExecutorWorker-11(16702899)
1004003 キューPUT エラー(DEFAULT) @DBExecutorWorker-18(1250330448)
1003007 DB コミットエラー(メッセージコミット失敗) @EventProcessorWorker-0(177506604)
```

図 4-5: データベース障害検知時のログ(抜粋)

これらのエラーが発生した個所を図 4-6 に示す。

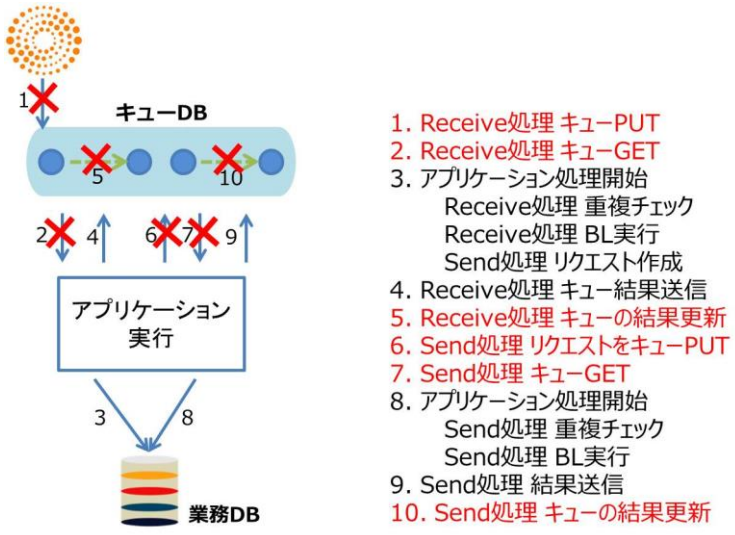


図 4-6: エラーの発生した処理

キューGET エラーはデータ取得実行中のエラーであり、2 または 7 の処理が該当する。
キューPUT エラーはデータ反映実行中のエラーであり、DB コミットエラーはデータ反映後の確定処理実行中のエラーであり、どちらのエラーも 5, 6, または 7 が該当する。いずれも障害発生時にデータベースにアクセスしていた処理でエラーが発生している。これらのエラーが発生した場合、冗長化しているセカンダリデータベースに接続が切り替わった後（F/O, Fail Over）、エラーのステータスとなったデータを再実行するか、アプリケーションがデータ登録できずにダンプしたファイルのデータを取込再実行することでリカバリが可能である。
この一連のエラーは 50[msec]程度の時間内で発生し、その後は特にエラーが発生し続けることなく、アプリケーションの処理を継続することができた。

障害の発生前後、及び復旧後に取得した値は以下に示す通り。
なお、復旧は意図的に障害を発生させたサーバにおいて、VM（OS）の再起動を行うことで、システム全体を復旧させている。

TPS（最大値）

表 4-3 に結果を示す。
障害検知後及び復旧後のいずれも性能は劣化せず、障害検知前よりも高い値となっている。障害検知後及び復旧後のいずれも問題なくシステムの処理性能を提供していることが分かる。

表 4-3 耐障害性測定データ（TPS 最大値）

確認 タイミング	TPS 最大値
障害発生前	2161
障害発生後	2713
復旧後	2409

CPU 使用率（アプリケーションサーバ F/O 移動元の最大）

表 4-4 に結果を示す。
障害発生後は、データベースが使用していた CPU リソースが使用されなくなったため、小さい値となっている。復旧後は、障害検知前よりもわずかに大きい値となっている。

CPU 使用率（アプリケーションサーバ F/O 移動先の最大）

表 4-4 に結果を示す。
障害発生後は、冗長化している 2 つのデータベースで分散していた全ての処理を 1 つのデータベースで行うことになるため、障害検知前よりも大きな値となっている。2 割程度の増加にとどまっている。復旧後は障害検知前よりも大きな値となっている。

表 4-4 耐障害性測定データ(CPU 使用率)

確認タイミング	CPU 使用率(最大)[%]		
	Application Server		DB server
	移動元 F/O	移動先 F/O	
障害発生前	62	62	13
障害発生後	31	73	15
復旧後	64	70	14

CPU 使用率(DB サーバの最大)

表 4-4 に示している。TPS(最大値)と同様の傾向を示している。

ヒープメモリ使用量

図 4-7 に結果を示す。

障害発生前後、及び復旧後のどのタイミングにおいても、アプリケーションサーバ F/O 移動元、アプリケーションサーバ F/O 移動先のいずれも GC が発生することで、ヒープの空き領域が同じように確保されることが分かる。

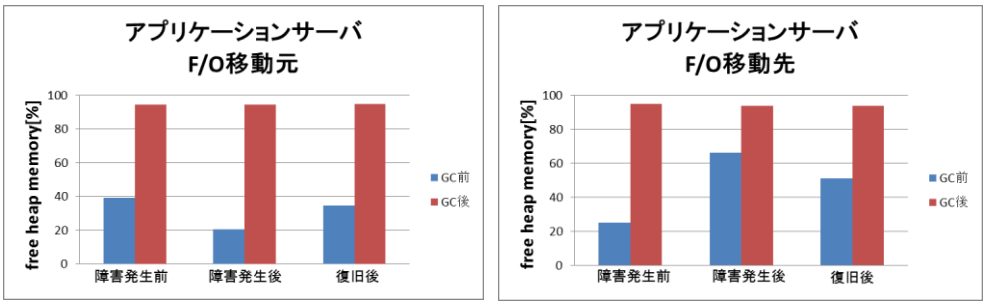


図 4-7: 耐障害性実験ヒープメモリ使用量
左:F/O 移動元 右:F/O 移動先

3. 性能劣化測定

- TPS(最大値)

図 4-8 に TPS(最大値)の値を示す。

全体的に正常時の値と比較して、1 台のデータベースが停止している状態の方が大きい値となっている。

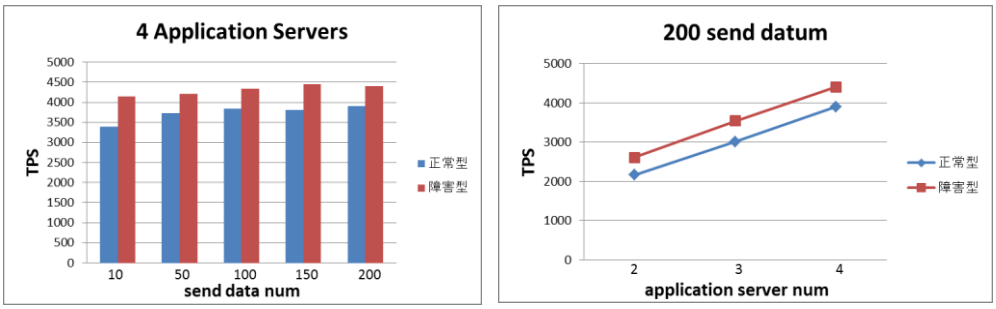


図 4-8: 性能劣化測定 TPS 比較(最大値)
左:データ数変更 右:サーバ数変更

・ TPS（平均）
図 4-9 に TPS（平均）の値を示す。
TPS（最大値）と同様に、全体的に正常時の値と比較して、1 台のデータベースが停止している状態の方が大きい値となっている。

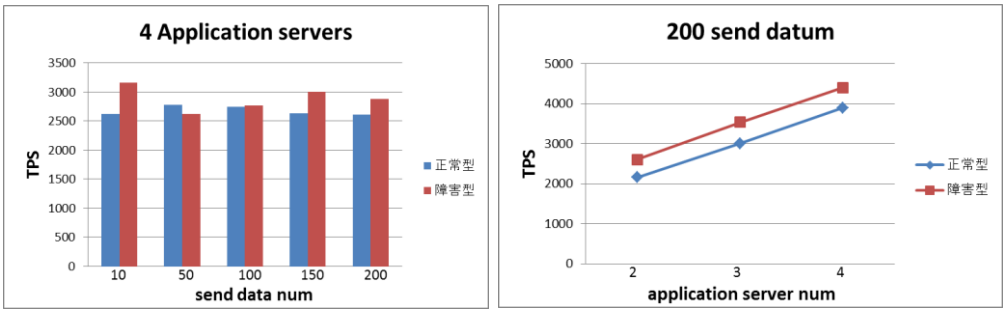


図 4-9: 性能劣化測定 TPS 比較（平均）
左：データ数変更　右：サーバ数変更

・ CPU 使用率（アプリケーションサーバの最大）
図 4-10 にアプリケーションサーバの CPU 使用率を示す。
データベースへの接続が変化しない（移動無、データベースを停止しても影響がない）アプリケーションサーバと比較して、データベースの停止が発生したアプリケーションサーバ（データベース接続 F/O の移動元）の CPU 使用率は小さくなる。データベースは冗長化しており、データベース接続 F/O の移動先となるアプリケーションサーバの CPU 使用率は、移動無アプリケーションサーバと比較して大きな値となっている。

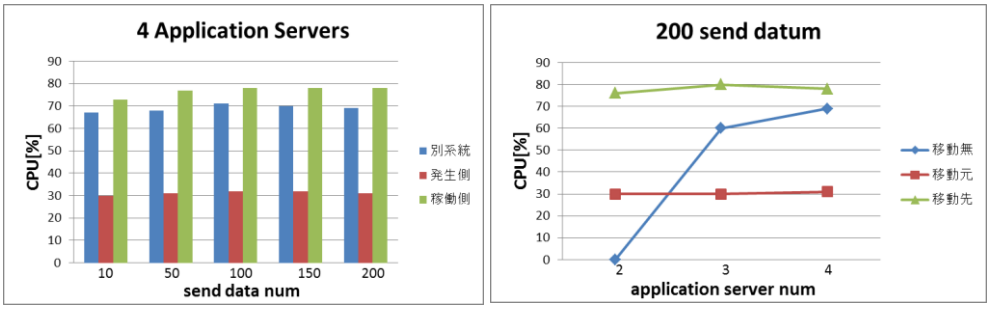


図 4-10: 性能劣化測定 CPU 比較（アプリケーションサーバ）
左：データ数変更　右：サーバ数変更

図 4-10 右において、アプリケーションサーバ 2 台の場合、データベース接続が移動無となるサーバは存在しないため、値は 0 となっている（グラフ作成ツールの仕様のためプロットしている）。

・ CPU 使用率（DB サーバの最大）
図 4-11 に DB サーバの CPU 使用率を示す。
全体的に正常時の値と比較して、1 台のデータベースが停止している状態の方が大きい値となっている。

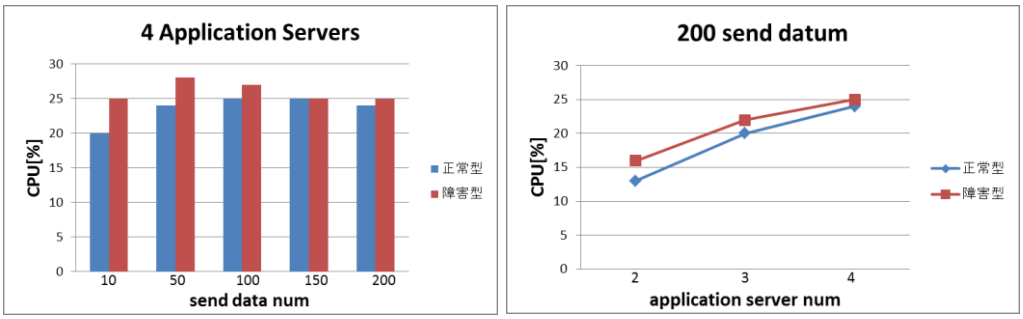


図 4-11: 性能劣化測定 CPU 比較(DB サーバ)
左:データ数変更　右:サーバ数変更

- メモリ使用量(ヒープサイズの空き容量の割合)
  データベース停止後にデータベース接続 F/O の移動元となるアプリケーションサーバ、移動先となるアプリケーションサーバ、移動しない(影響のない)アプリケーションサーバ、それぞれに対し、性能データ取得後の GC が発生する前後のタイミングでヒープメモリ領域の使用率を計測している。アプリケーションサーバ 4 台で計測した結果を表 4-4 に示す。

表 4-4 性能劣化測定メモリデータ

型	配信対象人数					測定
	10	50	100	150	200	タイミング
移動無	55.85	5.952	45.93	55.16	39.5	GC 前
	95.06	94.85	95.06	94.92	94.7	GC 後
移動元	73.96	8.167	38.4	30.6	40.92	GC 前
	94.64	94.58	94.58	94.56	94.58	GC 後
移動先	72	35.59	58.42	46.06	39.48	GC 前
	94.53	94.58	94.36	94.36	94.46	GC 後

GC は意図的にアプリケーションに対してコマンドを実行することで発生させている。
いずれのケースも GC 発生後は、ヒープ空き容量が 95%前後の値となり、不要なメモリを使用し続けることなく、問題なく空き容量が増加していることが分かる。

4.5 特定された傾向

実験の結果から特定された RDB 型と KVS 型の傾向は以下のようになる。

- 全体的に RDB 型よりも KVS 型の方が高い TPS となっており、性能が高い。
- アプリケーションサーバ数の増加に対し、RDB 型は性能の増加率がゆるやかであり効果が低い。KVS 型の方が TPS の増加率が高く、スケーラビリティが高い。
- KVS 型はキュー用 DB の障害後もデータベースへの接続の切替及び切替後の動作の継続が問題ない。
- KVS 型ではキュー用 DB が一つ停止した状態でも、大きく性能が低下することなく使用可能。
- RDB 型は KVS 型よりも業務用 DB サーバの CPU 使用率が高く、KVS 型は RDB 型よりもアプリケーションサーバの CPU 使用率が高く、ボトルネックが移動している。

上記を踏まえて、新しいアーキテクチャを用いるうえで適切な設計を行うことができるよう、次章で発生メカニズムの分析を行う。

第 5 章 分析

本章では、実験の結果について、RDB 型に対する KVS 型の有効な点、及び KVS 型を利用する上での考慮点について分析を行う。

5. 1 性能特性

前章では配信するメッセージ数が 10, 50, 100, 150, 200 と増加するにつれてシステムの処理負荷は高くなる。特にメッセージ数が 100 以上の場合において RDB 型より KVS 型の方が、TPS が高い値になっていることを示した。この理由として以下の 2 点が考えられる。

- キューDB に関して、RDB 型はアプリケーションサーバとは異なる VM 上に存在するデータベースにアクセスするのに対し、KVS 型はアプリケーションサーバと同一の VM 上に存在するデータベースにアクセスするため、通信処理のオーバーヘッドが少ない
- RDB 型は同一のテーブルに複数のアプリケーションからアクセスするため、処理の競合が発生する。同一の行データに対する競合は発生しないが、INDEX の更新での競合やデータベースサーバのセッション管理や更新データ保存において同時実行数が増えることになる。対して KVS 型は 1 データベースに 1 アプリケーションからのアクセスしか発生しないため、RDB 型と比較して、複数アプリケーションサーバ間で処理の競合が発生せず、応答時間が短くなる。どちらのアーキテクチャも write-write, read-write の競合は発生するが、KVS 型の方は頻度が少なく、競合する確率が低くなると考えられる

いずれもモデル上では Σm_{ci} が小さい値となり、処理時間 ti が小さくなったことで数値の差となったと考えられる。

特定の条件下（比較的負荷が低い場合）では RDB 型の方が KVS 型よりも TPS が高くなる場合が検出されている。アプリケーションサーバが 2 台以下のケースが相当する。これは KVS 型のデータベースはデータの冗長化を行っている分の処理コストが、負荷が低い場合、システムが処理を行う全体の処理コストに対する割合が大きい。一方 RDB 型は冗長性が存在しないため、全体の処理性能が高くなっていると考えることができる。このような状態はシステムの利用状況に問題がなく、特にアーキテクチャに依存した差が問題になることも少ない。ユーザの利用頻度が高くなりシステムが高負荷となる状態を考えた場合、KVS 型の方が安定して高い性能を出すことができると考えられる。

5. 2 スケーラビリティ特性

RDB 型はアプリケーションサーバの数が増加するにつれて処理性能が頭打ちになる傾向が見られる。対して KVS 型はアプリケーションサーバ数が 1 台→4 台に増加すると、TPS は線型に近い増加となり、アプリケーションサーバ数と性能がほぼ比例した高いスケーラビリティとなっている。

5. 3 負荷分散について

アプリケーションサーバの CPU 使用率は高い方が好ましい。ただし 90%を超えるとアプリケーションが正常に動作しなくなる可能性があるが、この上限値に達しない限りは可能な限り処理をし続ける状態であれば TPS が高くなるためである。

RDB 型のアプリケーションサーバの CPU 使用率をみると、アプリケーションサーバの台数が 1 台→4 台に増加するにつれて CPU 使用率が下がる傾向にある。これは DB サーバの負荷が高くなり、DB サーバからの応答が返りにくくなったことで空き時間が増加したためと考えられる。結果としてアプリケーションの処理要求数に対し、TPS が下がる傾向になったと考えられる。

KVS 型の方は DB サーバの処理負荷が下がった分、アプリケーションサーバの CPU 使用率が増加していると考えられる。DB サーバはアプリケーションサーバの 2 倍の CPU 性能を持つ（同一 CPU 種であり、コア数がアプリケーションサーバは 8 に対し、DB サーバは 16）。このため、

アプリケーションサーバの CPU 使用率の増加

= DB サーバの CPU 使用率の差×2 / アプリケーションサーバ台数

と増加量の見積もりを立てることができる。

アプリケーションサーバ 4 台、配信対象人数 200 のデータを例として、値を比較してみると

アプリケーションサーバの CPU 使用率増加: 47[%]

DB サーバの CPU 使用率の差: 17[%]

であり、上記の計算式からは、アプリケーションサーバの CPU 使用率は 8→9%の増加となるところ、30%以上の開きが出ている。これは、

- TPS が 30%ほど向上しているため CPU の使用率も向上していること
- 冗長構成を取っている相互のデータベース間でのデータ差異を相互に反映していること

この 2 点を考慮すると、

17[%]×1.3（TPS 増加分） × 2（冗長構成のデータ反映分） ×2 / 4 = 44.2 [%]

となり、妥当な増加率であると考えることができる。

5. 4 耐障害性実験

アプリケーションで障害を検知してから、冗長化しているデータベースへの接続に切り替わるまでの時間が短い方が好ましい。切替操作の完了は、障害を検知した後に、次の処理を切替わり先のデータベースを利用して処理を行うことが可能になることである。KVS 型は RDB 型と比較して短い時間で行われている。RDB 型の場合はデータ量や動作環境に依存するが、切替にかかる時間は 10 秒から数分となるのに対し、KVS 型では 50[msec]程度で切替が行われている。

RDB 型では切替で以下の操作を行っている。

- アプリケーションがプールしているコネクションの終了
- アプリケーションが全てのコネクションに対する終了処理後、プールの再作成
- データベースが待機系で処理を開始するための初期化
- データベースがアプリケーションからの接続を受付

対して KVS 型では切替で以下の操作を行わないため、短時間での切替となっている。

- 待機系データベースの初期化（全てのデータベースが常にアクティブな状態のため）
- プールの再作成（アプリケーションは起動時に切替先を把握しており、障害検知時後にプールを再作成することなく即座に切替先に接続する）

KVS 型アーキテクチャのアプリケーションサーバで発生しうる障害に対し、それぞれに対する復旧手順を考える。まず発生しうるケースは図 5-1 となる。

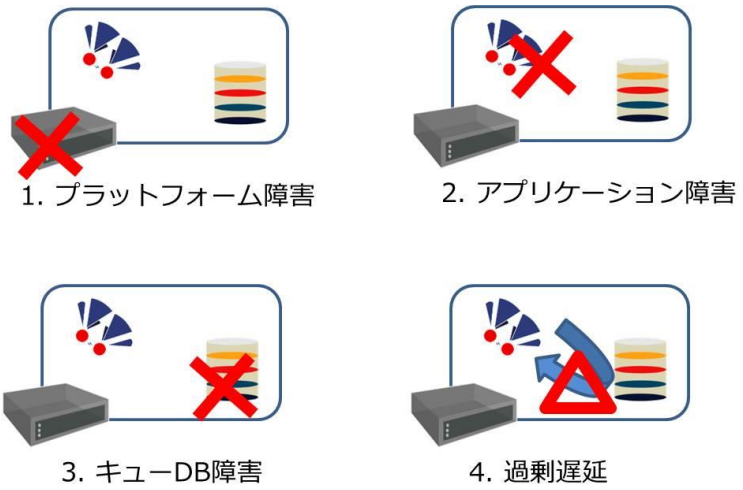


図 5-1: アプリケーションサーバ障害発生ケース

それぞれの障害と復旧までの考え方を以下にまとめている。

1. プラットフォーム障害
   アプリケーションが動作しているプラットフォーム（CPU，メモリ、HDD といった HW，VM や OS）に故障が発生した場合。アプリケーションサーバで動作している全てのアプリケーションが使用できなくなる。アプリケーションのレベルでの対応は困難である。処理中の要求は正しく終了することもできない状態になる。

2. アプリケーション障害
   アプリケーションプロセスに故障が発生した場合。アプリケーションが使用しているメモリ上のデータ不整合、メモリ不足、クラッシュといった状態が発生しアプリケーションが使用できなくなる。キューDB は使用可能な状態だが、アプリケーションが正常に動作しないため、バックグラウンドでデ

ータの複製を行う機能のみが動作している状態となる。

3. キューDB 障害

キューDB に故障が発生した場合。キューDB が使用しているメモリ上のデータ不整合、メモリ不足、クラッシュといった状態が発生し、キューDB が使用できなくなる。アプリケーションは使用可能な状態だが、キューDB が使用できないため、新しい要求に対してはエラーの応答を返し、処理中の要求も継続することができない状態となる。

4. 過剰遅延

システム内において、どこにも故障は発生しない。しかしシステムに対する要求が過剰に大きくなり、アプリケーションサーバの CPU リソースが枯渇し、正常な処理が行えなくなる。対応は状況に応じてさまざまであるが、手法については運用マニュアル等に記載してあることが望ましい。

