【VLDB 2011 勉強会】

Session 12: Cloud Computing and High-Availability

担当: 上田 高徳 (早稲田大学)

Session 12: Cloud Computing and High-Availability 概要

（論文1）“Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration”

- Sudipto Das (University of California, Santa Barbara), Shoji Nishimura (NEC Corporation), Divyakant Agrawal, Amr El Abbadi (University of California, Santa Barbara)

（論文2）“iCBS: Incremental Cost-based Scheduling under Piecewise Linear SLAs”

- Yun Chi, Hyun Jin Moon, Hakan Hacıgümüş (NEC Laboratories America)

（論文3）“RemusDB: Transparent High Availability for Database Systems”

- Umar Farooq Minhas (University of Waterloo), Shriram Rajagopalan, Brendan Cully (University of British Columbia), Ashraf Aboulnaga, Kenneth Salem (University of Waterloo), Andrew Warfield (University of British Columbia)

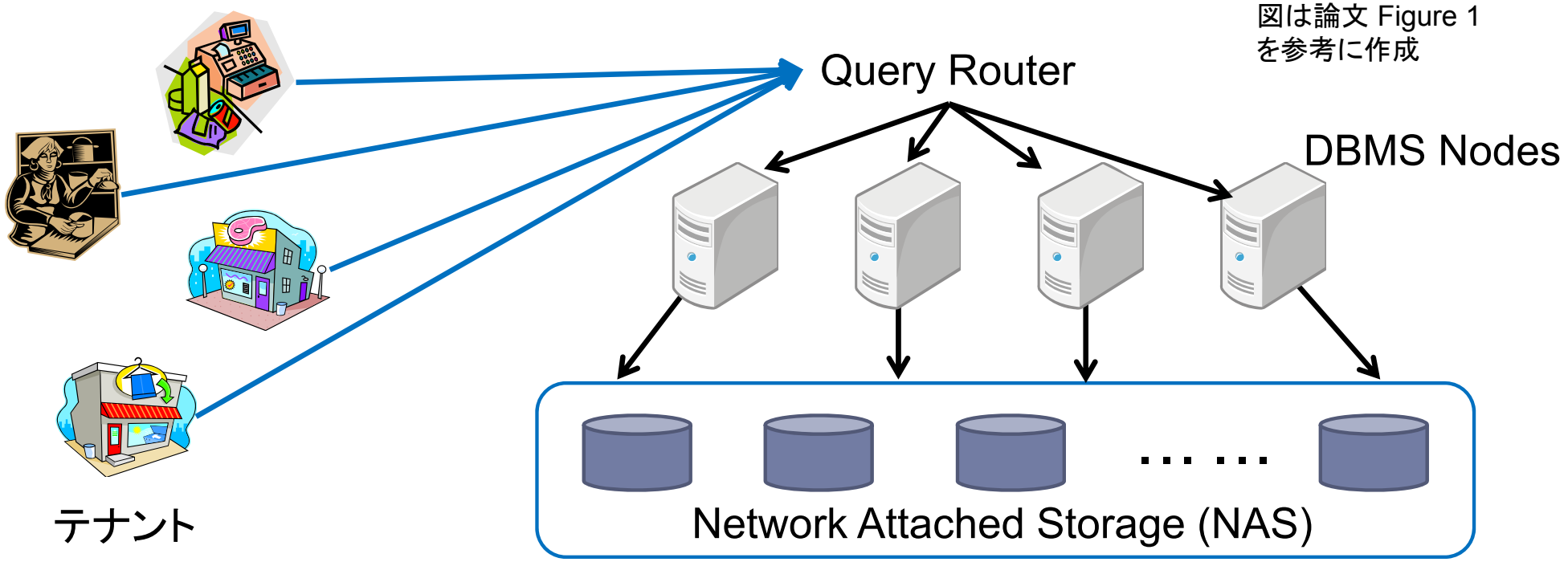
Best Paper

- 論文をより理解するためにあった方がよい知識
  - 論文1
    - 特になし
  - 論文2
    - 問題定義から始まりそれを解く論文なので特になし．積分の基本知識は必要
  - 論文3
    - Remus のネットワークバッファリング機能（NSDI 2008の論文を読むのが良い）