それぞれに対する復旧方法を表 5-1 に示す。

表 5-1 障害からの復旧方法

項目	1. プラットフォーム障害	2. アプリケーション障害	3. キューDB障害	4. 過剰遅延
復旧方法	・電源停止が必要 ・部品交換を行う等、HWの修復が発生する ・OSの再起動を行う ・復旧後に、ユーザが必要な処理を再実行する	・アプリケーションプロセスの再起動を行う ・復旧後にキューDBに格納されていたデータから処理を再実行することが可能	・キューDBプロセスの再起動を行う ・キューDBが冗長化されていれば、エラーとなった要求のみ再実行することでデータの整合性を取ることが可能	・新しい要求を受け付けないようにする ・アプリケーションの同時実行数を少なくする ・縮退して運用する ・キューに滞留している要求を十分に少なくする必要がある

前章の実験では、3 の状態を意図的に発生させることにより、アプリケーションによる障害検知と復旧後の動作を確認した。

障害発生時の影響範囲は図 5-2 となる。

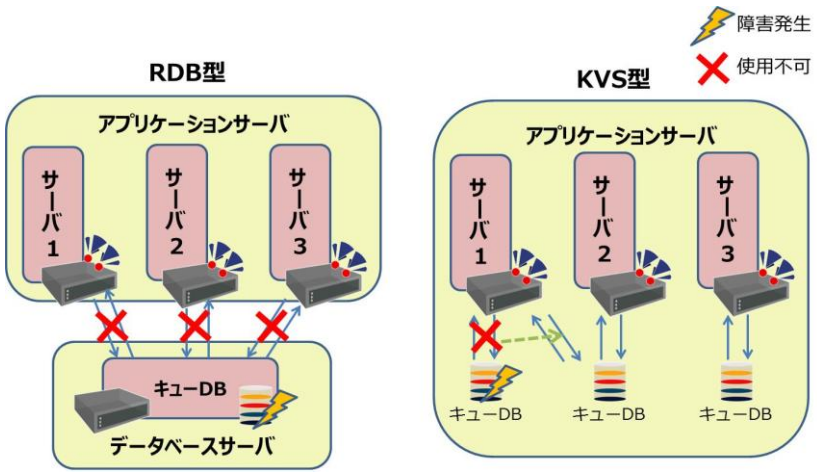


図 5-2: キューDB 障害発生時の影響

キューDB に障害が発生した場合、KVS 型の方がアプリケーションサーバへの影響範囲が小さく、障害発生時にも利用可能な状態を保てる。RDB 型のキューDB は中央集約型となっており、全てのアプリケーションサーバが同じキューDB を使用しているため、全てのアプリケーションサーバの動作に影響が出る。障害検知後はキューDB が復旧しない限りどのアプリケーションサーバも使用できない状態となるため、復旧されるまでシステム全体が使用できないことになる。一方、KVS 型の場合は各アプリケーションサーバで別々のキューDB を使用して分散して処理を行っているため、障害の発生したキューDB を使用していたアプリケーションのみに影響が出る。障害検知後、冗長化しているデータベースに接続が切り替わり、切り替わり先となるアプリケーションサーバの処理は最大で通常の 2 倍程度の負荷が発生し、処理負荷の偏りから性能が低下する可能性もあるが、システム全体としては使用可能であり、障害の発生したアプリケーションサーバを使用していたユーザのみ一時的にエラーが発生することになる。このエラーは完全に抑制することは不可能である。

要求完了状態での障害検知

受け付けた処理を全て実行した状態でもアプリケーションではシステム管理用にキューDB に対する死活監視や運用のためのデータ取得を定期的に行っている。この状態で障害が発生した場合を確認している。データベース接続用に JDBC ドライバを用いている。これを用いる際に複数のデータベース接続先を設定することが可能であり、JDBC ドライバが出力するログから定期的にデータベースに対して死活監視を行っている(デフォルトでは 30 秒間隔で実施している)ことが分かったため、アプリケーションからはキューDB に対する死活監視の機能を使用しないで実験を行った。結果として 1 台のデータベースを停止してもエラーとなるログも出力されることなく処理を継続することができた。これは JDBC ドライバの機能でデータベースの異常を検知し、あらかじめ設定していた切替先データベースに自動的に切替が行われたと考えられる。アプリケーションは起動時にデータベースに対する接続をプールしており、処理中はプールしている接続を繰り返し使用しているが、既に JDBC ドライバの機能でプールしている全ての接続が切り替わっていたため、エラーが発生することなく処理を行うことができたと考えられる。

アプリケーションの処理実行中での障害検知

いずれのエラーも下記の 2 パターンで発生している。

- 新しく処理の要求を行ったタイミングでエラーが発生
  - 接続先データベースが停止しており、使用できない状態
- 処理中に強制停止したためにエラーが発生
  - アプリケーションは正常な応答を待っていた状態

処理要求に対し応答がないためにタイムアウトが発生する、という現象は発生しておらず、エラーの収束後は問題なく処理を行うことができることから、障害の発生時にアプリケーションがデータベースに対し処理を行っていたもののみに影響が出ると考えられる。

データベースの復旧方法

データベースに格納されているデータはHW障害がない限り、データの実体が破壊されているケースは存在しないので、データベース自体を復旧させればよいことになる。

この論文における実験では OS を再起動することでシステムを復旧させている。データベースのみの再起動で復旧できることが望ましいが、サービスとして再起動するスクリプトの実行時にエラーが発生し、実行することができなかった。これはインストール時の設定あるいは構築環境に依存した問題であると考えられる。

復旧後の CPU 使用率

システムの復旧後は最大で 8%CPU 使用率が上昇している。この原因としては、冗長化しているデータベース間で、障害検知後→復旧までに更新されたデータの同期が行われているためと考えられる。このデータの同期を行うための CPU の処理コストの分、正常にアプリケーションの処理を行うために必要なCPU使用率に対し、復旧後は大きな値になったと考えることができる。なお、この CPU が増加する時間幅は障害検知後→復旧までに更新したデータ量に依存すると考えられるが、実験では 10 分以上かかる処理を行っており、復旧後の CPU 使用率が増加している時間も 10 分以上、継続していた。CPU 使用率の増加があったものの、90%を超えることはなかったため、復旧後は正常に処理を行えたと考えられる。

メモリの使用状況

GC はアプリケーションの処理時間に対して非常に小さい時間で行われている(コマンド実行時の体感時間 0[sec])。空き容量が増加するのは、アプリケーションが一時的に利用したものの、恒久的には不要となったオブジェクトがメモリ上に残留しており、GC を実行することで削除され、使用可能なメモリ空間が大きく確保された結果と考えることができる。

また、この実験においてはメモリリークが発生していない。仮にメモリリークが発生した場合は、下記のような現象が発生する。

- GC が実行されてもメモリ空間が十分に確保されない。
- 少しずつメモリ上にオブジェクトが残留し続け、アプリケーションが処理を繰り返し続けると、メモリ空間に十分な空き領域を確保できなくなり、メモリ不足が発生しエラーとなる。

数秒で正常な状態になると考えられるため、特に大きな問題として取り扱う必要がない。

5.5 性能劣化特性

データベースを 1 台停止した状態で、どの程度の性能低下が発生するかを確認するため、正常時の性能測定と同じ処理要求を実行することで比較を行っている。

性能の比較

耐障害性実験と同様の傾向となっているが、全体的に正常時の値よりも大きな値を示しており、最大

で 20%程度の差となっている。これは冗長構成がなくなったことが原因として考えられる。データベースが 1 台停止しているため、データベース間でデータの同期を行う処理が実行されなくなったため、相対的にアプリケーションから要求された処理のみにリソースを割くことが可能になっている。結果としてアプリケーションに対する応答時間が短くなり、システム全体として TPS が増加したと考えられる。

スケーラビリティの比較

正常系と同様に、サーバ台数の増加に対して線型に TPS が増加している。データベースが 1 台停止すると、冗長化している、もう片方のデータベースの処理負荷が上がることになるが、正常な状態の 2 倍の処理を行っても CPU 使用率が 90%を下回っている。このためアプリケーションサーバの処理負荷が増加しても、性能が劣化することなくシステム全体として処理を実行できたためと考えられる。

CPU 使用率

（移動元、移動先、移動無は「4. 3 測定」で定義したとおり）

データベース接続の移動元となるアプリケーションサーバの CPU 使用率は、データベースが使用しなくなった分、小さくなっていると考えられる。移動無アプリケーションサーバの値と比較すると、CPU 使用率で 30%以上、小さくなっている。対してデータベース接続の移動先となるアプリケーションサーバの CPU 使用率は大きくなっているものの、移動無アプリケーションサーバの値と比較すると CPU 使用率が 20%程度、大きくなっている。移動元アプリケーションサーバの CPU 減少値 30%と等しくならない原因として以下が考えられる。

- 冗長化しているデータベース間でデータの冗長化を行わなくなった
- 内部処理（死活監視、データ管理等）に割く処理コストの増加が、さほど発生しない
  - アプリケーションサーバからの処理要求に応える以外の処理コストが少ない

メモリ使用量

GC の結果、メモリの空き容量が常に 95%程度となっている。1 台のデータベースが停止した状態で処理を継続しても不必要にメモリを圧迫するデータが残留することがないことが分かる。また障害検知後に不要な参照が残留し続けるようなメモリリークが発生することなく、システムを利用可能な状態であると考えられる。

5. 6 追加実験

実験で得られた結果に対して、下記の項目について予想と異なる結果が見られた。より詳細な確認を行うため、それぞれ追加で実験を行っている。全データについては付録 C を参照。

1. 追加実験 1（メモリ使用量の再確認）

   性能劣化測定のメモリの使用量の結果をみると、配信対象人数 50 のとき、メモリ使用量が大きくなっており、特にヒープメモリの空き領域が 10%に満たない状態となる結果もあった。このため、アプリケーションサーバ 4 台の構成でデータベース 1 台を停止させた状態で追加実験を行い、メモリの使

用量が適切か再確認を行った。

実験は 10 回繰り返し、メモリの使用状態をそれぞれ確認した結果を表 5-2 に記載する。

表 5-2 性能劣化測定追加実験(メモリ使用量)

実施回	free heap memory[%]							
	App Server1		App Server2		App Server3		App Server4	
	GC 前	GC 後	GC 前	GC 後	GC 前	GC 後	GC 前	GC 後
1	53.57	94.52	47.51	94.53	43.71	94.17	**61.7**	93.94
2	38.89	95.03	50.69	94.93	37.55	94.57	60.24	94.47
3	28.57	94.75	44.84	94.63	37.59	94.36	50.76	94.27
4	40.81	95.04	36.35	95.03	32.73	94.47	35.93	94.36
5	48.91	95.05	31.25	95.06	51.1	94.46	44.44	94.35
6	56.53	94.73	55.95	94.74	47.25	94.26	31.37	94.37
7	40.67	95.05	59.13	95.03	43.66	94.46	50.25	94.35
8	42.96	94.92	44.05	95.06	44.46	94.56	39.56	94.46
9	38.99	95.05	54.27	94.92	**21.8**	94.5	44.84	94.35
10	32.14	95.05	44.49	95.02	28.26	94.47	41.86	94.36

平均:43.49[%] 最大:61.66[%] 最小:21.79[%]

ヒープメモリの空き容量が 10%以下になる現象は再現できなかったものの、最小→最大までの幅も 40%程度の差があり、特に規則性は見られないことが分かった。何らかの値が影響を与えた可能性があるが、実験を重ねて行う中で、データを取得するタイミングに依存して発生した現象あり、特に大きな問題とはならないと考えられる。

2. 追加実験 2 (2 台のデータベース停止状態の性能確認)

性能劣化測定において、1 台のデータベースが停止した状態でのデータを取得したが、より大規模なシステムにおいて、アプリケーションサーバの数が多くなった場合、複数のデータベースの停止が発生するケースも考えられる。このため、アプリケーションサーバ 4 台の構成において、2 台のデータベースが停止した場合、システムの性能がどのように変化するかを確認するため、データを取得した。図 5-3 に結果を示す。

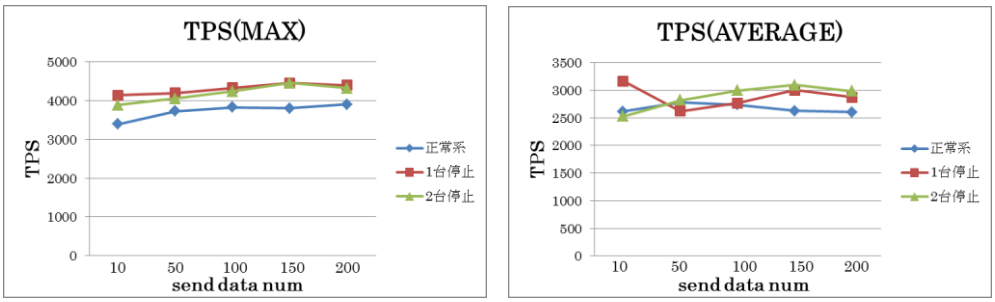


図 5-3: 性能劣化測定 追加実験(TPS 比較)
左:最大値 右:平均

2 台停止しても問題がなく、コンピュータリソースを過剰に消費することなく、不要なエラーが発生し続けることもなくデータを取得できた。性能に関しては、TPS と CPU 使用率ともに 2 台停止時の時も 1 台停止時の時と、ほぼ同じ傾向となっており、TPS(最大値)と TPS(平均)共に正常時よりも高い値となっており、TPS(最大値)は 1 台停止時の方がやや高い値となったが、これは負荷が低かったため、タイミングよくデータ測定が行われた影響と考えられる。全体の傾向として、TPS(平均)は 2 台停止時の方がやや高い値となっている。

3. 追加実験 3 (アプリケーションサーバ同時実行数の変更)

TPS 全体の処理傾向として、配信対象人数(send data num)が 50 を超えると数値があがりはじめ、100 以降は同程度の性能を出力している状態ともとらえることができる。これはアプリケーションサーバの同時実行スレッド数を 16 と設定しているため、配信対象人数が 10 だと、全てのスレッドを使用する状態にならず、50 を超えると全スレッドが同時に処理を行う状態になるためではないかと推測した。このため、アプリケーションサーバの同時実行スレッド数の設定を 8 とし、配信対象人数が 10→200 の範囲で同程度の性能値が現れるか確認を行った。アプリケーションサーバ数は 2 台、KVS 型を用いて実験を行っている。図 5-4 に結果を示す。

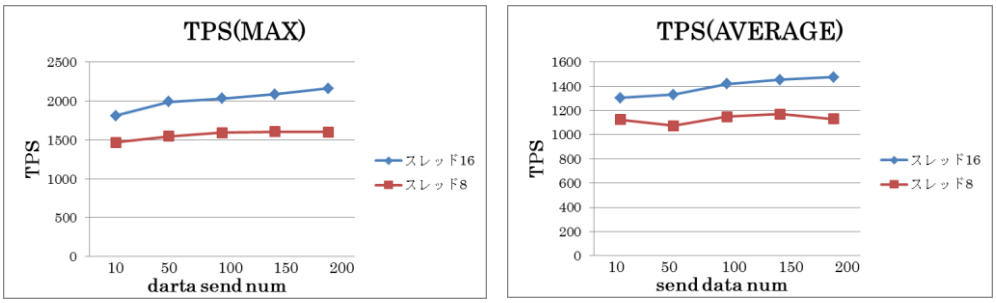


図 5-4: 追加実験 3 TPS 比較
左:最大値 右:平均

スレッド数が 16 の時に対し 8 にした場合は、TPS 値は全体的に 10→20%減少している。それぞれを相対的な数値として比較を行うと、スレッド数 16 での最大→最小の差は、TPS(MAX)で 19%, TPS(AVERAGE)

で 13%となっている。対してスレッド数 8 での最大→最小の差は、TPS(MAX)で 9%, TPS(AVERAGE)も 9%程度となり、スレッド数 16 に比べてスレッド数 8 の方が小さい差となっており、変化率が緩やかである。予想通り、配信対象人数の影響度が下がり、性能に差が出なくなってくる傾向があることが分かり、実験で得た結果の妥当性を確認することができた。
結果として TPS の値はアプリケーションの同時実行数の影響が大きいものの、結果として取得した値の傾向として、さほどの違いが出ないことが分かった。アプリケーションの同時実行数を変化させる場合、全てのデータを再取得することにもなるため、今回は実験としては行っていない。全体の傾向を把握する意味では十分な結果を得ていると考えられる。

いずれの追加実験の結果も、実験（正常時の性能測定、耐障害性実験、性能劣化測定）で取得した結果に対して影響がないことが分かった。

5.6 分析結果のまとめ

KVS 型は RDB 型と比較してスケーラビリティに優れており、システムを利用する上で十分な信頼性を備えていることが分かった。このアーキテクチャを利用する上で、本章で分析した結果を総合して判明した留意点を下記に記載する。

- 発生しなかった問題
  - 障害発生後のメモリリーク
  - データベース停止時の性能劣化
- 問題が発生するが極小的な影響に留まる問題
  - データベース障害発生時のエラー検知（重複したエラーとならない）
  - メモリ使用量の増加（GC による解決）
- 頻度が低く、システムの特性上、問題とならないもの
  - 負荷が低い場合、RDB 型の方が KVS 型よりも高い性能となる

この他に、アプリケーションサーバに関しては RDB 型よりも KVS 型の方が高性能なサーバが必要となる。これはシステムの CPU ボトルネックがデータベースサーバからアプリケーションサーバに移動しているためである。

第 6 章　議論

本章では、KVS を用いた新しいアーキテクチャについて、下記への適用、実現性について議論を行う。

1. 株価アラート配信システム
2. キューDB を利用する別システム（貨物データ配信システム）
3. キューDB を必要としない、分散 DB を利用するシステム
4. データベースへの依存度が低い、一般的な分散システム

6. 1　株価アラート配信システム

(1) 正常時の性能測定の予想と結果対比

性能測定の予想

1. TPS 値
   - DB サーバの負荷が小さくなる。
   - 同一テーブルに対する競合が少なくなる。
   - キューDB にアクセスする際に、別サーバへの通信が発生しない。

これらの理由から、RDB 型よりも KVS 型の方が全体的に TPS 値は大きくなり、アプリケーションサーバの台数を増やすと、RDB 型は頭打ちの傾向となるスケーラビリティに対し、KVS 型の方は台数に比例した線型のスケーラビリティとなる。

2. CPU 使用率

RDB 型は DB サーバに負荷が集中するため DB サーバの CPU 使用率が高くなり、DB サーバがボトルネックとなる傾向がみられるのに対し、KVS 型はアプリケーションサーバの CPU 使用率が高くなり、DB サーバがボトルネックとなりにくくなる。

データ永続化サービスに対して SQL の実行回数から CPU 使用率の減少幅を予測すると、

アプリケーションサーバ用として処理キュー永続化のためのアクセス --- ⅰ

合計：6 回

キューPUT、commit 処理

キューGET、commit 処理

ステータス更新、commit 処理

業務アプリケーションとしてのアクセス --- ⅱ

合計：7 回

重複確認

アプリケーション用データ取得

アプリケーション実行データ作成

後続処理用データ作成、commit

ステータス更新、commit

RDB 型はデータベースサーバに対して ⅰ + ⅱ =13 回の DB アクセスが発生するのに対し、KVS 型はデータベースサーバに対し ⅱ = 7 回の DB アクセスが発生することになる。このため、DB サーバの使用率は、KVS 型の方が RDB 型の方に対し、4→5 割程度、CPU 使用率が下がると考えられる。

逆に、i の DB アクセス処理が発生するため、KVS 型の方が RDB 型よりもアプリケーションサーバの CPU 使用率は DB サーバの CPU 使用率の減少幅×2 程度、増加すると考えられる（DB サーバの方がアプリケーションサーバの倍程度のスペックとなっているため）。

性能測定の結果対比

1. TPS

予想通り KVS 型の方が全体的に大きな数値を示し、スケーラビリティはアプリケーションサーバに比例した線型での増加を見ることができた。また、KVS 型の方が予想される処理量に対し、期待されるシステム性能を予測しやすく、アプリケーションサーバを適切に追加し、より拡張しやすいシステム構築が実現可能となる。

2. CPU 使用率

最も負荷の高くなるケース（アプリケーションサーバ：4 台、配信対象人数：200 人）において
DB サーバの CPU 使用率は以下の通り。

RDB 型：41%
KVS 型：24%

相対的に 4 割程度、使用率が小さくなっている。他のテストケースでも同程度の減少幅となっており、予想通りの CPU 減少幅となっている。

アプリケーションサーバの CPU 使用率は以下のように 47%の増加が発生している。

RDB 型：18%
KVS 型：65%

予想値の計算式であれば(41-24)×2=34[%]程度であるが、これは

- アプリケーションサーバ台数による処理負荷の分担
- KVS 型データベースの冗長化によるデータの同期化

この 2 点の考慮が不足していたと考えられる。増加傾向に関しては「4. 4 結果」で示した通り、妥当な値であると考えられる。

(2) 耐障害性実験の予想と結果対比

耐障害性実験の予想

1. アプリケーションサーバのエラー検知

下記のタイミングでエラーを検知する。

- アプリケーションから新しくデータベースにアクセスが発生するタイミング
- アプリケーションがデータベースに処理を要求し、応答を待っているタイミング

また、エラーの検知回数は、プールしているデータベースへの接続 1 つにつき、それぞれ 1 回、障害を検知する。

2. アプリケーションサーバからの接続切替

JDBC ドライバの機能で接続が切り替わる。データベースへの接続定義に複数の接続先を設定することが可能である。①のエラー検知後、数秒程度の時間が経過した後、再接続が終了し、アプリケーションの処理が再開可能な状態となる。

3. データストアサービスの復旧

データストアサービスのプロセスを強制終了するため、下記のいずれかの復旧を行う
ⅰ．データストアサービスのプロセスを再起動
ⅱ．OS の再起動
ⅲ．VM イメージからの復旧

4. アプリケーションサーバの復旧
復旧方法は 3 の復旧方法にも依存しているが、下記のいずれかを行う。
ⅰ．アプリケーションのプロセス再起動
ⅱ．OS 再起動後、アプリケーションの再開
ⅲ．VM イメージから復旧後、アプリケーションの再開

3，4 において、いずれの復旧を行ったとしても、特に不要なエラーは発生することなく、障害発生前と同程度の性能でシステム利用可能な状態になる。

耐障害性実験の結果対比

1. アプリケーションサーバのエラー検知
データベースの障害を検知したアプリケーションのログから、予想したタイミングでのみエラーを検知していることが分かる。このログに出力されたエラー検知の件数と、プールしているデータベースへの接続数を比較するとログに出力されたエラー検知数の方が少ない。これは、データベースの障害発生時に、プールしているデータベース接続の全てが使用されてはいないためと考えられる。障害検知をしていないデータベース接続は、死活監視機能でエラーを検知し、自動で接続が切り替わったと考えられる。
また、繰り返しエラーは発生していないことから、プールしている各データベース接続が使用できない状態となっているプライマリデータベースへの接続情報が残留することなく、予想通り各データベース接続 1 つにつき、障害検知は 1 回のみ行われていると考えられる。

2. アプリケーションサーバからの接続切替
特にアプリケーションでデータベース接続を切替る実装を追加することなく、データベースの障害をエラー検知した後、次の処理要求に対して継続して実行可能となったことから、予想通り JDBC ドライバの機能で接続切替を実施することができた。
また接続切替に要する時間は、予想よりも速く、1 秒に満たない（エラーのログ検知時間幅を見ると 50msec に満たない）時間内に接続が切り替わっている。これは初回接続時に指定した複数のデータベース接続先情報を確保しているからではないかと予想している（プロセス起動時に複数のデータベースへの接続を確立しプールして保持しているが、この処理に要する時間が RDB 型は数秒程度に対し、KVS は 10 秒以上と明らかな差が出ているため）。

3. データストアサービスの復旧
「ⅲ.VM イメージからの復旧」を行う必要はなかったが、「ⅰ．データストアサービスのプロセスを再起動」で復旧を行うことができず、「ⅱ．OS の再起動」での復旧となった。復旧を行う上で作業コストに差はないものの、復旧までの時間及び影響度は課題と認識している。

4. アプリケーションサーバの復旧
3 の復旧方法に伴い、OS 再起動を実施後、アプリケーションをプロセスから起動を行うことで復旧を実施した。OS 及びアプリケーションのメモリ情報が完全に初期化されることもあり、復旧後は予想通り不要なエラーが発生することなく、システムを利用可能な状態となった。

(3) 性能劣化測定の予想と結果対比

性能劣化測定の予想

1. アプリケーションサーバの処理状態

   データベースが 1 台停止した状態でも、特にエラーなくアプリケーションは使用可能な状態である。

2. 処理性能

   キュー用データベースのうち、一つが使用できなくなることで、全体の処理性能（TPS 値）は小さくなる。特にアプリケーションサーバ 2 台の場合は影響が大きいため、性能劣化が顕著にみられる。アプリケーションサーバ 4 台の場合は 1 台当たりの影響が小さくなるため、性能劣化の度合いが小さくなる。

3. コンピュータリソース

   冗長化構成となっているアプリケーションサーバにおいて、データ永続化サービスの障害発生後にデータベース接続が切り替わり、データベース接続 F/O の移動先となるアプリケーションサーバの CPU 使用率が高くなる。一方、障害の発生したデータベース接続 F/O の移動元となるアプリケーションサーバの CPU 使用率は減少する。この 2 つの CPU 使用率の平均値をとると、障害の発生していない正常な状態の CPU 使用率と同程度の値になる。

性能劣化測定の結果対比

1. アプリケーションサーバの処理状態

   データベースを一つ停止した状態でも、予想通り特にエラーなく処理を行うことができた。これは、事前に性能限界を見積もる実験（後述する（4）性能限界テスト）を行っており、最も負荷の高い実験を行った場合も問題なくデータが取得できるよう、アプリケーションの同時実行数を調整してコンピュータリソースに余裕のできる設定を行っている。このため、データベースの停止時には冗長構成しているアプリケーションサーバで倍の同時実行数となっても、余力として見積もったコンピュータリソース範囲内で継続して処理を行うことができたと考えられる。

2. 処理性能

   予想に反してデータベース停止時にシステム全体の処理性能が向上した。この結果を踏まえるならば、業務アプリケーション用に用いているデータベースサーバ側は冗長構成としていないため、もし冗長構成としたならば、依存度の高い RDB 型の処理性能は、依存度の低い KVS 型と比較して、相対的に性能が低下する可能性がある。性能に関して KVS 型の優位性が高まるという仮説を立てることができる。

3. コンピュータリソース

   障害検知後の状態で、データベース接続 F/O の移動先となるアプリケーションサーバの CPU 使用率増加、移動元となるアプリケーションサーバの CPU 使用率が減少することは予想通りの結果である。

   最も負荷の高いアプリケーションサーバ 4 台、配信対象人数 200 人で行った実験で取得した CPU 使用率をみると

   データベース接続 F/O の移動元アプリケーションサーバ：31[%]

   データベース接続 F/O の移動先アプリケーションサーバ：78[%]

   障害の影響がないアプリケーションサーバ：69[%]

   となり、(31 + 78) / 2 = 54.5[%] < 69[%]となり、予想よりも小さい値となっている。全体的に同様の傾向となっている。これは移動元のアプリケーションサーバでは、データベース全体に用いられる CPU リソースが減少しており、移動先アプリケーションサーバでは同時接続数

の増加のみに対応するための CPU リソースが増加しているため（2 台のデータベースが稼働することにはならない）、減少量よりも増加量が少なかったことが影響していると考えられる。

(4) 性能限界テスト

実験に必要なパラメータとして、

- アプリケーションの同時実行数
- 配信対象人数

実験環境において、どの程度の値まで許容可能かを確認するための見積もりを行っている。

- アプリケーションサーバ 1 台
- アプリケーションの同時実行数を 24 スレッド

上記の設定において配信対象人数の上限値を確認する実験を行った。結果として配信対象人数が 300 の場合に RDB 型も KVS 型も使用できない状態となった。
このとき発生した現象は以下の通り。

- RDB 型はアプリケーションのメモリが不足し例外が発生
- KVS 型はアプリケーションサーバの CPU が枯渇

正常にデータを取得した範囲では、配信対象人数が 200 人において最も負荷の高い状態となり、DB サーバの CPU 使用率は最大で 26%、アプリケーションサーバの CPU 使用率は最大で 87%となった。アプリケーションサーバを 4 台とした実験を考えると、アプリケーションサーバの処理負荷は、完全に均等に処理を行う前提であれば、ほぼ同程度の負荷になると予想されるが、少しでも偏りが出ると 90%の閾値を超えてしまう可能性がある。DB サーバにはアプリケーションサーバ 4 台から同時に処理を行うことになるため、負荷が最大で 4 倍になると仮定すると、26 ×4 = 104[%]となり、CPU リソースが枯渇する懸念がある。このため最も高負荷が予想される設定（アプリケーションサーバ数が 4 台、配信対象人数が 200）において、問題なくデータを取得できるよう、アプリケーションの同時実行数を 16 とした場合、この設定における CPU 使用率の試算は

16 / 24 × 104 = 69.33[%]

となり、正常にデータを取得できる見込みとなる。
同様にアプリケーションサーバの処理負荷を考えると

16 / 24 × 87 = 58 [%]

となり、1.5 倍の処理を行ったとしても 90[%]を超えることがなく、余力が出る。このため下記の設定で実験を行った。

- アプリケーションサーバ 1 台当たりの同時実行数を 16 スレッド
- 配信対象人数を最大で 200

結果として、DB サーバの CPU 使用率の最大は 42[%]、アプリケーションサーバの CPU 使用率の最大

は 80[%]となり、見積もりの範囲内に収まる結果となったため、全ての実験において正しくデータを取得することができた。

(5) 新アーキテクチャ設計時の注意点

以上の結果から、KVS を使用した新アーキテクチャは、従来の RDB を用いたアーキテクチャに対して、特にスケーラビリティに優れており、障害発生時にも問題なくシステムを継続して使用する事が可能であることが分かった。1 台のデータベースが停止しても大きく性能が落ちることなく、十分に実用可能であると考えられる。

なお、キューDB を RDB から KVS(Cassandra)に置き換えただけで実現可能となった点は下記である。

- キューDB で発生した障害を検知するまでの時間短縮
- キューDB の可用性

下記はアプリケーションと組み合わせることで実現可能な点である。

- 性能(TPS)の向上
- スケーラビリティの向上

アプリケーションからはデータベースへのアクセスと SQL の実行を行っている。またシステムに対する処理要求を複数のアプリケーションサーバ間で均等化しており、一部のアプリケーションサーバへの処理負荷の偏りを少なくすることでスケーラビリティの向上に繋がっている。

ただし、従来のアーキテクチャとは異なる仕組みとなるため、以下の点を考慮してシステムの設計、及び運用を行う必要が出てくる。

- システムのボトルネック

  正常な状態において、システムのボトルネックが DB サーバからアプリケーションサーバに移動しており、アプリケーションサーバの CPU 使用率が大きくなる。このため、システムへの処理要求数に対して、従来よりもアプリケーションサーバの CPU が十分な性能を担保できるか見積もりを行う必要がある。
- キュー用 DB 停止時の処理負荷の見積もり

  キュー用 DB が停止すると、冗長構成をとっているアプリケーションサーバに接続が移動し、処理の偏りが発生する。この状態においても問題なく処理を継続することが可能か、見積もりを行う必要がある。
- DB サーバ高負荷時の運用

  アプリケーションサーバに余力があるものの、DB サーバが高負荷となり過剰遅延が発生する場合、冗長構成を取っているアプリケーションサーバの一部を停止することで DB サーバの処理負荷を下げ、システムを正常に使用可能な状態とする運用を行うケースがある。このとき、従来であれば、どのアプリケーションサーバを選択してもシステムに与える影響は同じと考えられるが、新しいアーキテクチャの場合、KVS 間で冗長構成を取っているため、

アプリケーションサーバの処理負荷の偏りを考慮する必要がある。特に 2 台以上を停止するのであれば、冗長構成を取っているペアの両方を停止しないよう、あらかじめ、どのアプリケーションサーバを停止することが可能かをマニュアル等に記載する等、十分に対策を用意しておくことが望ましい。