（論文1）“Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration”

- 前提
  - テナント型のクラウドサービスにおけるDBMS
  - NASストレージにデータが格納されている
  - DBMSプロセスを共有するモデル（Shared Process Model）
- 本論文の貢献
  - テナントのマシン間マイグレーションにおいて 300ms 以下の利用停止時間を実現
  - 移動直後のパフォーマンス低下を 5%～15% に削減

図は論文 Figure 1 を参考に作成



マイグレーションの3ステップ（テナントcをノードAからノードBに移動）

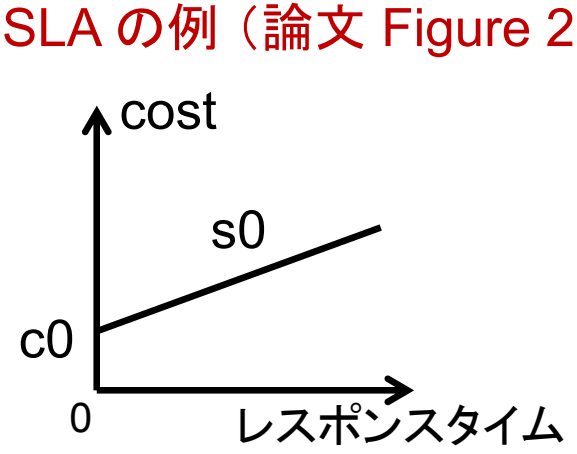
1. Begin Migration
   - AのDBキャッシュのスナップショットをBに転送開始
   - 転送中もAでのトランザクション処理は継続
2. Iterative Copying
   - 初回のスナップショット以降に更新があった分をBに転送開始
   - 転送を繰り返して，i 回目と $i+1$ 回目の転送量がほぼ等しくなる（十分に転送された）か，指定の回数に達したら終了
3. Atomic Handover
   - 最終処理として cの処理を停止し完全にAの状態をBに転送する
   - Aのアクティブなトランザクションの状態をBに転送
   - Aのコミット済みトランザクションの状態をNASへフラッシュしてからBに処理を引き継ぐ
   - 障害の可能性を考慮してatomic性を保証

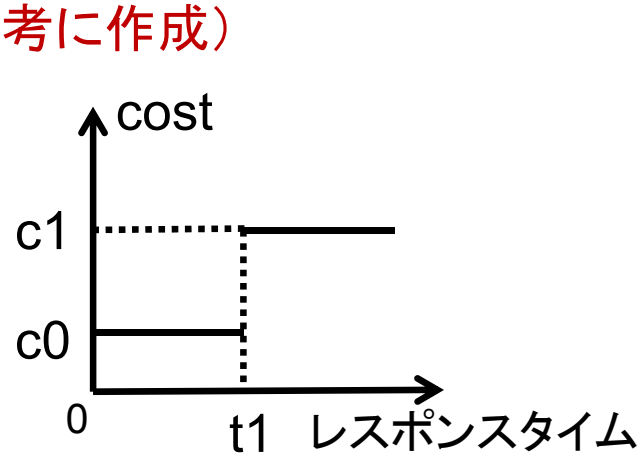
(論文2)

“iCBS: Incremental Costbased Scheduling under Piecewise Linear SLAs”