6. 2 キューDB を利用する別システム（貨物データ配信システム）

次に、今回提案したアーキテクチャを、別のシステムに応用する場合を検討する。

例として宅配便配送データ管理システム(配送データ管理システム)を取り上げる。配送データ管理システムとは、宅配便の位置情報を収集し状況を把握し、各営業所の担当者やドライバ及び荷物の依頼や受け取りを行う顧客が利用するシステムである。1 日に 1 億件の貨物に関するデータが発生し、データを活用する社内ユーザ数は約 6000 人である。各荷物には「送り状番号」が設定され、インターネットから、受け取り予定の荷物または配達を依頼した荷物の位置情報の確認をする等、リアルタイムに更新される貨物の集荷、配達状況をデータとして管理するシステムである。図 6-1 に概要を記載する。

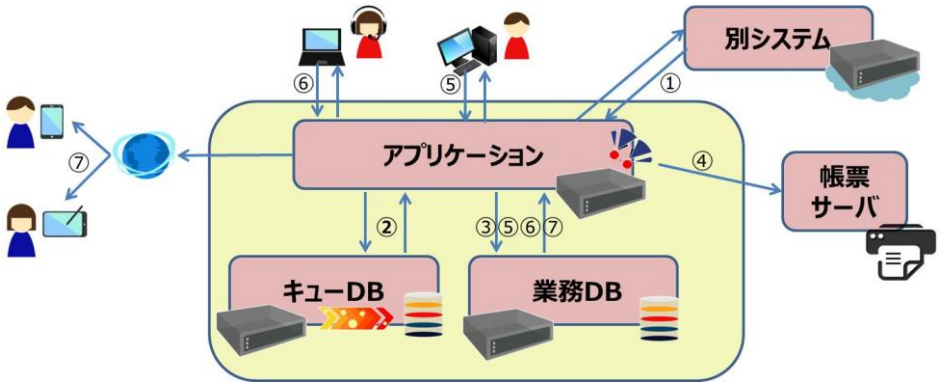


図 6-1: 宅配便配送データ管理システム

図 6-1 における、処理フローは以下となる。

① Another System(取引先、商品販売サイト、ドライバ)から貨物のデータを受信
② キューDB に受信データ格納
③ 貨物データを更新
④ 貨物伝票を電子帳票として保管
⑤ 事故や遅延等、貨物状況を更新
⑥ 貨物の問合データ参照⑦貨物の配達状況を参照

3 章でモデル化を行った株価アラート配信システムとの違いとして、特に特徴的な点を以下にあげる。

- 1 日に取り扱う要求処理数が多い。
  - ➢ システム全体として 1 日に 3000 万処理×5 [kbyte]のデータが発生。
    - ✧ 株価アラート配信システムの高負荷時と同等の負荷が常時発生している。
  - ➢ データベース 1 ノードあたり数 1000～数 10000 の接続が発生している。

➢ システム全体として 100VM, 300 プロセスが常時稼働する。

- キューDB と業務 DB が物理的に分かれている。
- 24 時間 365 日、無停止で稼働しており、定常的にデータ処理を行う。

KVS 型アーキテクチャを適用する際に、株価アラート配信システムと同じ仕組みを利用可能な点は以下である。

- キューDB
  冗長構成の仕組みはキューDB で実現可能であり、プロセス数は多いものの、正常な動作及び障害が発生した時の切替の操作は全く同じ動作を行うため、特に変更することなく利用可能である。
- アプリケーションサーバ
  アプリケーションを実行するためのフレームワークとして全く同じものを使用しているため、特に変更することなく利用可能である。

株価アラート配信システムと異なる仕組みが必要となる点は以下である。

- キュー用データの初期化
  データベースに格納されるデータ量が大きくなるため、定期的に不要なデータを削除する必要が出てくるが、システムは無停止で稼働を続けている。この条件下において、キューDB のデータ量が過剰に大きくならないよう、不要なデータを削除する仕組みが必要となる。
- 可用性実現のための冗長構成
  24 時間 365 日稼働を行うための可用性を満たすようシステムを構成する必要がある。
  さらに、接続している別システムやユーザが異なるため、システムに障害が発生した場合、相互影響がないように各アプリケーションの配置や可用性を考慮した構成を考える必要がある。
  一例として、インターネットからアクセスするユーザ向け、帳票出力、別システムとの連携と別々に異なるアプリケーション毎に冗長性を実現している。

6. 3 キューDB を必要としない、分散 DB を用いるシステム

KVS で扱うデータ特徴は、Key と Value を 1 対 1 としたシンプルな構造である。キューDB で扱うデータは、この特長に当てはまっており、アプリケーションからみた特徴は下記の通り。

- 常に一意なデータを取り扱うため、Key と Value の組み合わせで全データを表現できる。
- Key を用いた検索、Key を用いたデータの特定と更新を行う。
- テーブルの種類は一つであり、複数テーブルを組みあわせたデータ処理は行わない。

これらの特長に当てはまるデータのみを取り扱うシステムであれば、十分に KVS を用いることが可能と考えられる。
またキューDB を用いる理由の一つとして、更新処理の発生量及び発生頻度が高いシステムにおいて有効である。逆に、更新量が少ないまたは頻度が少なく、更新量が多くても同一データに対す

る更新競合が少なく、データを参照する処理の方が多いようなシステムではキューDB は特に必要とならない。
このようなキューDB を用いることなく、KVS を適用可能なシステム例を下記に記載する。

- ソーシャルコミュニケーションシステム（チャット、Blog 等）
  - テキストデータ
  - 画像、動画等のコンテンツ
- 運用管理ツール（ログ収集、ユーザ/操作履歴管理等）
  - ログファイル
  - システム操作情報
  - ユーザ権限管理

一方、下記のようにデータを取り扱う特徴がある場合、KVS は適さないと考えられる。

- 正規化されたデータモデルを中心に扱う。
- 集計処理等、複数種類のデータをグルーピングして計算処理を行う。
  - RDB 型は関数処理が用意されているが、KVS では関数が存在せず、例として要素数を数える SQL(例 **`select count(*)`**)でも応答が遅延する。
- 検索条件が複雑であり、主となる Key 以外にも様々な項目が検索条件の対象となる。
  - 結合条件があると、KVS は遅延が発生する。
  - Key に該当しない項目を検索条件とすると、KVS は遅延が発生する。

データモデルを工夫し、KVS に適したデータ構造とすることで利用しやすくする可能性はあるものの、KVS が適さないと考えられるシステム例を下記に記載する。

- DWH：Data Ware House（店舗別売上データ分析、年間発注データ分析等）
- マスタデータ関連システム（BOM、MD 等）
- 業務支援システム（需要予測、自動発注等）

それぞれ様々なデータ処理関数、解析軸を持つため、KVS を活用する利点がない。

6. 4 データベースへの依存度が低い、一般的な分散システム

データベースに依存度が低いシステムの場合、データの送受信はあるものの、データベースの機能を活用した仕組みとなっていない可能性が高い。このため、RDB ではなく KVS を用いることができる可能性も高くなると考えられる。
KVS を用いることができる可能性のあるシステム例を以下にあげる。

- P2P（ファイル位置情報管理システム、ユーザ情報管理システム）

データベースへの依存度が低い、一般的な分散システムにおいても、KVS を有効に活用できる可能性があると考えられる。

第 7 章 おわりに

非同期でキューイング処理を行うシステムにおいて、RDB を用いたアーキテクチャに対して懸念されている問題を解決するため、KVS を用いたアーキテクチャが有効となるという予想を立てた。実際に実験を行うことで、実際に利用するにあたり十分な耐障害性を持ち合わせており、データベース停止時にも大きな性能劣化が発生することなく、RDB 型よりも性能が向上し、スケーラビリティに優れていることを確認することができた。
今後のシステム構築において、KVS を用いたアーキテクチャを利用する見込みが立った。RDB 型の抱える問題点を払拭しつつ、データベースの障害発生時にも継続してサービスが提供でき、ユーザ数やデータ量に対し、より柔軟にコンピュータリソースを活用することが可能となる。

本論文では Cassandra を用いて KVS 型アーキテクチャを実現したが、もし他の KVS を用いて実現する場合は、下記を考慮する必要がある。

- Oracle NoSQL
  - 可用性（マスタノードの冗長構成）

    マスタノードが停止した場合に備えた冗長構成は、他の参照のみしか行わないノードよりも重点的に考慮したシステム構築が必要になる。
  - スケーラビリティ

    マスタノードはスケーラビリティに欠けており、更新処理が多いシステムの場合、負荷が集中しボトルネックになる可能性があるため、将来のシステム拡張やユーザ数の増加も見込んだ上で、十分な性能を持つ HW を使用する必要がある。
- HBase
  - 可用性（Zookeeper, Master）

    Zookeeper 及び Master は、停止してしまうとシステムを利用できなくなってしまう。このため他の Region とは別に冗長構成を考慮したシステム構築が必要になる。
  - スケーラビリティ

    Zookeeper 及び Master はスケーラビリティに欠けており、システムに対する要求数が増加した場合、負荷が集中しボトルネックになる可能性がある。将来のシステム拡張やユーザ数の増加も見込んだ上で、十分な性能を持つ HW を使用する必要がある。

今後の課題としては以下があげられる。

- データベース接続に関する耐障害性への影響調査

  プライマリまたはセカンダリとなるデータベースを明確に設定して接続先を固定している状態と考えていたが、耐障害性実験のログから、プライマリまたはセカンダリをランダムに選択され、データベースの接続先を意図したとおりに接続できていないことが判明した。この現象は、意図通りであればエラーを検知しないアプリケーションサーバであるにもかかわらず、データベースの

障害発生後にエラーを検知するログを出力していたことから確認できた。この原因は JDBC ドライバの設定や動作仕様によるものと考えられる。このため、アプリケーションで使用する JDBC ドライバの調査を行う必要がある。

- 障害復旧時の操作に関する調査
  障害から復旧するためにデータベース用のプロセスではなく OS の再起動を行っている。プロセス単位で復旧を行うよりも、復旧作業に要する時間が大きくなる（数 10 秒→数分程度）。加えて、同一の OS 上で別のアプリケーションがあった場合や OS の監視の仕組みでエラーを検知してしまう可能性があることから、データベース用のプロセスのみを復旧できることが望ましい。OS からの再起動を行う場合、上記の影響も考慮した運用が必要となる
- システムの可用性
  性能劣化測定の実験で、データベース 1 台を停止した状態で動作可能なことを確認しているが、「3. 4 モデル化」で示した通り、株価配信アラートシステムがデータの提供元からデータを受信し、ユーザに配信するメインとなる機能のみを対象としている。実際の可用性を考えるならば、システムの全てのアプリケーションの実行や運用で用いる機能が全て実行可能であることを網羅的に確認する必要がある
- アプリケーション機能とパラメータの違いが性能に与える影響
  4 章で行った実験と、実際のシステム稼働を比較した場合、アプリケーションから実行する SQL の回数は同程度のため、さほどの影響はない。しかし、下記の相違点に関して影響が出てくると考えられる
  - システムからデータを配信する端末数
  - ユーザが事前に登録する配信データ種類の設定

  データを配信する処理単位とアプリケーションの処理フローやロジックが変わることになり、1つのトランザクション内での計算量が増加して性能が劣化する可能性がある。性能要件を満たすための HW を用意し、現行の RDB 型を用いたシステム開発でも行っているように、KVS 型でも同様にテストと確認を実施する必要がある
- 他のシステムで KVS 型を用いる場合の考慮点
  6.2, 6.3, 6.4 で取り上げているように、他のシステムで KVS 型アーキテクチャまたは KVS を用いる場合、十分にデータモデルを吟味し、システムの特性を考慮したうえで設計を行い、機能テストや性能テストを含めた動作確認を行うことが必要となる。

謝辞

本論文の作成にあたり、終始熱心なご指導を頂いたソフトウェア科学領域の鈴木准教授に深く感謝いたします。実験環境を用意していただいた情報社会基盤研究センターの皆様、様々な質問にも回答していただき、ありがとうございました。

参考文献

[1] アプリケーションの安定稼動を実現するシステム基盤の統合ノウハウ (著) 谷口 俊一、 飯田 博記、 石川 辰雄、櫻井 義晴

[2] データベース設計 構築 基礎+実践マスターテキスト (著) 弓場 秀樹、武田 喜美子

[3] ニコニコ大百科(タイムシフト)

http://dic.nicovideo.jp/a/%E3%82%BF%E3%82%A4%E3%83%A0%E3%82%B7%E3%83%95%E3%83%88%E6%A9%9F%E8%83%BD

[4] ネットワークはなぜつながるのか (著)戸根 勤

[5] マスタリング TCP/IP 入門編 (著)竹下 隆史、村山 公保、荒井 透、苅田 幸雄

[6] Apache Cassandra

http://cassandra.apache.org/

[7] Cassandra: The Definitive Guide By Eben Hewitt, Publisher: O'Reilly Media

[8] DataStax 社認定トレーニング「Cassandra の技術概念と基礎、ツール」 配布資料

[9] DATASTAX ACADEMY

https://academy.datastax.com/

[10] HBase: The Definitive Guide By Lars George, Publisher: O'Reilly Media

[11] High-availability cluster

https://en.wikipedia.org/wiki/High-availability_cluster

[12] Managing & Using MySQL By Tim King, George Reese, Randy Yarger, Hugh Williams

[13] MySQL Replication Tutorial - O'Reilly Media

http://assets.en.oreilly.com/1/event/2/MySQL%20Replication%20Tutorial%20Presentation%202.pdf

[14] Oracle NoSQL Database

http://www.oracle.com/technetwork/database/database-technologies/nosqldb/overview/index.html

[14] Oracle RAC の概要

https://docs.oracle.com/cd/E16338_01/rac.112/b56290/admcon.htm

[15] Queueing Systems: Problems and Solutions By Leonard Kleinrock, Richard Gail

[16] SE のための Oracle チューニングハンドブック (著)後藤 孝憲、名和 満、五嶋 和彦

[17] WikiPedia (Apache HBase)

https://en.wikipedia.org/wiki/Apache_HBase

[18] WikiPedia(BitTorrent)

https://en.wikipedia.org/wiki/BitTorrent

[19] WikiPedia(Distributed computing)

https://en.wikipedia.org/wiki/Distributed_computing
[20] WikiPedia(Distributed database)
https://en.wikipedia.org/wiki/Distributed_database
[21] WikiPedia(HTTP)
https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol
[22] WikiPedia(Network File System)
https://en.wikipedia.org/wiki/Network_File_System
[23] WikiPedia(NoSQL)
https://en.wikipedia.org/wiki/NoSQL
[24] WikiPedia(Message queue)
https://en.wikipedia.org/wiki/Message_queue
[25] WikiPedia(Multitier architecture)
https://en.wikipedia.org/wiki/Distributed_computing
[26] WikiPedia(Peer-to-peer)
https://en.wikipedia.org/wiki/Peer-to-peer
[27] WikiPedia(Remote procedure call)
https://en.wikipedia.org/wiki/Remote_procedure_call
[28] WikiPedia(TCP/IP)
https://en.wikipedia.org/wiki/Transmission_Control_Protocol
[29] World Community Grid
https://www.worldcommunitygrid.org/discover.action

URL は 2017 年 7 月 18 日　現在

付録 A

A1_正常時の性能測定データ.pdf

A2_耐障害性実験データ.pdf

A3_性能劣化測定データ.pdf

付録 B

B1_障害検知時のログ.pdf

付録 C

C1_追加実験データ.pdf

付録A

A1_正常時の性能測定データ

正常時の性能測定データー覧

アプリサーバ1台

配信対象人数	TPS（最高到達点）		TPS（平均）		CPU使用率（最大）[%]			
					アプリケーションサーバ		データベースサーバ	
	RDB型	KVS型	RDB型	KVS型	RDB型	KVS型	RDB型	KVS型
10	882	1244	583	925	16	34	15	9
50	1092	1323	797	908	20	50	18	10
100	1222	1328	913	791	22	51	20	10
150	1232	1295	928	894	23	52	21	10
200	1244	1371	927	888	24	54	21	11

アプリサーバ2台

配信対象人数	TPS（最高到達点）		TPS（平均）		CPU使用率（最大）[%]			
					アプリケーションサーバ		データベースサーバ	
	RDB型	KVS型	RDB型	KVS型	RDB型	KVS型	RDB型	KVS型
10	1859	1810	1358	1304	16	56	28	10
50	1977	1989	1370	1330	21	59	29	13
100	2170	2034	1545	1417	21	61	28	14
150	2081	2087	1490	1452	21	62	31	13
200	2124	2161	1473	1474	21	62	31	13

アプリサーバ3台

配信対象人数	TPS（最高到達点）		TPS（平均）		CPU使用率（最大）[%]			
					アプリケーションサーバ		データベースサーバ	
	RDB型	KVS型	RDB型	KVS型	RDB型	KVS型	RDB型	KVS型
10	2582	2715	1893	2021	19	57	31	17
50	2680	3091	1739	2041	19	59	34	19
100	2722	3035	1824	2105	19	64	35	20
150	2770	2982	1803	2073	20	63	35	20
200	2771	3008	1627	2080	21	63	36	20

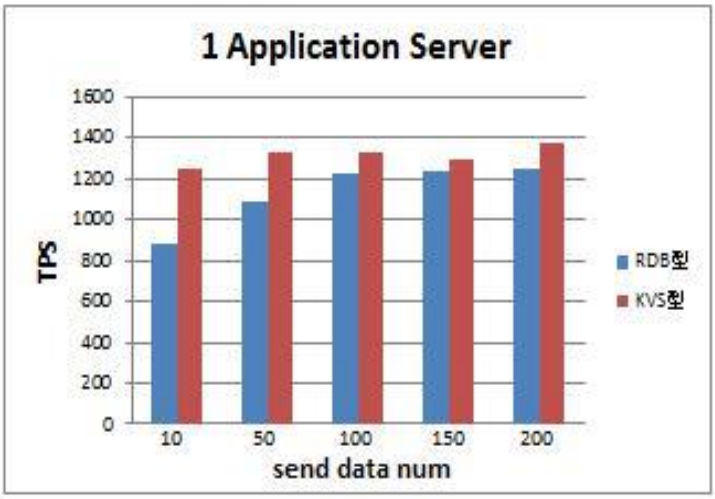
アプリサーバ4台

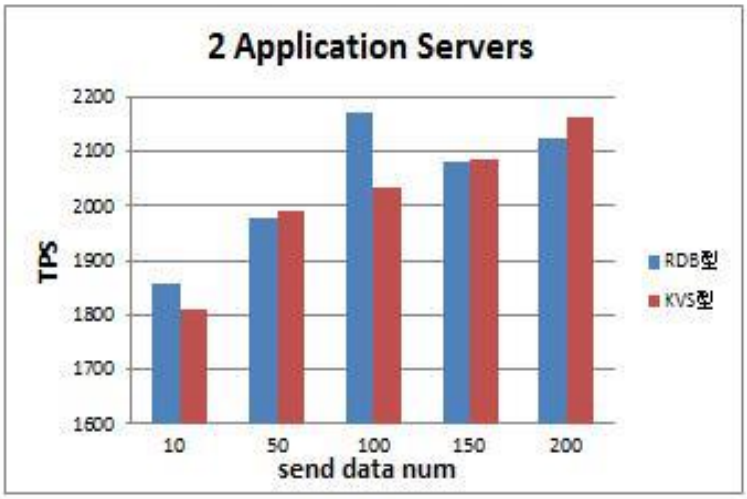
配信対象人数	TPS（最高到達点）		TPS（平均）		CPU使用率（最大）[%]			
					アプリケーションサーバ		データベースサーバ	
	RDB型	KVS型	RDB型	KVS型	RDB型	KVS型	RDB型	KVS型
10	3252	3395	2223	2617	19	60	38	20
50	3122	3733	2193	2783	18	66	42	24
100	3252	3834	2092	2742	18	64	40	25
150	3073	3806	2043	2631	18	65	41	25
200	3055	3902	2015	2607	18	65	41	24

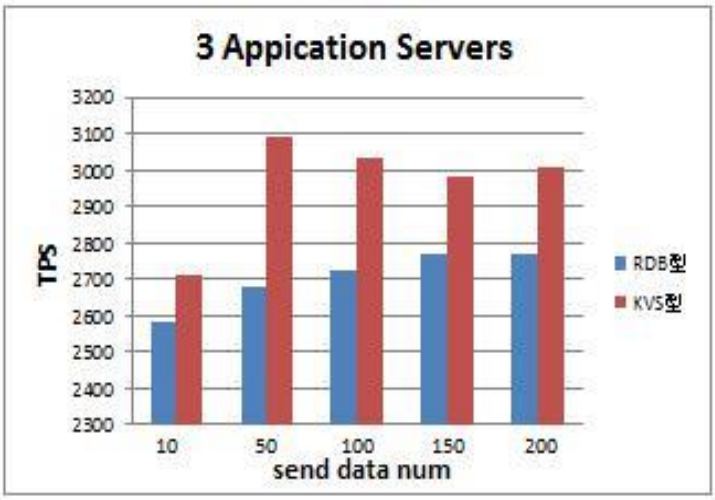
CPUの各平均は、最大-3～5%程度

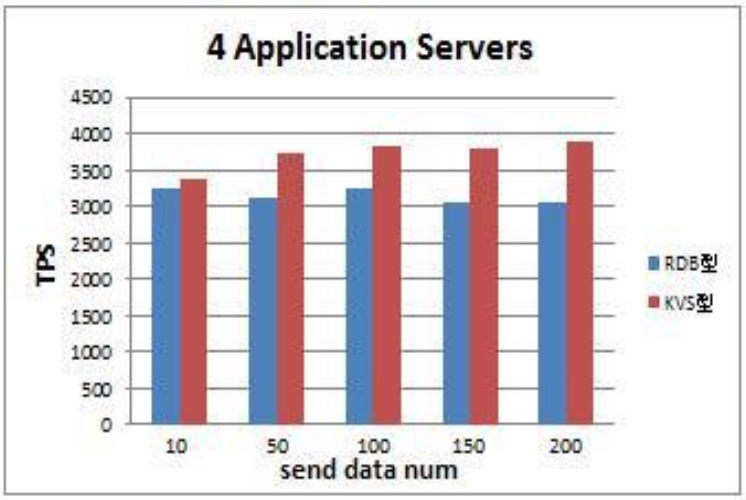
TPS（最大値）

アプリケーションサーバ数	配信対象人数					型
	10	50	100	150	200	
4台	3252	3122	3252	3073	3055	RDB
	3395	3733	3834	3806	3902	KVS
3台	2582	2680	2722	2770	2771	RDB
	2715	3091	3035	2982	3008	KVS
2台	1859	1977	2170	2081	2124	RDB
	1810	1989	2034	2087	2161	KVS
1台	882	1092	1222	1232	1244	RDB
	1244	1323	1328	1295	1371	KVS



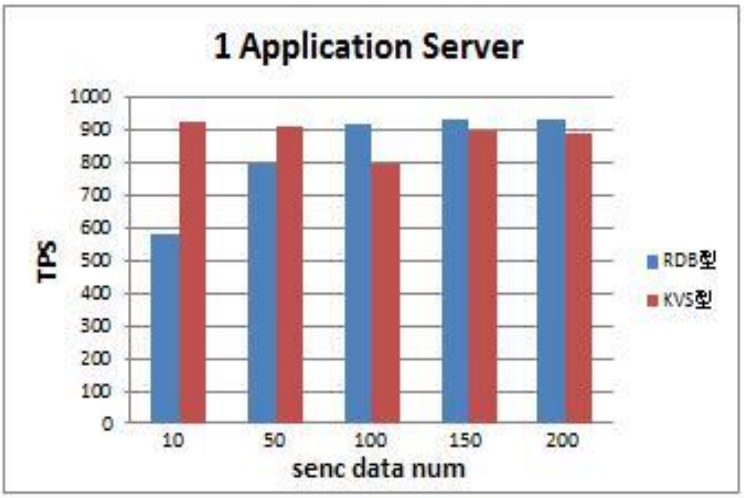


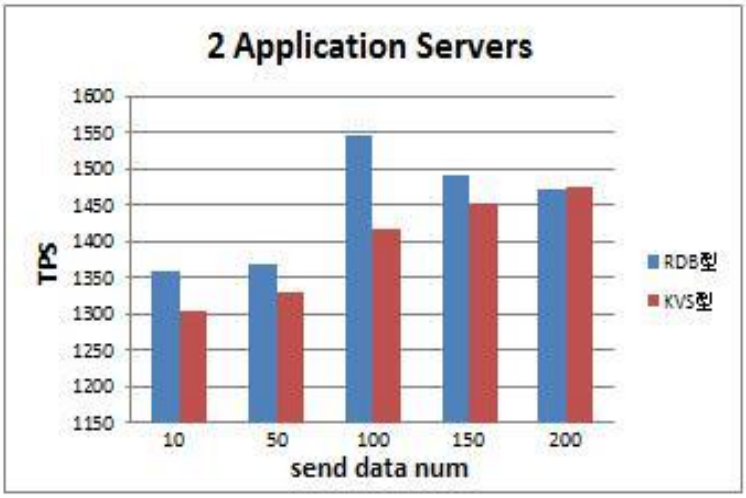


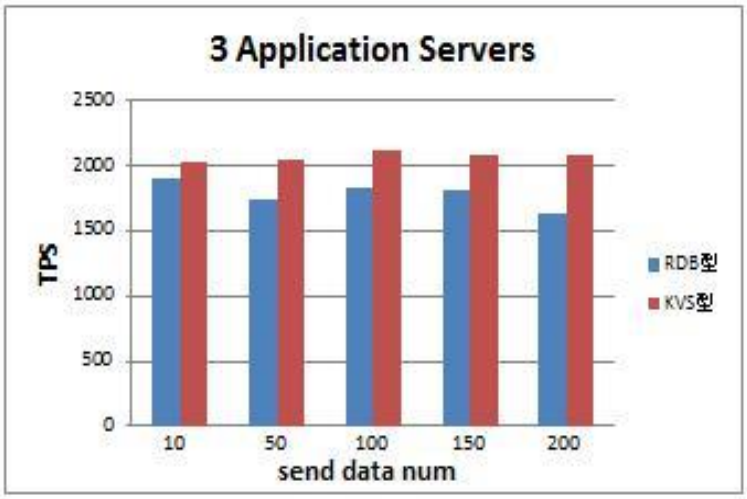


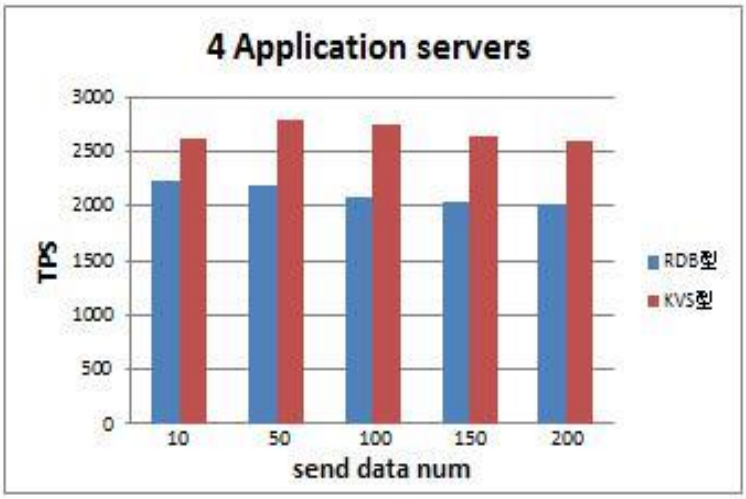
TPS（平均値）

アプリケーションサーバ数	配信対象人数					型
	10	50	100	150	200	
4台	2223	2193	2092	2043	2015	RDB
	2617	2783	2742	2631	2607	KVS
3台	1893	1739	1824	1803	1627	RDB
	2027	2041	2105	2073	2080	KVS
2台	1358	1370	1545	1490	1473	RDB
	1304	1330	1417	1452	1474	KVS
1台	583	797	913	928	927	RDB
	925	908	791	894	888	KVS



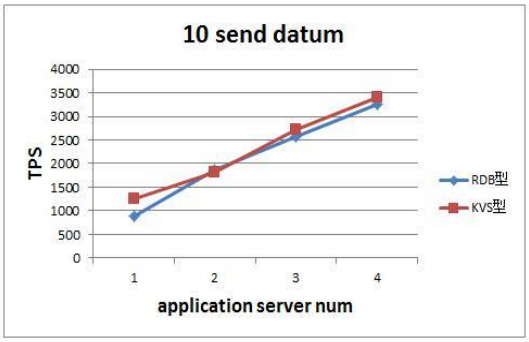


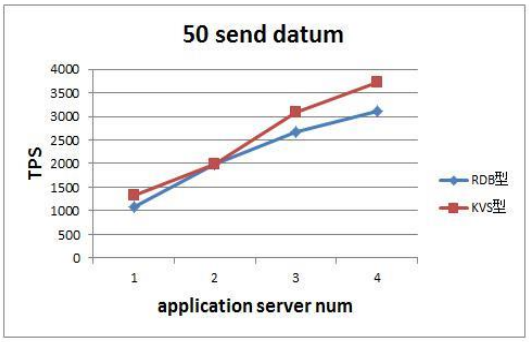


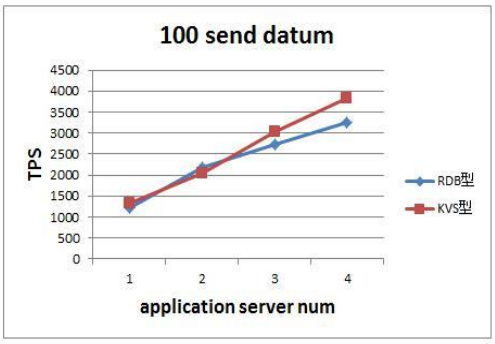


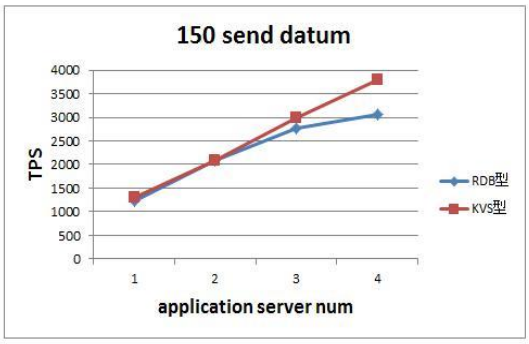
TPSスケーラビリティ（最大値）

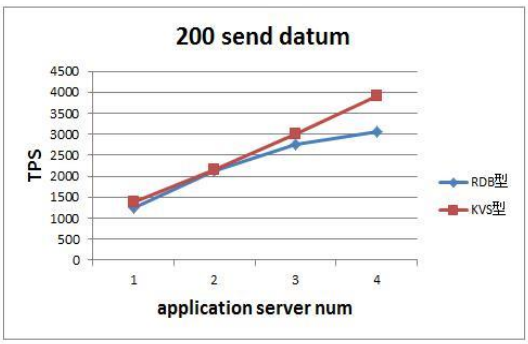
配信対象人数	アプリケーションサーバ数				型
	1	2	3	4	
200	1244	2124	2771	3055	RDB
	1371	2161	3008	3902	KVS
150	1232	2081	2770	3073	RDB
	1295	2087	2982	3806	KVS
100	1222	2170	2722	3252	RDB
	1328	2034	3035	3834	KVS
50	1092	1977	2680	3122	RDB
	1323	1989	3091	3733	KVS
10	882	1859	2582	3252	RDB
	1244	1810	2715	3395	KVS





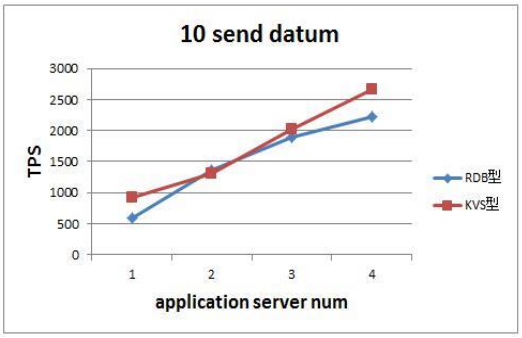


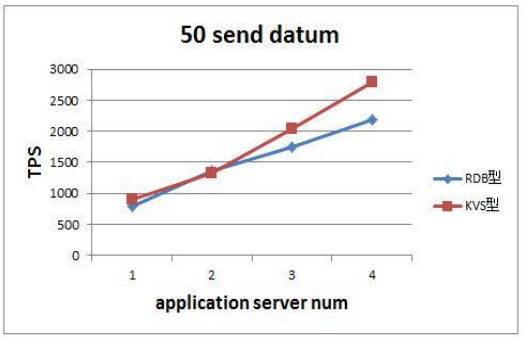


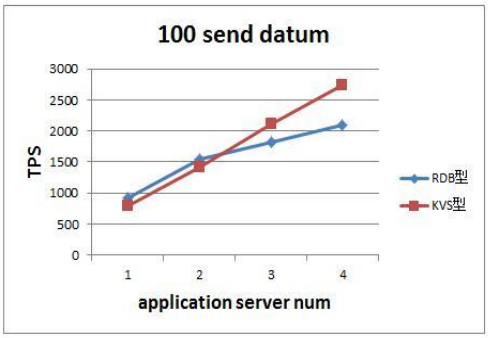


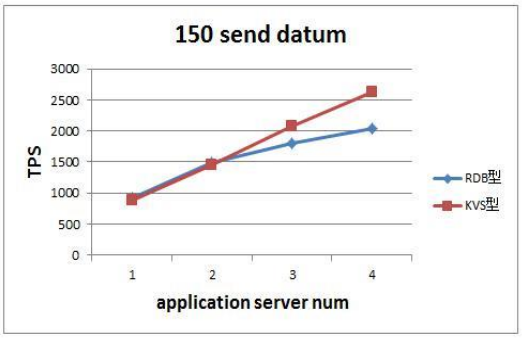
TPSスケーラビリティ（平均値）

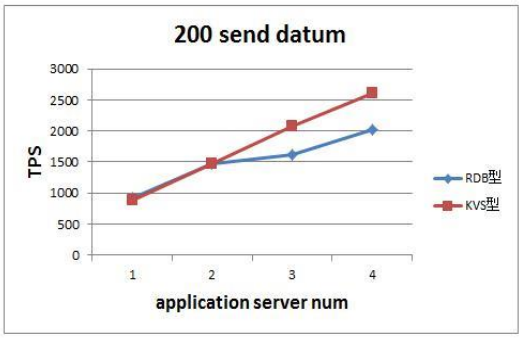
配信対象人数	アプリケーションサーバ数				型
	1	2	3	4	
200	927	1473	1627	2015	RDB
	888	1474	2080	2607	KVS
150	928	1490	1803	2043	RDB
	894	1452	2073	2631	KVS
100	913	1545	1824	2092	RDB
	791	1417	2105	2742	KVS
50	797	1370	1739	2193	RDB
	908	1330	2041	2783	KVS
10	583	1358	1893	2223	RDB
	925	1304	2027	2671	KVS





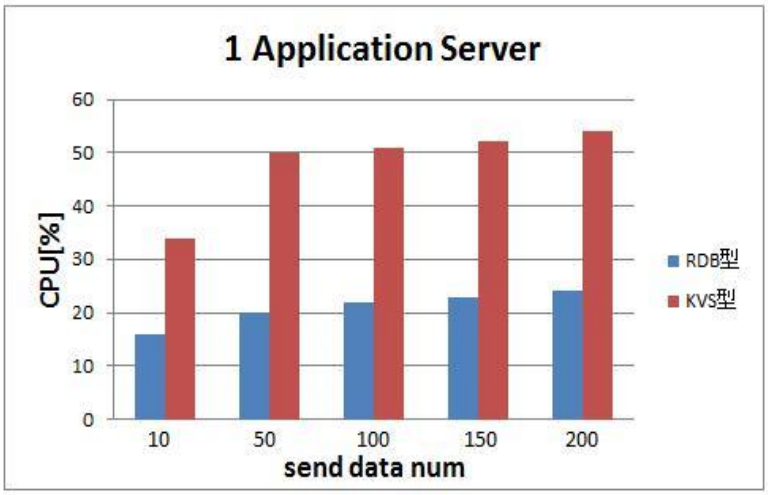


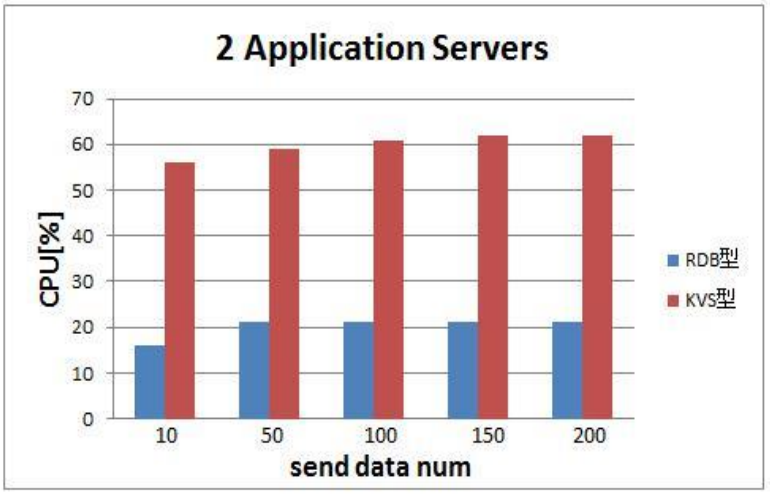


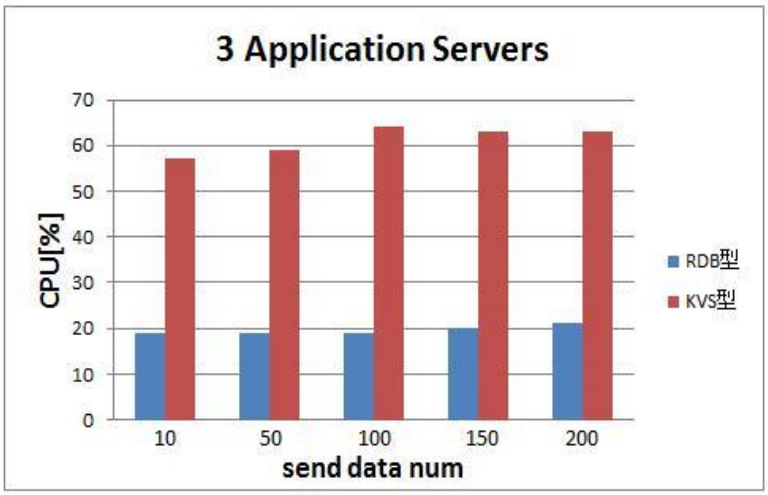


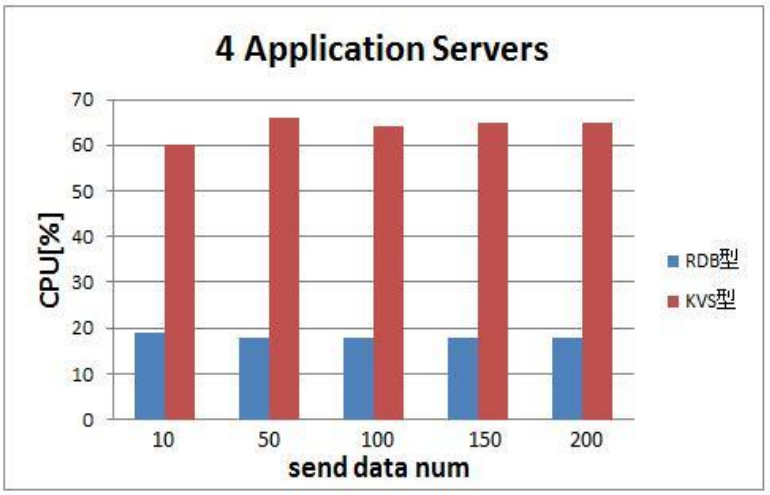
CPU（アプリケーションサーバ）

アプリケーションサーバ数	配信対象人数					型
	10	50	100	150	200	
4台	19	18	18	18	18	RDB
	60	66	64	65	65	KVS
3台	19	19	19	20	21	RDB
	57	59	64	63	63	KVS
2台	16	21	21	21	21	RDB
	56	59	61	62	62	KVS
1台	16	20	22	23	24	RDB
	34	50	51	52	54	KVS



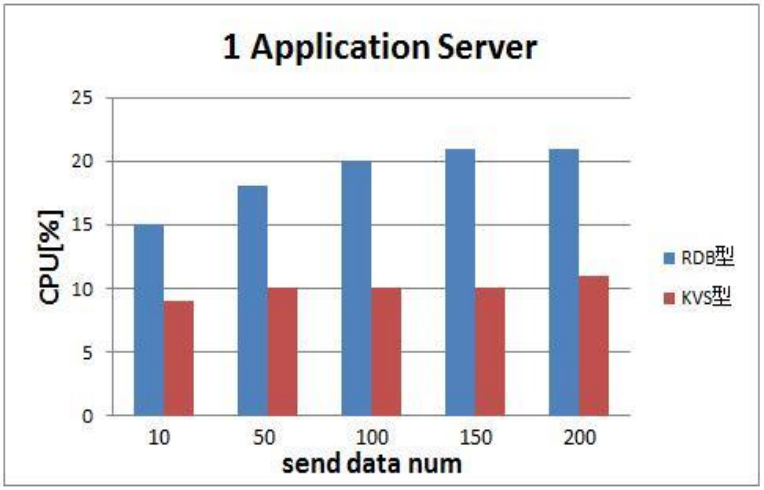


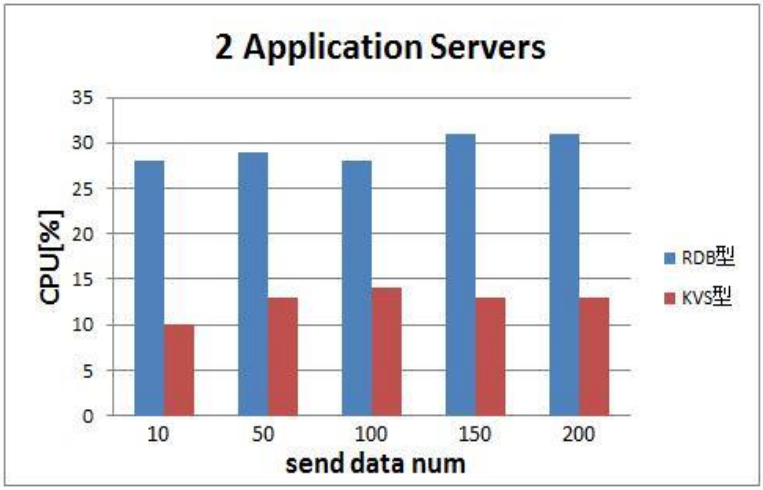


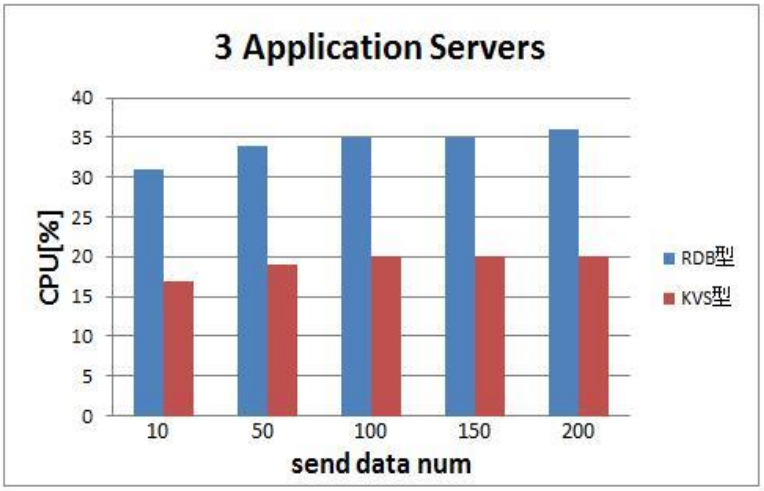


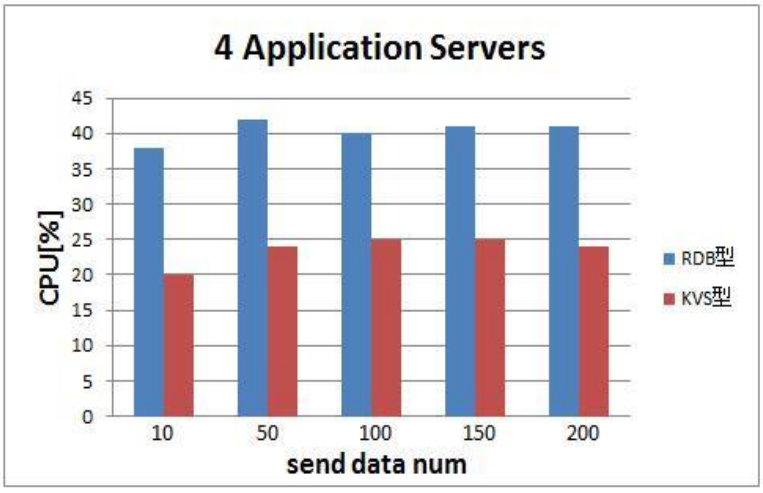
CPU（データベースサーバ）

アプリケーションサーバ数	配信対象人数					型
	10	50	100	150	200	
4台	38	42	40	41	41	RDB
	20	24	25	25	24	KVS
3台	31	34	35	35	36	RDB
	17	19	20	20	20	KVS
2台	28	29	28	31	31	RDB
	10	13	14	13	13	KVS
1台	15	18	20	21	21	RDB
	9	10	10	10	11	KVS



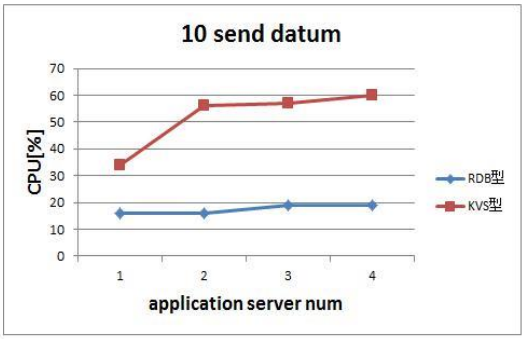


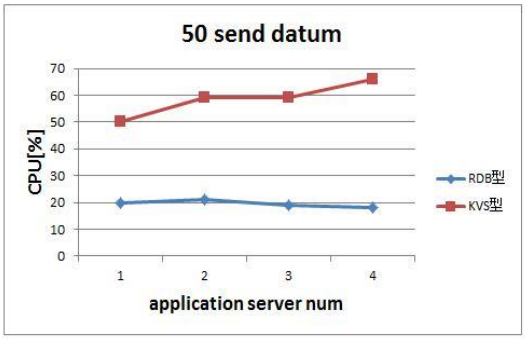


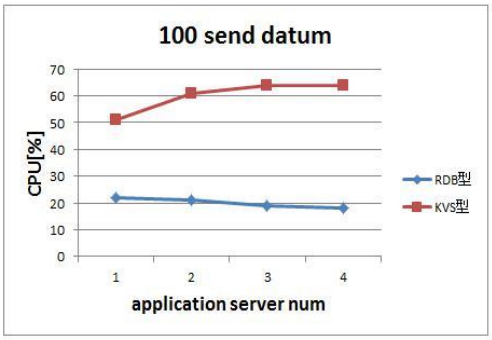


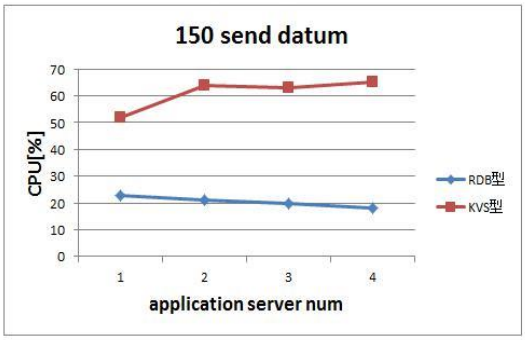
CPUスケーラビリティ（アプリケーションサーバ）

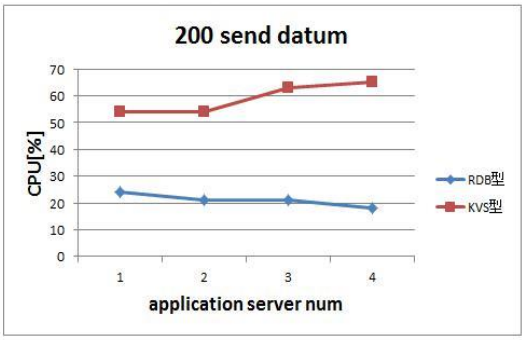
配信対象人数	アプリケーションサーバ数				型
	1	2	3	4	
200	24	21	21	18	RDB
	54	54	63	65	KVS
150	23	21	20	18	RDB
	52	64	63	65	KVS
100	22	21	19	18	RDB
	51	61	64	64	KVS
50	20	21	19	18	RDB
	50	59	59	66	KVS
10	16	16	19	19	RDB
	34	56	57	60	KVS





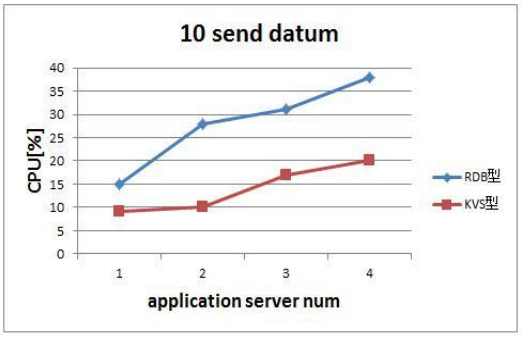


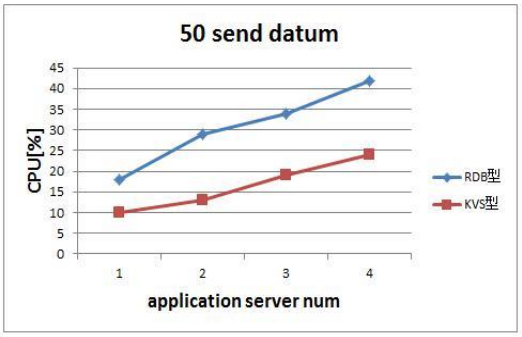


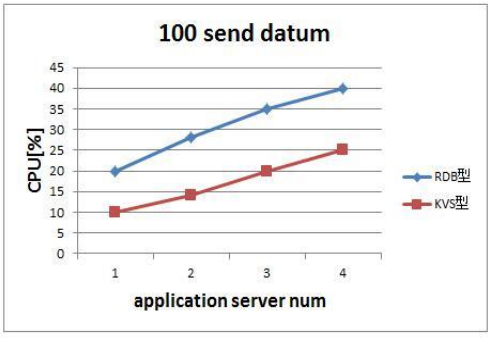


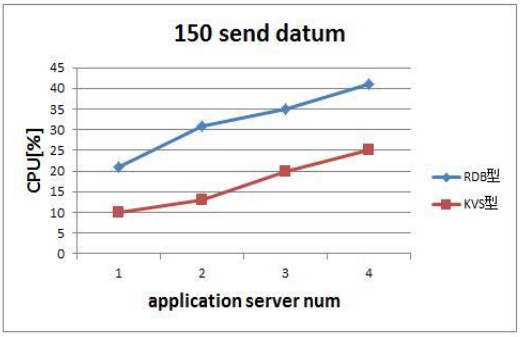
CPUスケーラビリティ（データベースサーバ）

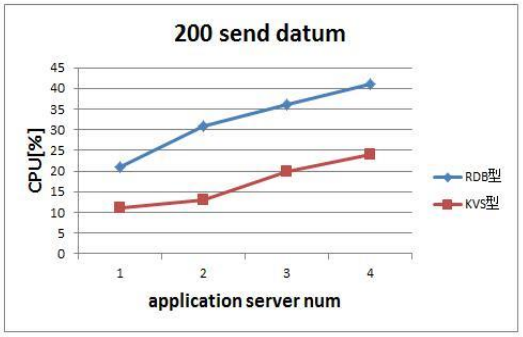
配信対象人数	アプリケーションサーバ数				型
	1	2	3	4	
200	21	31	36	41	RDB
	11	13	20	24	KVS
150	21	31	35	41	RDB
	10	13	20	25	KVS
100	20	28	35	40	RDB
	10	14	20	25	KVS
50	18	29	34	42	RDB
	10	13	19	24	KVS
10	15	28	31	38	RDB
	9	10	17	20	KVS











性能限界テスト結果

配信対象人数	TPS（最高到達点）		CPU使用率（最大）[%]			
			アプリケーションサーバ		データベースサーバ	
	RDB型	KVS型	RDB型	KVS型	RDB型	KVS型
10	1500	1940	25	62	17	10
50	1800	1930	28	68	23	13
100	1860	1960	29	71	25	13
200	1904	1980	31	87	26	13
300	計測不能 RDB型：Applicationのメモリ不足 KVS型：Application ServerのCPU枯渇					
500						
1000						

アプリケーションサーバ1台で実施
アプリケーションサーバの同時実行数を24スレッドとした場合のデータ
KVS型は配信対象人数に対して、ほぼ線型のCPU使用率の増加がみられた

アプリケーションサーバ4台で問題なくデータが取得できるように調整し、

アプリケーションサーバの同時実行数は16として、全体のデータ取得を行った

KVS型の方が限界が見極めやすい
RDB型の方がメモリ情報に影響があるため、アプリケーション依存度が高く、
見積もりが難しい可能性がある

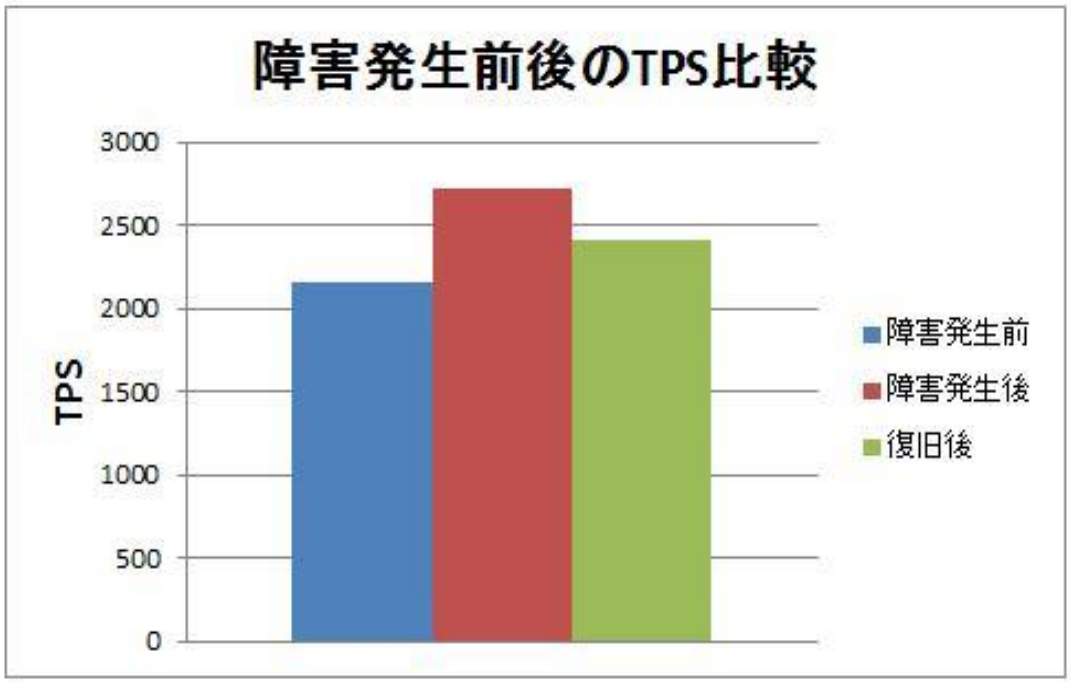
付録A

A2_耐障害性実験データ

TPS（最大値）比較

アプリケーションサーバ2台、200人送信

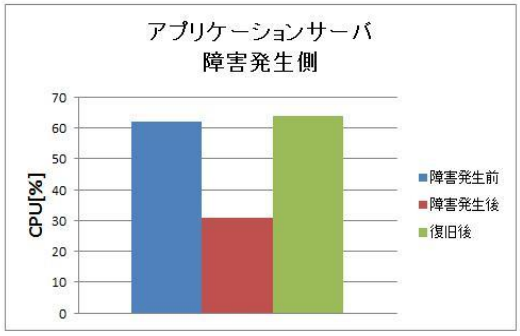
確認 タイミング	TPS
障害発生前	2161
障害発生後	2713
復旧後	2409

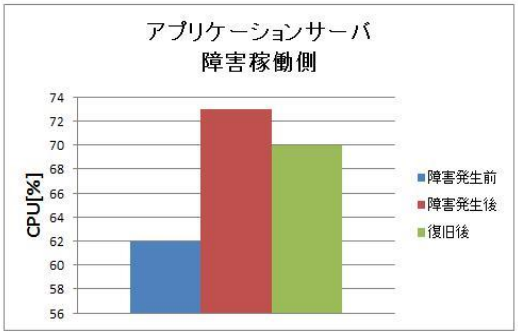


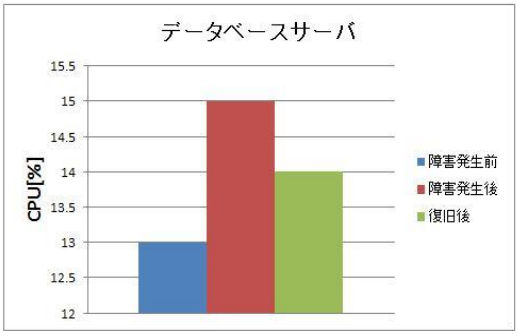
CPU比較

アプリケーションサーバ2台、200人送信

サーバ	TPS	確認タイミング
アプリケーションサーバ移動元	62	障害発生前
	31	障害発生後
	64	復旧後
アプリケーションサーバ移動先	62	障害発生前
	73	障害発生後
	70	復旧後
データベースサーバ	13	障害発生前
	15	障害発生後
	14	復旧後



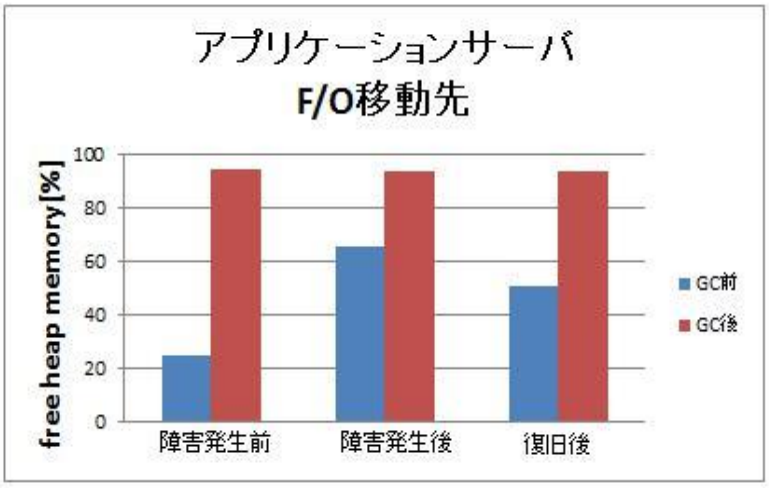


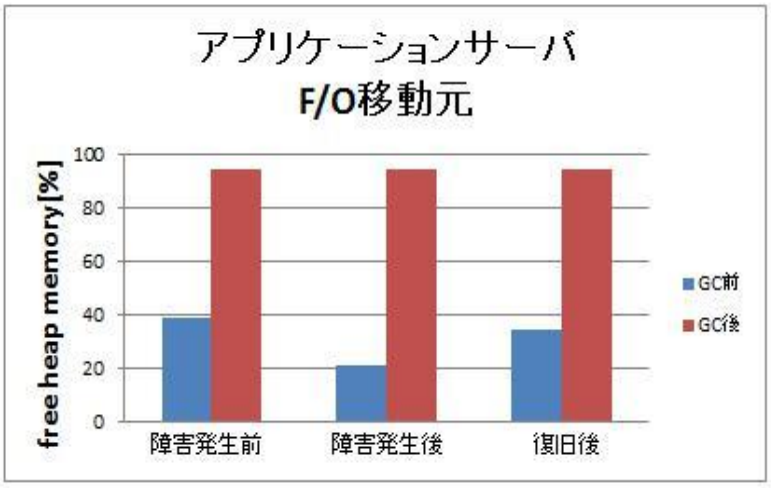


ヒープメモリ使用量比較

アプリケーションサーバ2台、200人送信

サーバ	障害発生前	障害発生後	復旧後	確認タイミング
アプリケーションサーバF/O移動元	39.19	20.68	34.71	GC前
	94.46	94.47	94.84	GC後
アプリケーションサーバF/O移動先	24.83	66.00	50.90	GC前
	94.97	93.80	93.77	GC後





付録A

A3_性能劣化測定データ

性能劣化測定データー覧

アプリサーバ1台

配信対象人数	TPS 最高到達点		TPS 平均		CPU使用率（最大）[%]						メモリ（ヒープ空率[%]）					
					アプリケーションサーバ				データベースサーバ		GC前			GC後		
	正常型	障害型	正常型	障害型	正常型	移動元障害型	移動元障害型	移動無障害型	正常型	障害型	移動元障害型	移動元障害型	移動無障害型	移動元障害型	移動元障害型	移動無障害型
10	nop															
50																
100																
150																