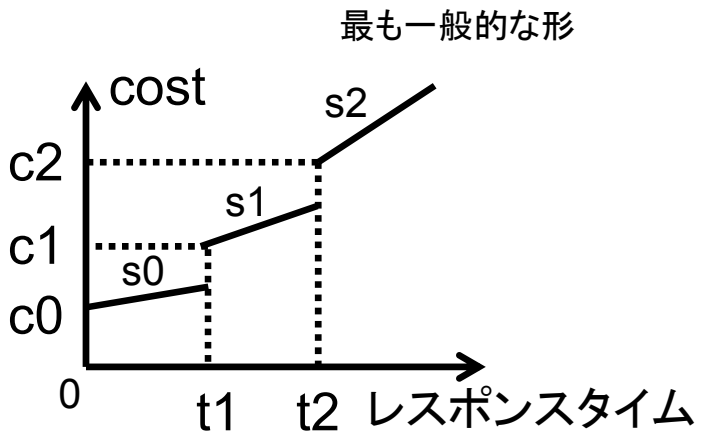
- **前提**
  - クラウドサービスのプロバイダの立場
  - クエリを返すまでの時間でSLA (Service Level Agreement) が定義される
    - 時間が掛かるほどコスト（ペナルティ）が高くなる
- **目標**
  - SLA がそれぞれ異なるクエリが複数あるときに，掛かるコストを最小化したい
    - オフライン問題（到着するクエリが事前に分かっている場合）でもNP完全な問題
- **本論文の貢献**
  - 既存手法 CBS はクエリ数が N のとき計算時間が $O(N)$ なのでインクリメンタル化
  - SLA が区分線形関数で記述されるときの大半で計算量を $O(\log N)$ に
  - 区分線形関数で記述されるときのすべての場合で $O(\log^2 N)$ に

SLA の例（論文 Figure 2 を参考に作成）





最も一般的な形



- ベースになる既存手法 CBS の考え方と問題点
  - SLA が $f(t)$ で定義されたあるクエリを時刻 t において実行したときに削減できるコストの推定値を式(1) で定義
  - もっとも値が大きいクエリから実行していくヒューリスティックな方法をとる
  - スケジューリング時に全クエリで 式(1) の積分を実行する必要がある ⇒ $O(N)$
- 提案手法 iCBSの着眼点
  - 式 (1) が簡単に計算できる場合がある
  - あるいは値が変わってもクエリ間の優先順序が変わらない場合がある
  - SLAの概形を場合分けして頑張って解く！

t_0: クエリ到着時間
a : パラメータ

$$r(t) = \int_0^{\infty} ae^{-a\tau} f(t - t_0 + \tau) d\tau - f(t - t_0) \qquad (1)$$

実行を延期した場合に発生するコスト.
$ae^{-a\tau}$ で緊急度を調整している.