200																

アプリサーバ2台

配信対象人数	TPS 最高到達点		TPS （平均）		CPU使用率（最大）[%]						メモリ（ヒープ空率[%]）					
					アプリケーションサーバ				データベースサーバ		GC前			GC後		
	正常型	障害型	正常型	障害型	正常型	移動元障害型	移動元障害型	移動無障害型	正常型	障害型	移動元障害型	移動元障害型	移動無障害型	移動元障害型	移動元障害型	移動無障害型
10	1810	2427	1304	1910	56	31	67	nop	10	14	68.2	87.9	nop	94.4	94.5	nop
50	1989	2529	1330	1756	59	35	73		13	16	56.7	22.7		94.6	94.5	
100	2034	2512	1417	1731	61	35	77		14	16	49.6	53.1		94.7	95	
150	2087	2533	1452	1775	62	31	76		13	16	55.1	56.1		94.5	94.4	
200	2161	2603	1474	1733	62	30	76		13	16	14.8	28.3		94.5	94.5	

アプリサーバ3台

配信対象人数	TPS 最高到達点		TPS 平均		CPU使用率（最大）[%]						メモリ（ヒープ空率[%]）					
					アプリケーションサーバ				データベースサーバ		GC前			GC後		
	正常型	障害型	正常型	障害型	正常型	移動元障害型	移動元障害型	移動無障害型	正常型	障害型	移動元障害型	移動元障害型	移動無障害型	移動元障害型	移動元障害型	移動無障害型
10	2715	2730	2021	2005	57	24	64	48	17	16	64.5	69.1	77.3	94.7	94.5	95.2
50	3091	3165	2041	2087	59	29	70	57	19	19	79.2	18.1	86.7	94.6	94.5	95.1
100	3035	3352	2105	2235	64	30	74	58	20	22	70.4	85.2	66.4	94.5	94.5	94.9
150	2982	3236	2073	2203	63	29	77	59	20	22	49.1	72.5	81.1	94.5	94.5	95.1
200	3008	3533	2080	2456	63	30	80	60	20	22	50.9	74.6	39.2	94.4	94.5	94.9

アプリサーバ4台

配信対象人数	TPS 最高到達点		TPS （平均）		CPU使用率（最大）[%]						メモリ（ヒープ空率[%]）					
					アプリケーションサーバ				データベースサーバ		GC前			GC後		
	正常型	障害型	正常型	障害型	正常型	移動元障害型	移動元障害型	移動無障害型	正常型	障害型	移動元障害型	移動元障害型	移動無障害型	移動元障害型	移動元障害型	移動無障害型
10	3395	4141	2617	3164	60	30	73	67	20	25	74	72	55.9	94.6	94.5	95.1
50	3733	4201	2783	2621	66	31	77	68	24	28	8.17	35.6	5.95	94.6	94.6	94.8
100	3834	4332	2742	2770	64	32	78	71	25	27	38.4	58.4	45.9	94.6	94.4	95.1
150	3806	4455	2631	3000	65	32	78	70	25	25	30.6	46.1	55.2	94.6	94.4	94.9
200	3902	4398	2607	2874	65	31	78	69	24	25	40.9	39.5	39.5	94.6	94.5	94.7

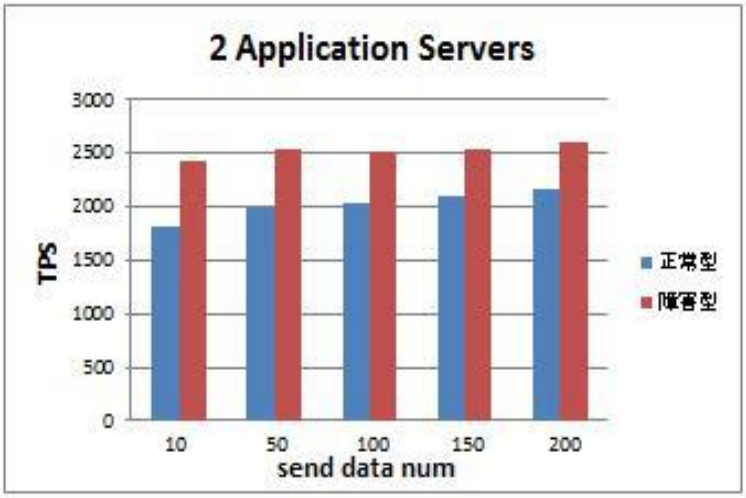
CPUの各平均は、最大-3～5%程度

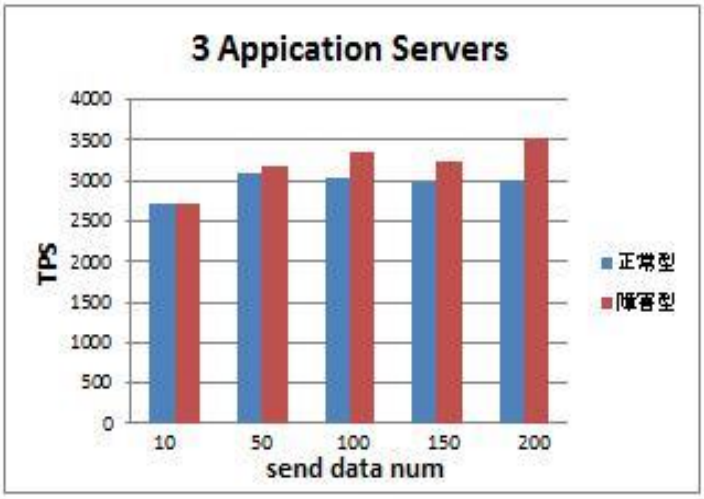
正常型：使用しているCassandraノードは全て正常に稼働

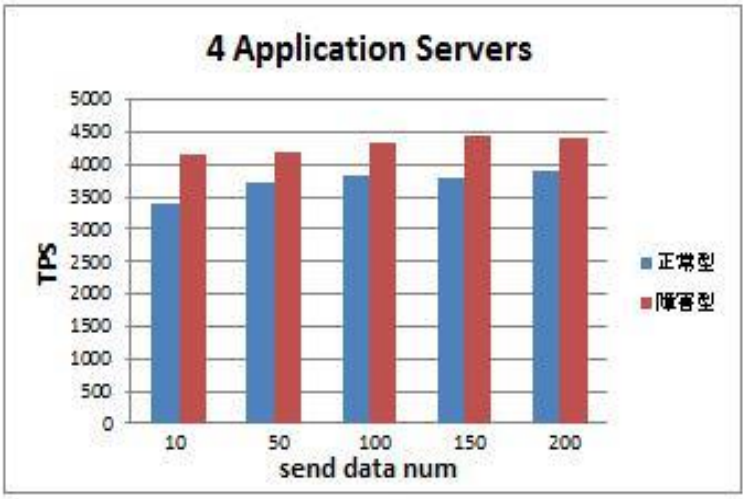
障害型：使用しているCassandraノードのうち、1台が停止しての稼働

TPS（最大値）

アプリケーション サーバ数	配信対象人数					型
	10	50	100	150	200	
4台	3395	3733	3834	3806	3902	正常型
	4141	4201	4332	4455	4398	障害型
3台	2715	3091	3035	2982	3008	正常型
	2730	3165	3352	3236	3533	障害型
2台	1810	1989	2034	2087	2161	正常型
	2427	2529	2512	2533	2603	障害型
1台	nop					正常型
						障害型

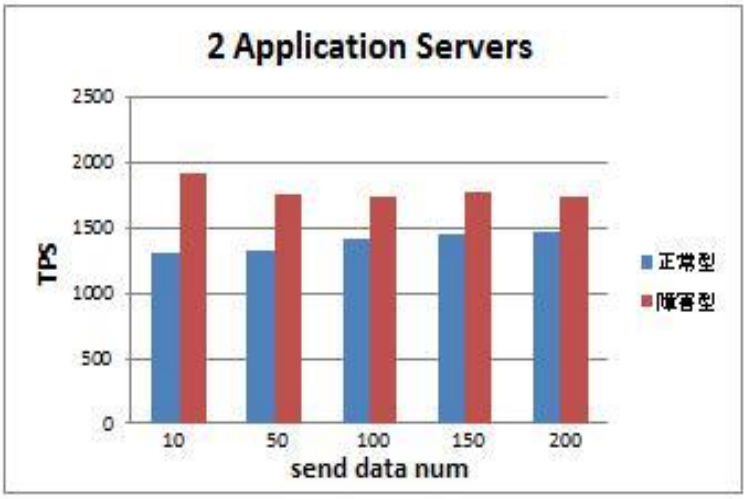


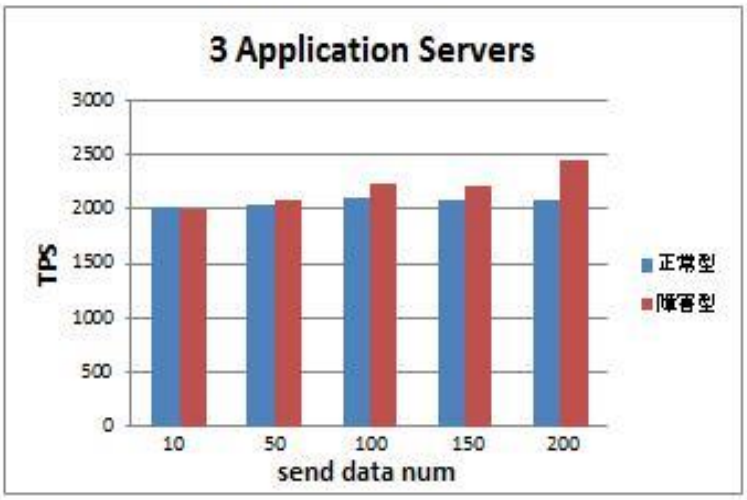


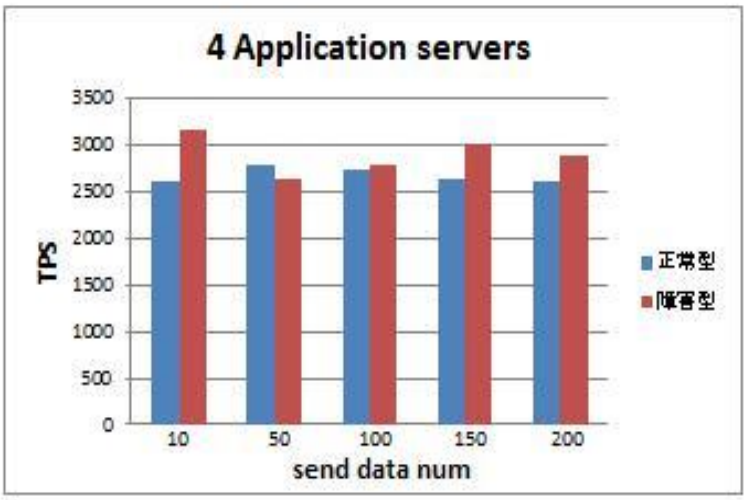


TPS（平均値）

アプリケーション サーバ数	配信対象人数					型
	10	50	100	150	200	
4台	2617	2783	2742	2631	2607	正常型
	3164	2621	2770	3000	2874	障害型
3台	2027	2041	2105	2073	2080	正常型
	2005	2087	2235	2203	2456	障害型
2台	1304	1330	1417	1452	1474	正常型
	1910	1756	1731	1775	1733	障害型
1台	nop					正常型
						障害型

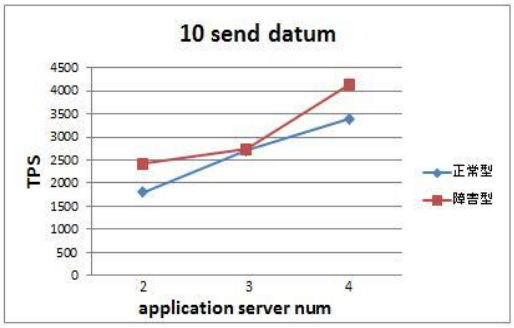


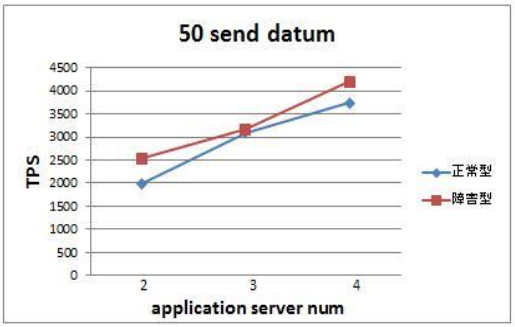


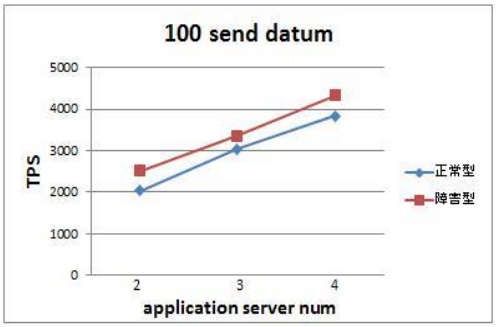


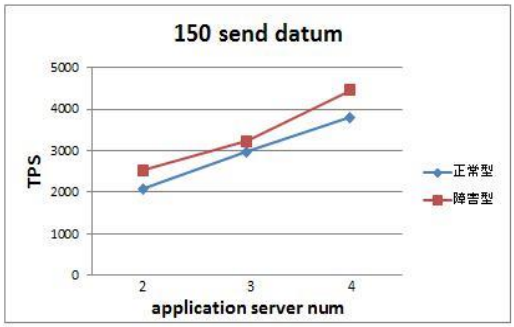
TPSスケーラビリティ（最大値）

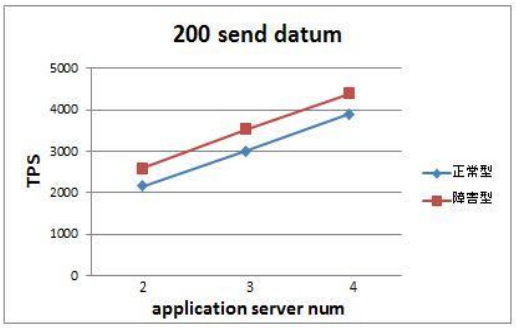
配信対象人数	アプリケーションサーバ数				型
	1	2	3	4	
200	1371	2161	3008	3902	正常型
	nop	2603	3533	4398	障害型
150	1295	2087	2982	3806	正常型
	nop	2533	3236	4455	障害型
100	1328	2034	3035	3834	正常型
	nop	2512	3352	4332	障害型
50	1323	1989	3091	3733	正常型
	nop	2529	3165	4201	障害型
10	1244	1810	2715	3395	正常型
	nop	2427	2730	4141	障害型





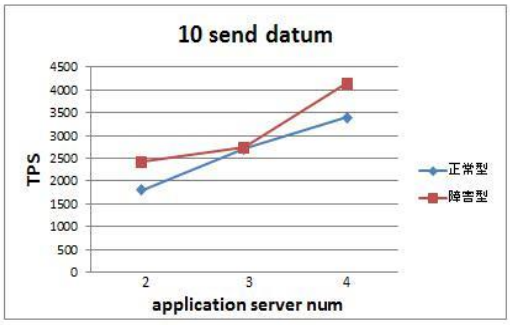


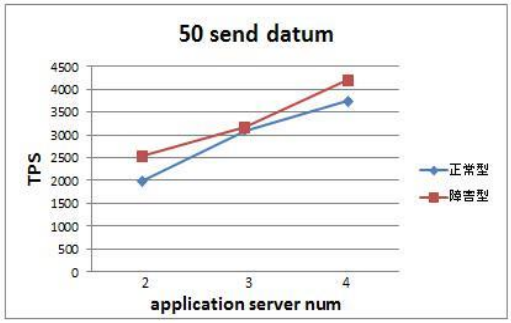


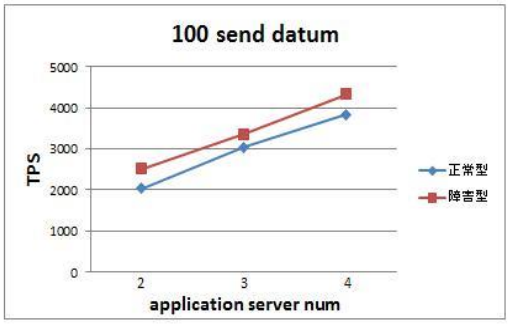


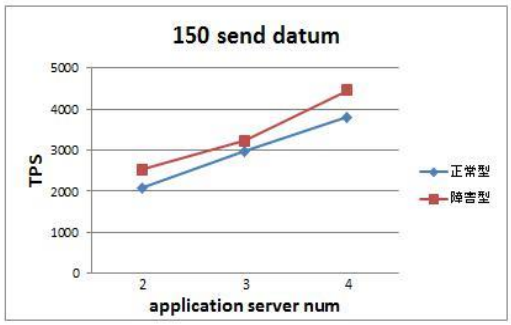
TPSスケーラビリティ（平均値）

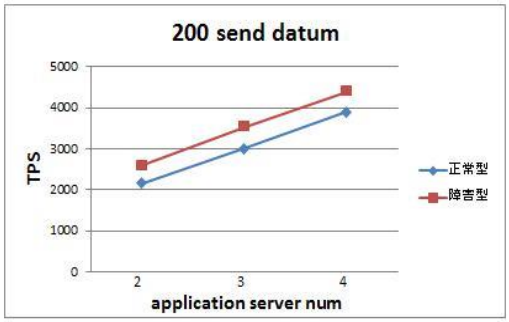
配信対象人数	アプリケーションサーバ数				型
	1	2	3	4	
200	888	1474	2080	2607	正常型
	nop	1733	2456	2874	障害型
150	894	1452	2073	2631	正常型
	nop	1775	2203	3000	障害型
100	791	1417	2105	2742	正常型
	nop	1731	2235	2770	障害型
50	908	1330	2041	2783	正常型
	nop	1756	2087	2621	障害型
10	925	1304	2027	2671	正常型
	nop	1910	2005	3164	障害型





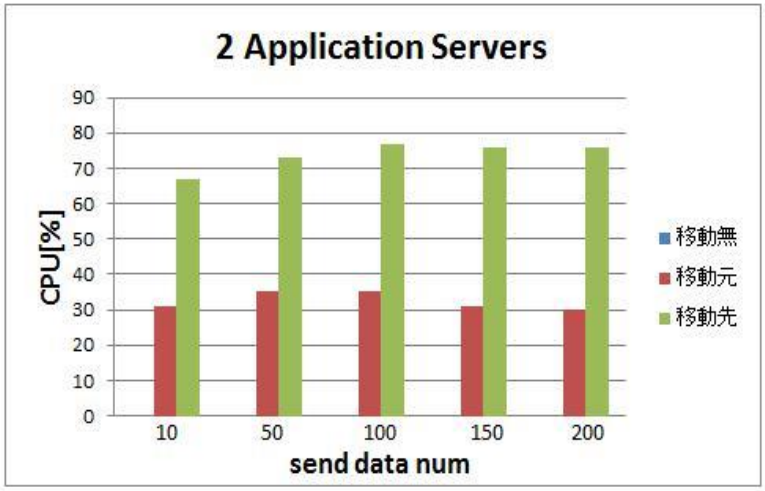


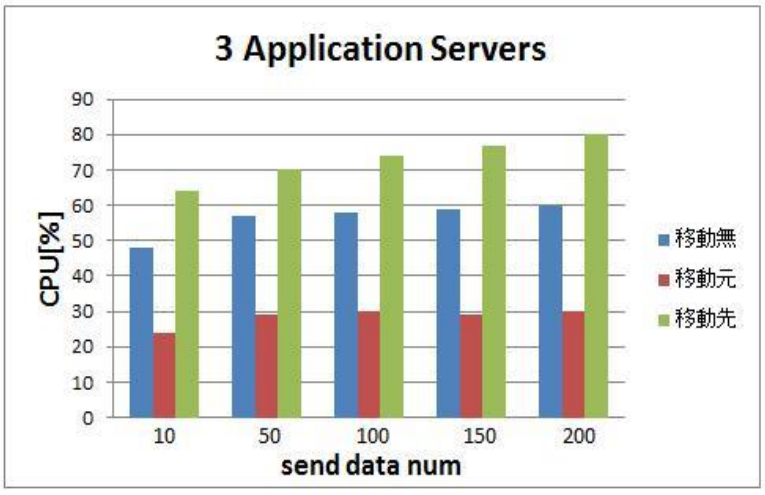


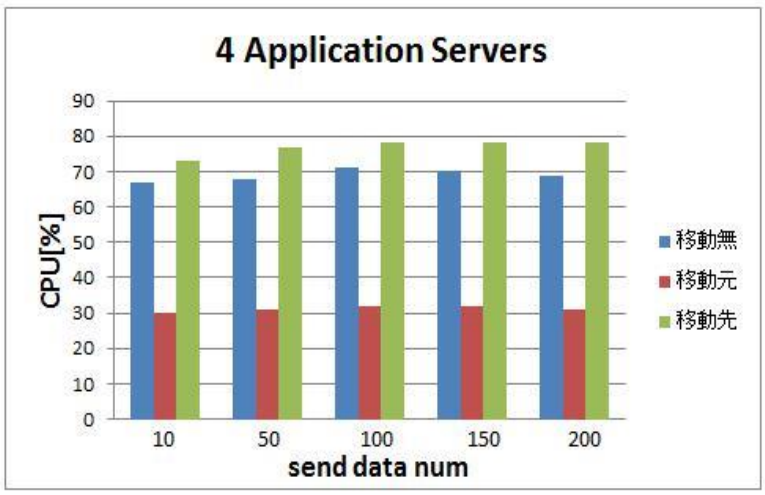


CPU（アプリケーションサーバ）

アプリケーション サーバ数	配信対象人数					型
	10	50	100	150	200	
4台	67	68	71	70	69	移動無
	30	31	32	32	31	移動元
	73	77	78	78	78	移動先
3台	48	57	58	59	60	移動無
	24	29	30	29	30	移動元
	64	70	74	77	80	移動先
2台	nop					移動無
	31	35	35	31	30	移動元
	67	73	77	76	76	移動先
1台	nop					移動無
						移動元
						移動先

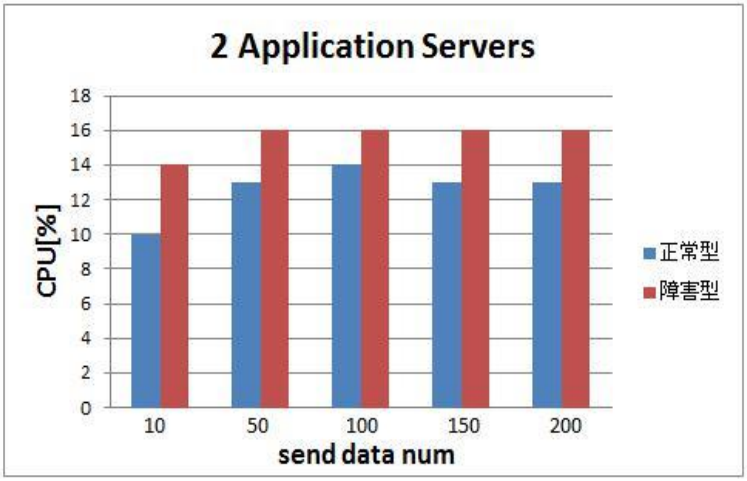


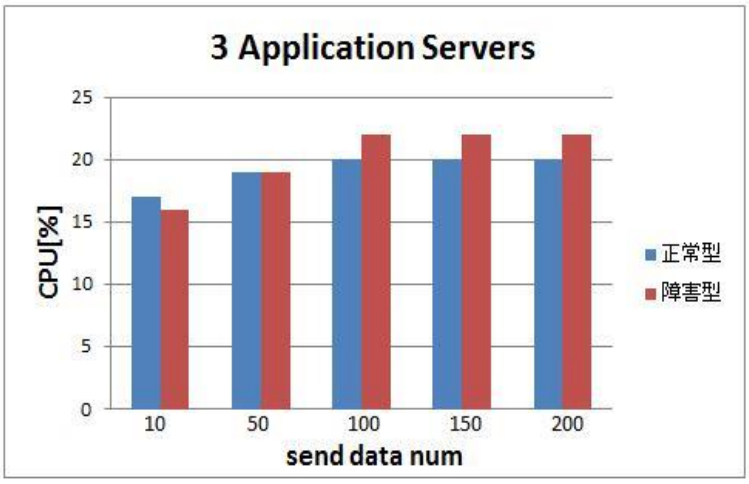


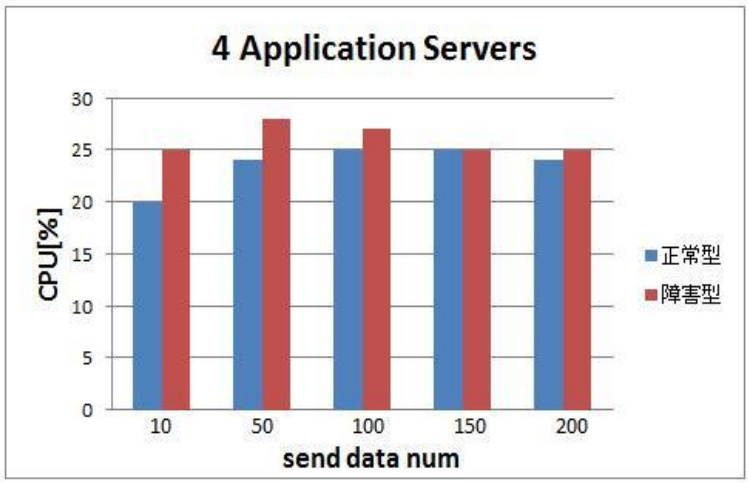


CPU（データベースサーバ）

アプリケーション サーバ数	配信対象人数					型
	10	50	100	150	200	
4台	20	24	25	25	24	正常型
	25	28	27	25	25	障害型
3台	17	19	20	20	20	正常型
	16	19	22	22	22	障害型
2台	10	13	14	13	13	正常型
	14	16	16	16	16	障害型
1台	nop					正常型
						障害型

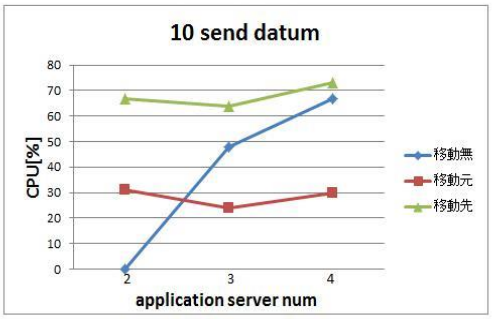


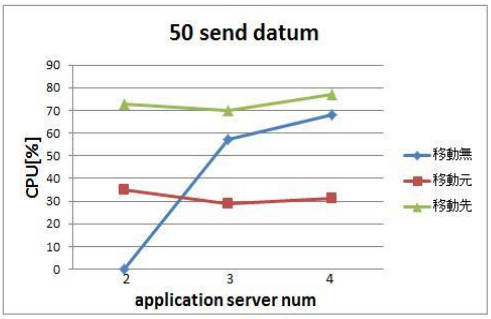


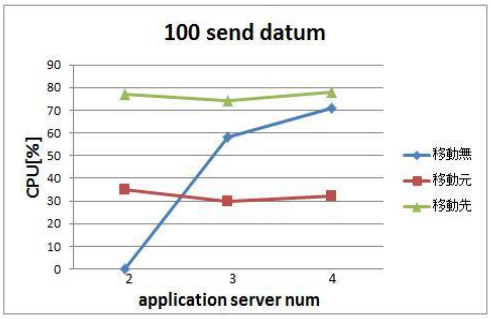


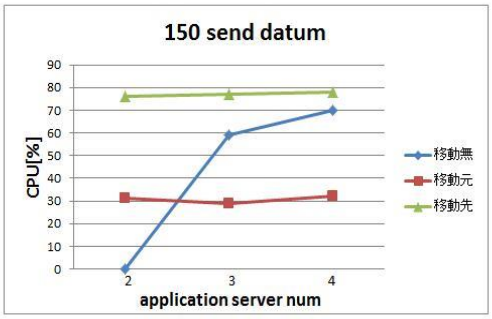
CPUスケーラビリティ（アプリケーションサーバ）

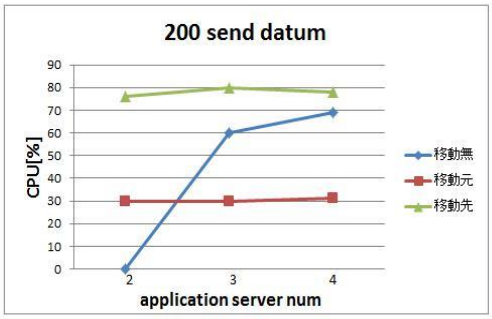
配信対象人数	アプリケーションサーバ数				型
	1	2	3	4	
200	nop	nop	60	69	移動無
		30	30	31	移動元
		76	80	78	移動先
150		nop	59	70	移動無
		31	29	32	移動元
		76	77	78	移動先
100		nop	58	71	移動無
		35	30	32	移動元
		77	74	78	移動先
50		nop	57	68	移動無
		35	29	31	移動元
		73	70	77	移動先
10		nop	48	67	移動無
		31	24	30	移動元
		67	64	73	移動先





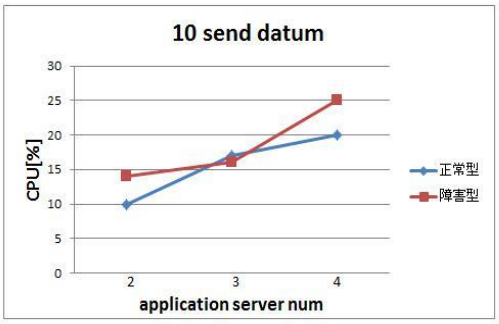


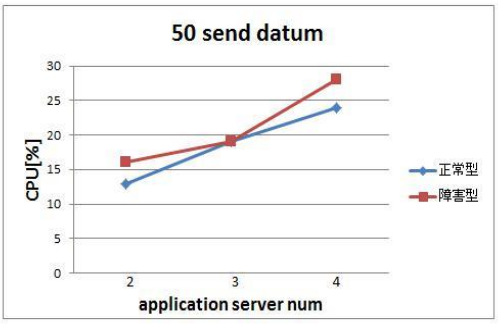


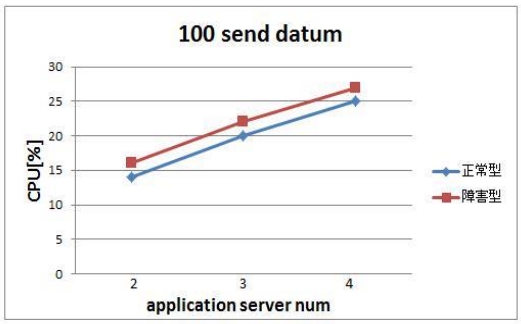


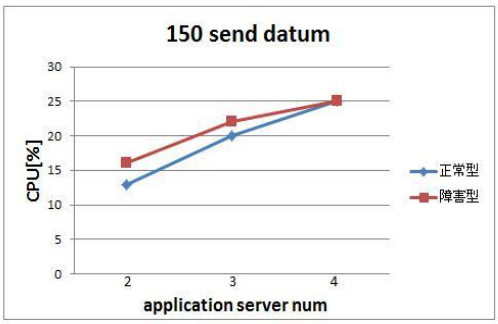
CPUスケーラビリティ（データベースサーバ）

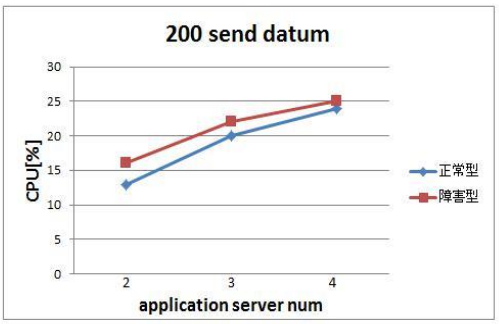
配信対象人数	アプリケーションサーバ数				型
	1	2	3	4	
200	11	13	20	24	正常型
	nop	16	22	25	障害型
150	10	13	20	25	正常型
	nop	16	22	25	障害型
100	10	14	20	25	正常型
	nop	16	22	27	障害型
50	10	13	19	24	正常型
	nop	16	19	28	障害型
10	9	10	17	20	正常型
	nop	14	16	25	障害型





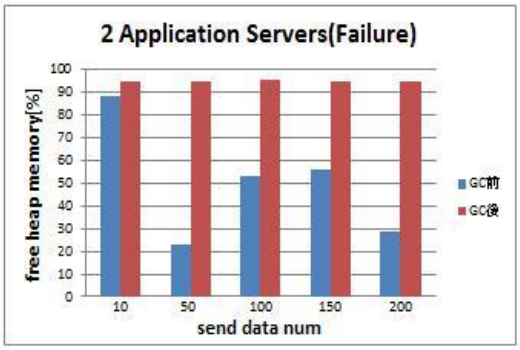


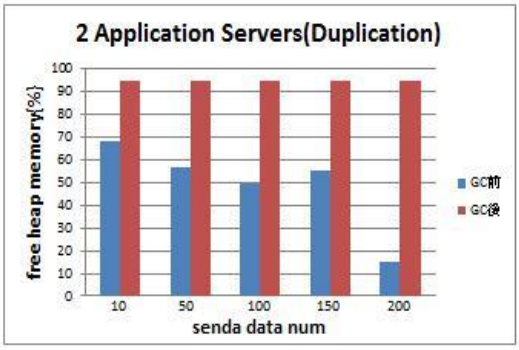


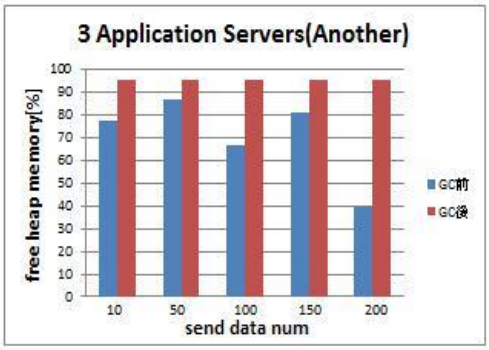


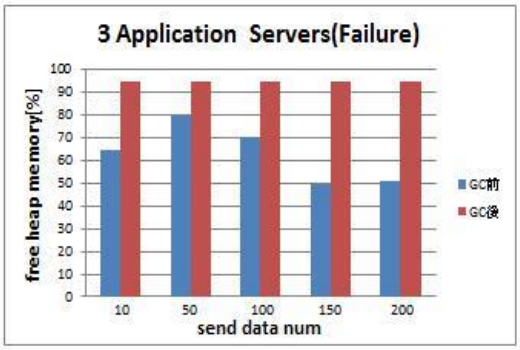
ヒープメモリ使用状況

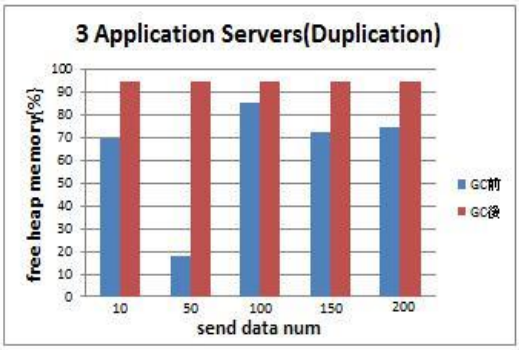
アプリケーション サーバ数	型	配信対象人数					確認タイミング
		10	50	100	150	200	
4台	移動無	55.85	5.952	45.93	55.16	39.5	GC前
		95.06	94.85	95.06	94.92	94.7	GC後
	移動元	73.96	8.167	38.4	30.6	40.92	GC前
		94.64	94.58	94.58	94.56	94.58	GC後
	移動先	72	35.59	58.42	46.06	39.48	GC前
		94.53	94.58	94.36	94.36	94.46	GC後
3台	移動無	77.27	86.7	66.37	81.11	39.23	GC前
		95.19	95.14	94.92	95.13	94.92	GC後
	移動元	64.48	79.16	70.41	49.15	50.9	GC前
		94.74	94.59	94.48	94.48	94.36	GC後
	移動先	69.15	18.06	85.17	72.54	74.55	GC前
		94.46	94.45	94.47	94.48	94.45	GC後
2台	移動無	nop					GC前
							GC後
	移動元	68.18	56.73	49.55	55.08	14.77	GC前
		94.44	94.6	94.71	94.5	94.55	GC後
	移動先	87.86	22.73	53.13	56.09	28.29	GC前
		94.53	94.47	94.98	94.44	94.47	GC後

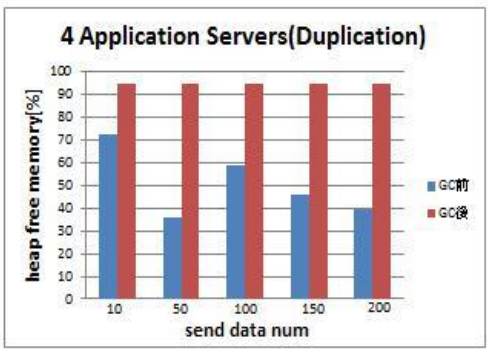


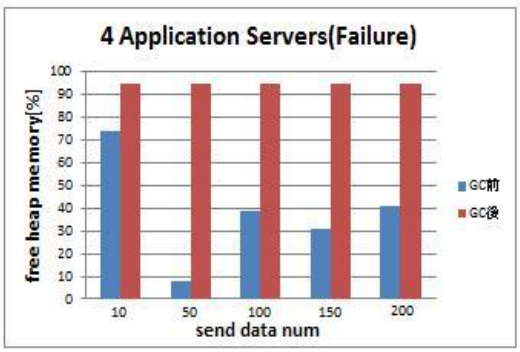


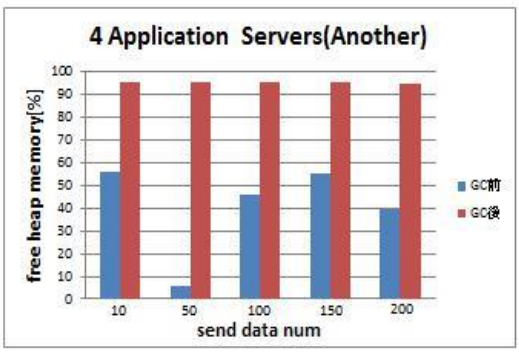












付録B

B1_障害検知時のログ

kill -9 [プロセスID]実行後に発生したエラー

1回目

アプリケーションサーバ（移動元）で発生したエラー
2017/04/27 20:53:55,066 E MSV_003 1004004 キューGETエラー(DEFAULT) @DBExecutorWorker-11(16702899)
2017/04/27 20:53:55,066 MSV_003 QueDBQueue.get() [/150.65.106.62:9042] Connection has been closed @DBExecutorWorker-11(16702899)
2017/04/27 20:53:55,066 MSV_003 QueTransactionManager.notifyError() DBエラー通知受信[1/3] () @DBExecutorWorker-11(16702899)
2017/04/27 20:53:55,067 E MSV_003 1004199 その他キューエラー(DEFAULT) @DBExecutorWorker-28(2139088189)
2017/04/27 20:53:55,067 E MSV_003 1004199 その他キューエラー(DEFAULT) @DBExecutorWorker-23(1222446138)
2017/04/27 20:53:55,067 MSV_003 QueDBQueue.delete() [/150.65.106.62:9042] Connection has been closed:20170427205246845.3551.FSV_003.FsvBLContainer[7] @DBExecutorWorker-23(1222446138)
2017/04/27 20:53:55,067 MSV_003 QueDBQueue.delete() [/150.65.106.62:9042] Connection has been closed:20170427205246840.3549.FSV_003.FsvBLContainer[7] @DBExecutorWorker-28(2139088189)
2017/04/27 20:53:55,067 E MSV_003 1004199 その他キューエラー(DEFAULT) @DBExecutorWorker-19(1452840601)
2017/04/27 20:53:55,067 MSV_003 QueDBQueue.delete() [/150.65.106.62:9042] Connection has been closed:20170427205246838.3548.FSV_003.FsvBLContainer[7] @DBExecutorWorker-19(1452840601)
2017/04/27 20:53:55,068 E MSV_003 1003007 DBコミットエラー(メッセージコミット失敗) @EventProcessorWorker-0(940975362)
2017/04/27 20:53:55,068 MSV_003 SystemCommand.executeCommit()メッセージコミット失敗 @EventProcessorWorker-0(940975362)
2017/04/27 20:53:55,068 E MSV_003 1003007 DBコミットエラー(メッセージコミット失敗) @EventProcessorWorker-0(940975362)
2017/04/27 20:53:55,068 MSV_003 SystemCommand.executeCommit()メッセージコミット失敗 @EventProcessorWorker-0(940975362)
2017/04/27 20:53:55,068 E MSV_003 1003007 DBコミットエラー(メッセージコミット失敗) @EventProcessorWorker-0(940975362)
2017/04/27 20:53:55,068 MSV_003 SystemCommand.executeCommit()メッセージコミット失敗 @EventProcessorWorker-0(940975362)

アプリケーションサーバ（移動先）で発生したエラー
2017/04/27 20:53:47,454 E MSV_004 1004199 その他キューエラー(DEFAULT) @DBExecutorWorker-29(2112259123)
2017/04/27 20:53:47,462 MSV_004 QueDBQueue.remove() [/150.65.106.62:9042] Connection has been closed:20170427205214201.547.MSV_003.150.65.106.64.45261.1493293927676 @DBExecutorWorker-29(2112259123)
2017/04/27 20:53:47,467 W MSV_004 1003008 DB削除エラー(メッセージ物理削除失敗) @EventProcessorWorker-0(1005353427)
2017/04/27 20:53:47,467 MSV_004 SystemCommand.executeDelete()メッセージ物理削除失敗 @EventProcessorWorker-0(1005353427)

2回目

アプリケーションサーバ（移動元）で発生したエラー
2017/04/27 21:30:26,149 E MSV_003 1004003 キューPUTエラー(DEFAULT) @DBExecutorWorker-18(1250330448)
2017/04/27 21:30:26,149 MSV_003 QueDBQueue.put() [/150.65.106.62:9042] Connection has been closed:20170427213024772.77554.FSV_003.FsvBLContainer[14] @DBExecutorWorker-18(1250330448)
2017/04/27 21:30:26,149 MSV_003 QueTransactionManager.notifyError() DBエラー通知受信[1/3] () @DBExecutorWorker-18(1250330448)
2017/04/27 21:30:26,154 E MSV_003 1004003 キューPUTエラー(DEFAULT) @DBExecutorWorker-3(501011841)
2017/04/27 21:30:26,154 E MSV_003 1004199 その他キューエラー(DEFAULT) @DBExecutorWorker-19(736444218)
2017/04/27 21:30:26,154 MSV_003 QueDBQueue.put() [/150.65.106.62:9042] Connection has been closed:20170427213024812.79114.FSV_003.FsvBLContainer[1] @DBExecutorWorker-3(501011841)
2017/04/27 21:30:26,154 MSV_003 QueTransactionManager.notifyError() DBエラー通知受信[1/3] () @DBExecutorWorker-3(501011841)
2017/04/27 21:30:26,154 MSV_003 QueDBQueue.delete() [/150.65.106.62:9042] Connection has been closed:20170427213016027.79394.FSV_003.FsvBLContainer[0] @DBExecutorWorker-19(736444218)
2017/04/27 21:30:26,155 E MSV_003 1004003 キューPUTエラー(DEFAULT) @DBExecutorWorker-13(1686023502)
2017/04/27 21:30:26,155 MSV_003 QueDBQueue.put() [/150.65.106.62:9042] Channel has been closed:20170427213024792.77936.FSV_003.FsvBLContainer[6] @DBExecutorWorker-13(1686023502)
2017/04/27 21:30:26,155 MSV_003 QueTransactionManager.notifyError() DBエラー通知受信[1/3] () @DBExecutorWorker-13(1686023502)
2017/04/27 21:30:26,155 E MSV_003 1004003 キューPUTエラー(DEFAULT) @DBExecutorWorker-12(1487309573)
2017/04/27 21:30:26,155 MSV_003 QueDBQueue.put() [/150.65.106.62:9042] Connection has been closed:20170427213024786.80084.FSV_003.FsvBLContainer[12] @DBExecutorWorker-12(1487309573)
2017/04/27 21:30:26,155 MSV_003 QueTransactionManager.notifyError() DBエラー通知受信[1/3] () @DBExecutorWorker-12(1487309573)
2017/04/27 21:30:26,186 E MSV_003 1004008 トランザクション保存発生(/app/RtFA/RtFA_E3/dump/tmpMSV_003_20170427_FSV_003) @EventProcessorWorker-0(177506604)
2017/04/27 21:30:26,186 MSV_003 QueManager.saveTransaction() トランザクションデータ保存:(/app/RtFA/RtFA_E3/dump/tmpMSV_003_20170427_FSV_003) @EventProcessorWorker-0(177506604)
2017/04/27 21:30:26,186 E MSV_003 1003007 DBコミットエラー(メッセージコミット失敗) @EventProcessorWorker-0(177506604)
2017/04/27 21:30:26,186 MSV_003 SystemCommand.executeCommit()メッセージコミット失敗 @EventProcessorWorker-0(177506604)
2017/04/27 21:30:26,188 E MSV_003 1004008 トランザクション保存発生(/app/RtFA/RtFA_E3/dump/tmpMSV_003_20170427_FSV_003) @EventProcessorWorker-0(177506604)
2017/04/27 21:30:26,188 MSV_003 QueManager.saveTransaction() トランザクションデータ保存:(/app/RtFA/RtFA_E3/dump/tmpMSV_003_20170427_FSV_003) @EventProcessorWorker-0(177506604)
2017/04/27 21:30:26,188 E MSV_003 1003007 DBコミットエラー(メッセージコミット失敗) @EventProcessorWorker-0(177506604)
2017/04/27 21:30:26,188 MSV_003 SystemCommand.executeCommit()メッセージコミット失敗 @EventProcessorWorker-0(177506604)
2017/04/27 21:30:26,188 E MSV_003 1003007 DBコミットエラー(メッセージコミット失敗) @EventProcessorWorker-0(177506604)
2017/04/27 21:30:26,188 MSV_003 SystemCommand.executeCommit()メッセージコミット失敗 @EventProcessorWorker-0(177506604)
2017/04/27 21:30:26,190 E MSV_003 1004008 トランザクション保存発生(/app/RtFA/RtFA_E3/dump/tmpMSV_003_20170427_FSV_003) @EventProcessorWorker-0(177506604)
2017/04/27 21:30:26,190 MSV_003 QueManager.saveTransaction() トランザクションデータ保存:(/app/RtFA/RtFA_E3/dump/tmpMSV_003_20170427_FSV_003) @EventProcessorWorker-0(177506604)
2017/04/27 21:30:26,190 E MSV_003 1003007 DBコミットエラー(メッセージコミット失敗) @EventProcessorWorker-0(177506604)
2017/04/27 21:30:26,190 MSV_003 SystemCommand.executeCommit()メッセージコミット失敗 @EventProcessorWorker-0(177506604)
2017/04/27 21:30:26,192 E MSV_003 1004008 トランザクション保存発生(/app/RtFA/RtFA_E3/dump/tmpMSV_003_20170427_FSV_003) @EventProcessorWorker-0(177506604)
2017/04/27 21:30:26,192 MSV_003 QueManager.saveTransaction() トランザクションデータ保存:(/app/RtFA/RtFA_E3/dump/tmpMSV_003_20170427_FSV_003) @EventProcessorWorker-0(177506604)
2017/04/27 21:30:26,192 E MSV_003 1003007 DBコミットエラー(メッセージコミット失敗) @EventProcessorWorker-0(177506604)
2017/04/27 21:30:26,192 MSV_003 SystemCommand.executeCommit()メッセージコミット失敗 @EventProcessorWorker-0(177506604)

アプリケーションサーバ（移動先）で発生したエラー
2017/04/27 21:30:15,928 E MSV_004 1004199 その他キューエラー(DEFAULT) @DBExecutorWorker-23(1219537849)
2017/04/27 21:30:15,928 MSV_004 QueDBQueue.delete() [/150.65.106.62:9042] Connection has been closed:20170427213005530.71855.FSV_004.FsvBLContainer[11] @DBExecutorWorker-23(1219537849)
2017/04/27 21:30:15,928 E MSV_004 1004004 キューGETエラー(DEFAULT) @DBExecutorWorker-0(1704788886)
2017/04/27 21:30:15,928 MSV_004 QueDBQueue.get() [/150.65.106.62:9042] Connection has been closed @DBExecutorWorker-0(1704788886)
2017/04/27 21:30:15,928 MSV_004 QueTransactionManager.notifyError() DBエラー通知受信[1/3] () @DBExecutorWorker-0(1704788886)
2017/04/27 21:30:15,930 E MSV_004 1003007 DBコミットエラー(メッセージコミット失敗) @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,930 MSV_004 SystemCommand.executeCommit()メッセージコミット失敗 @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,930 E MSV_004 1004003 キューPUTエラー(DEFAULT) @DBExecutorWorker-26(1574839908)
2017/04/27 21:30:15,930 MSV_004 QueDBQueue.put() [/150.65.106.62:9042] Connection has been closed:20170427213016036.79401.FSV_003.FsvBLContainer[0] @DBExecutorWorker-26(1574839908)
2017/04/27 21:30:15,930 MSV_004 QueTransactionManager.notifyError() DBエラー通知受信[1/3] () @DBExecutorWorker-26(1574839908)
2017/04/27 21:30:15,936 E MSV_004 1004199 その他キューエラー(DEFAULT) @DBExecutorWorker-1(1357031265)
2017/04/27 21:30:15,936 MSV_004 QueDBQueue.delete() [/150.65.106.62:9042] Connection has been closed:20170427213005534.71859.FSV_004.FsvBLContainer[11] @DBExecutorWorker-1(1357031265)
2017/04/27 21:30:15,938 E MSV_004 1004199 その他キューエラー(DEFAULT) @DBExecutorWorker-22(1223797689)
2017/04/27 21:30:15,938 MSV_004 QueDBQueue.delete() [/150.65.106.62:9042] Connection has been closed:20170427213005532.71857.FSV_004.FsvBLContainer[11] @DBExecutorWorker-22(1223797689)
2017/04/27 21:30:15,950 E MSV_004 1004003 キューPUTエラー(DEFAULT) @DBExecutorWorker-2(1367869833)
2017/04/27 21:30:15,950 E MSV_004 1004003 キューPUTエラー(DEFAULT) @DBExecutorWorker-20(377998828)
2017/04/27 21:30:15,950 MSV_004 QueDBQueue.put() [/150.65.106.62:9042] Connection has been closed:20170427213014621.73607.FSV_004.FsvBLContainer[6] @DBExecutorWorker-20(377998828)
2017/04/27 21:30:15,950 MSV_004 QueDBQueue.put() [/150.65.106.62:9042] Connection has been closed:20170427213014590.72739.FSV_004.FsvBLContainer[2] @DBExecutorWorker-2(1367869833)
2017/04/27 21:30:15,950 MSV_004 QueTransactionManager.notifyError() DBエラー通知受信[2/3] () @DBExecutorWorker-2(1367869833)
2017/04/27 21:30:15,950 MSV_004 QueTransactionManager.notifyError() DBエラー通知受信[1/3] () @DBExecutorWorker-20(377998828)
2017/04/27 21:30:15,950 E MSV_004 1004199 その他キューエラー(DEFAULT) @DBExecutorWorker-16(808075424)
2017/04/27 21:30:15,950 MSV_004 QueDBQueue.delete() [/150.65.106.62:9042] Connection has been closed:20170427213005531.71856.FSV_004.FsvBLContainer[11] @DBExecutorWorker-16(808075424)
2017/04/27 21:30:15,954 E MSV_004 1004008 トランザクション保存発生(/app/RtFA/RtFA_E3/dump/tmpMSV_004_20170427_unknown) @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,954 MSV_004 QueManager.saveTransaction() トランザクションデータ保存:(/app/RtFA/RtFA_E3/dump/tmpMSV_004_20170427_unknown) @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,954 MSV_004 MsvMeshMessageHandler.storeMessage() com.fsc.exceptions.QueException: DEFAULT [1004003-0]DEFAULT @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,950 E MSV_004 1004004 キューGETエラー(DEFAULT) @DBExecutorWorker-11(451041867)
2017/04/27 21:30:15,957 MSV_004 QueDBQueue.get() [/150.65.106.62:9042] Connection has been closed @DBExecutorWorker-11(451041867)
2017/04/27 21:30:15,957 MSV_004 QueTransactionManager.notifyError() DBエラー通知受信[1/3] () @DBExecutorWorker-11(451041867)
2017/04/27 21:30:15,958 E MSV_004 1004199 その他キューエラー(DEFAULT) @DBExecutorWorker-9(606226240)
2017/04/27 21:30:15,958 MSV_004 QueDBQueue.delete() [/150.65.106.62:9042] Connection has been closed:20170427213005522.71847.FSV_004.FsvBLContainer[11] @DBExecutorWorker-9(606226240)
2017/04/27 21:30:15,959 E MSV_004 1003007 DBコミットエラー(メッセージコミット失敗) @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,959 MSV_004 SystemCommand.executeCommit()メッセージコミット失敗 @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,959 E MSV_004 1003007 DBコミットエラー(メッセージコミット失敗) @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,959 MSV_004 SystemCommand.executeCommit()メッセージコミット失敗 @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,963 E MSV_004 1004008 トランザクション保存発生(/app/RtFA/RtFA_E3/dump/tmpMSV_004_20170427_FSV_004) @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,963 MSV_004 QueManager.saveTransaction() トランザクションデータ保存:(/app/RtFA/RtFA_E3/dump/tmpMSV_004_20170427_FSV_004) @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,963 E MSV_004 1003007 DBコミットエラー(メッセージコミット失敗) @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,963 MSV_004 SystemCommand.executeCommit()メッセージコミット失敗 @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,971 E MSV_004 1004008 トランザクション保存発生(/app/RtFA/RtFA_E3/dump/tmpMSV_004_20170427_FSV_004) @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,971 MSV_004 QueManager.saveTransaction() トランザクションデータ保存:(/app/RtFA/RtFA_E3/dump/tmpMSV_004_20170427_FSV_004) @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,971 E MSV_004 1003007 DBコミットエラー(メッセージコミット失敗) @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,971 MSV_004 SystemCommand.executeCommit()メッセージコミット失敗 @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,971 E MSV_004 1003007 DBコミットエラー(メッセージコミット失敗) @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,971 MSV_004 SystemCommand.executeCommit()メッセージコミット失敗 @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,971 E MSV_004 1003007 DBコミットエラー(メッセージコミット失敗) @EventProcessorWorker-0(1005353427)
2017/04/27 21:30:15,972 MSV_004 SystemCommand.executeCommit()メッセージコミット失敗 @EventProcessorWorker-0(1005353427)

付録C

C1_追加実験データ

追加実験1

メモリの使用量の再確認
send data num = 50 のとき、メモリ使用量が大きくなっており、
特にヒープメモリの空き領域が10%に満たない状態となったケースで追加実験を行っている。
（アプリケーションサーバ4台、キュー用DBサーバ（Cassandra）1台を停止させた状態）
追加で実験を行い、メモリの使用量の再確認を行った。
10回、実験を繰り返し、メモリの使用状態を確認している。

実施回	ヒープメモリ空き容量[%]							
	アプリケーションサーバ1		アプリケーションサーバ2		アプリケーションサーバ3		アプリケーションサーバ4	
	GC前	GC後	GC前	GC後	GC前	GC後	GC前	GC後
1	53.57	94.52	47.51	94.53	43.71	94.17	61.7	93.94
2	38.89	95.03	50.69	94.93	37.55	94.57	60.24	94.47
3	28.57	94.75	44.84	94.63	37.59	94.36	50.76	94.27
4	40.81	95.04	36.35	95.03	32.73	94.47	35.93	94.36
5	48.91	95.05	31.25	95.06	51.1	94.46	44.44	94.35
6	56.53	94.73	55.95	94.74	47.25	94.26	31.37	94.37
7	40.67	95.05	59.13	95.03	43.66	94.46	50.25	94.35
8	42.96	94.92	44.05	95.06	44.46	94.56	39.56	94.46
9	38.99	95.05	54.27	94.92	21.8	94.5	44.84	94.35
10	32.14	95.05	44.49	95.02	28.26	94.47	41.86	94.36

平均　43.49
最大　61.66
最小　21.79

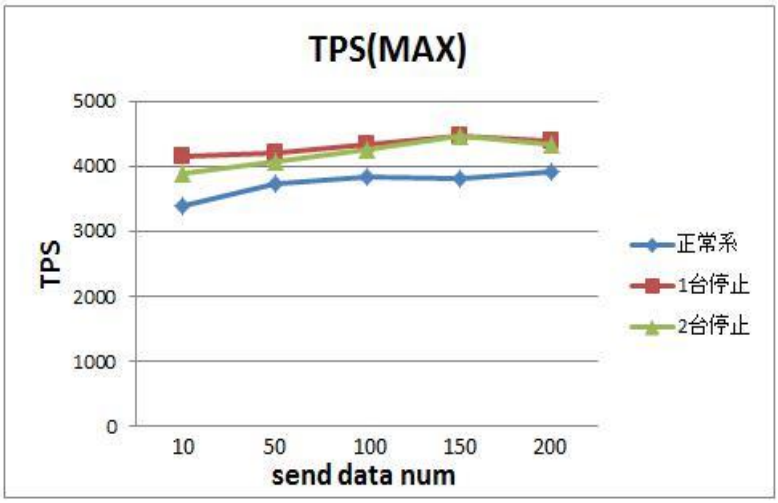
ヒープメモリの空き容量が10%以下になる現象は再現できなかったものの、
最小で21.8%, 最大で61.7%と幅があり、特に規則性は見られないことが分かった。
実験を重ねて行う中で、データを取得するタイミングに依存して発生した現象と考えられる。

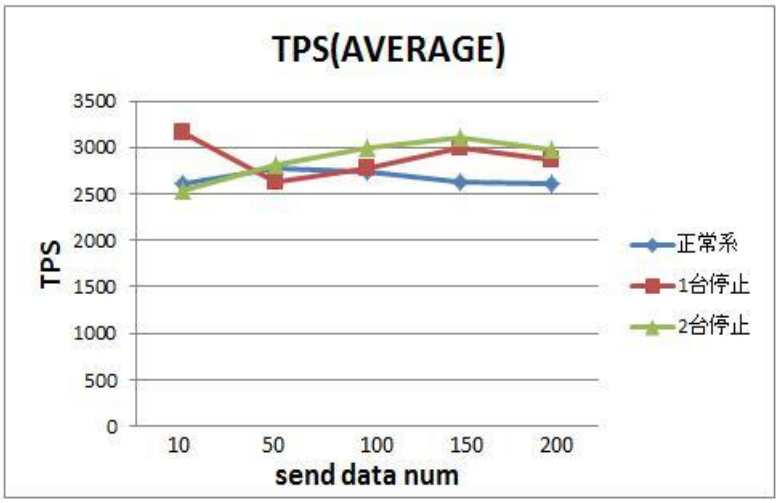
追加実験2

アプリサーバ4台、2台のCassandraが停止している状態のデータを取得して比較

配信対象人数	TPS 最高到達点			TPS（平均）			CPU使用率（最大）[%]								
							アプリケーションサーバ						データベース サーバ		
	正常型	1台停止	2台停止	正常型	1台停止	2台停止	正常型	1台停止移動元	1台停止移動元	1台停止移動無	2台停止移動元	2台停止移動元	正常型	1台停止	2台停止
10	3395	4141	3885	2617	3164	2529	60	30	73	67	33	67	20	25	22
50	3733	4201	4058	2783	2621	2821	66	31	77	68	39	76	24	28	24
100	3834	4332	4243	2742	2770	2998	64	32	78	71	35	77	25	27	25
150	3806	4455	4453	2631	3000	3099	65	32	78	70	36	76	25	25	25
200	3902	4398	4321	2607	2874	2985	65	31	78	69	35	78	24	25	26

※2台停止で取得したApplication ServerのCPUの値は、2台のうち大きい方の値を記載





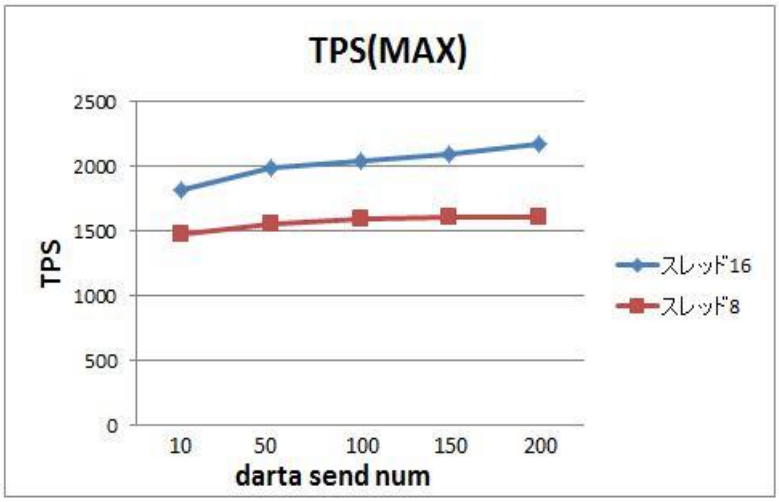
2台停止時も1台停止時と傾向は、ほぼ同じ。
正常時よりも数値が高い。MAX値は1台停止時の方がやや高い値だったが、
AVERAGEでは2台停止時の方がやや高い値となっている。

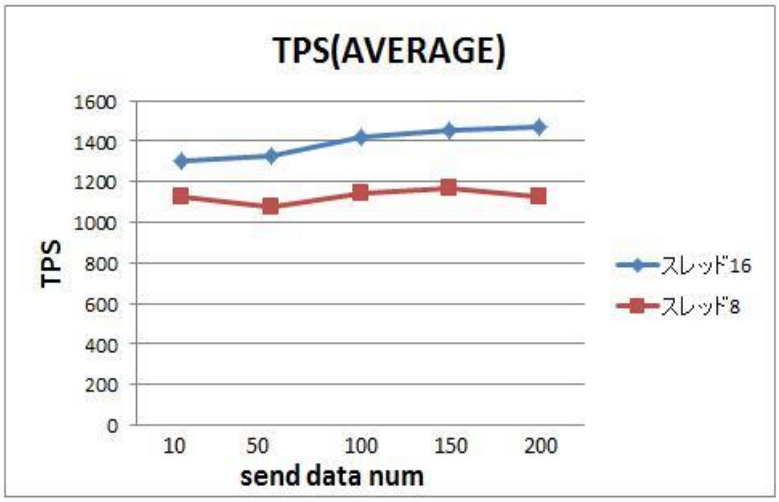
追加実験3

TPS全体の処理傾向として、send message num 50を超えると数値があがりはじめ、
100以降は同程度の性能を出力している状態ともとらえることができる。
これはアプリケーションサーバの同時実行スレッド数を16と設定しているため、
send message numが10だと、全てのスレッドを使用する状態にならず、
50を超えると全スレッドが同時に処理を行う状態になるためではないかと推測した。
このため、アプリケーションサーバの同時実行スレッド数の設定を8にすることで、
send message numが10～200で、同程度の性能値が現れるか確認を行った。
アプリケーションサーバ数は2台、KVS型を用いて実験を行っている。

配信対象人数	TPS（最高到達点）		TPS（平均）		CPU使用率（最大）[%]			
					アプリケーションサーバ		データベースサーバ	
	スレッド16	スレッド8	スレッド16	スレッド8	スレッド16	スレッド8	スレッド16	スレッド8
10	1810	1468	1304	1123	56	49	10	11
50	1989	1546	1330	1073	59	53	13	11
100	2034	1592	1417	1147	61	55	14	11
150	2087	1607	1452	1170	62	55	13	10
200	2161	1602	1474	1128	62	57	13	11

※アプリケーションサーバのCPUの値は、2台のうち大きい方の値を記載





スレッド数16->8に対し、TPS値は全体的に10～20%減少している。
それぞれを相対的な数値として比較を行うと、
スレッド16での最大～最小の差は、TPS(MAX)で19%, TPS(AVERAGE)で13%となっている。
スレッド8での最大～最小の差は、TPS(MAX)で9%, TPS(AVERAGE)で9%
全体的に差が小さく変化率が緩やかになっており、水平に近い変化を示し、
予想通り、配信対象人数の影響度が下がり、性能に差が出なくなる傾向であることが分かった。