いますぐ実行した場合のコスト

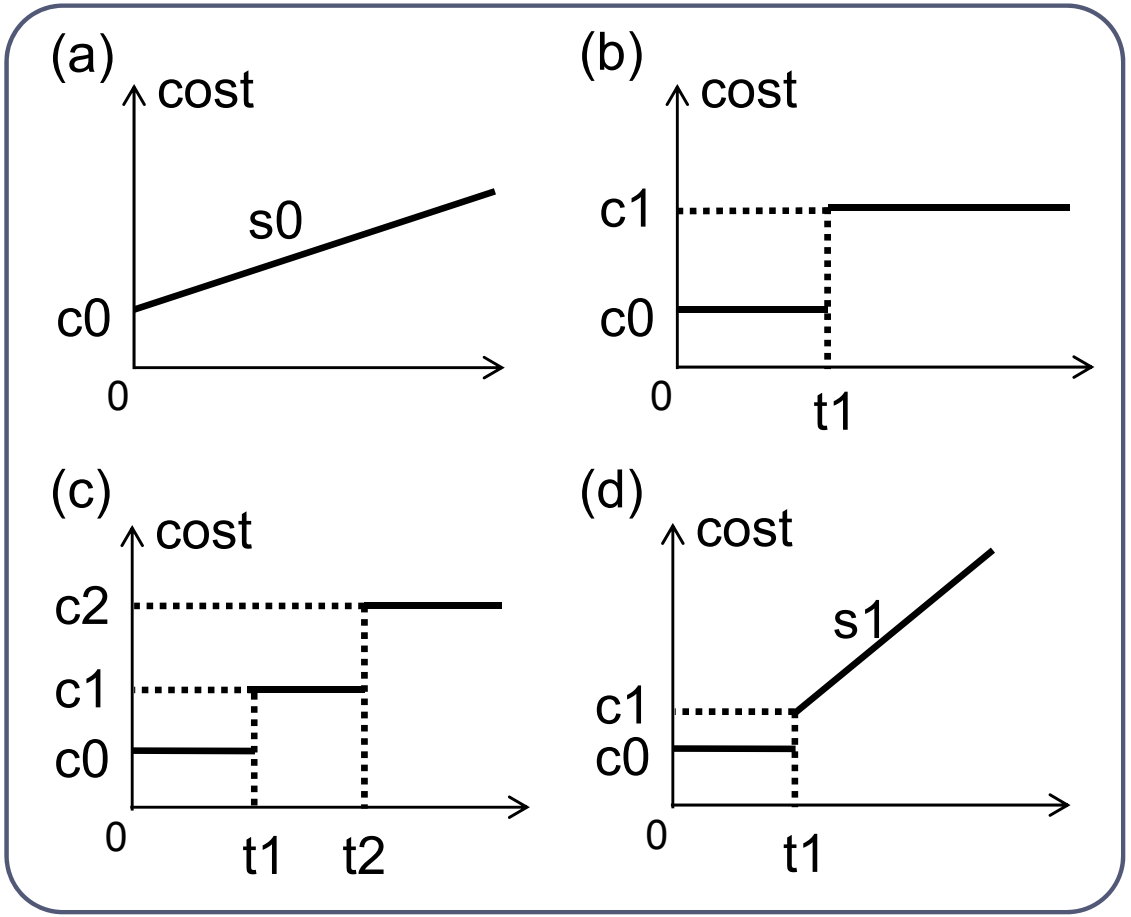
（論文の Figure 3 や次のスライドの (c) の $r(t)$ を参照するとよい）

(論文2)

“iCBS: Incremental Costbased Scheduling under Piecewise Linear SLAs”

- SLAの概形ごとの $r(t)$ の値を求めてみると
  - 以下の (a) ～ (d) の 4パターンは Priority Queue で優先順序を管理できて O(logN)
  - ただし (c) は 2つ，(d) は3つの Priority Queue が必要
    - $r(t)$ の不連続な部分で優先順序が入れ替わるため

SLAの概形（論文 Figure 2 を参考に作成）

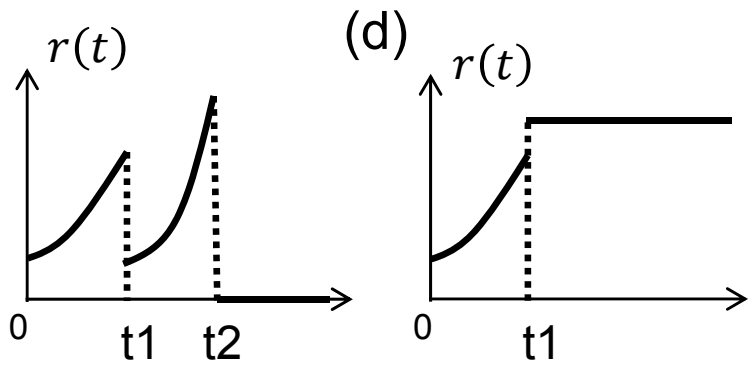


左図のSLAごとの $r(t)$ の式や概形
（論文 Figure 5 を参考に作成）

(a) $r(t) = b/a$ 時間に依存しない

(b) $r(t) = c_1 e^{-a(t_1 - t)}$

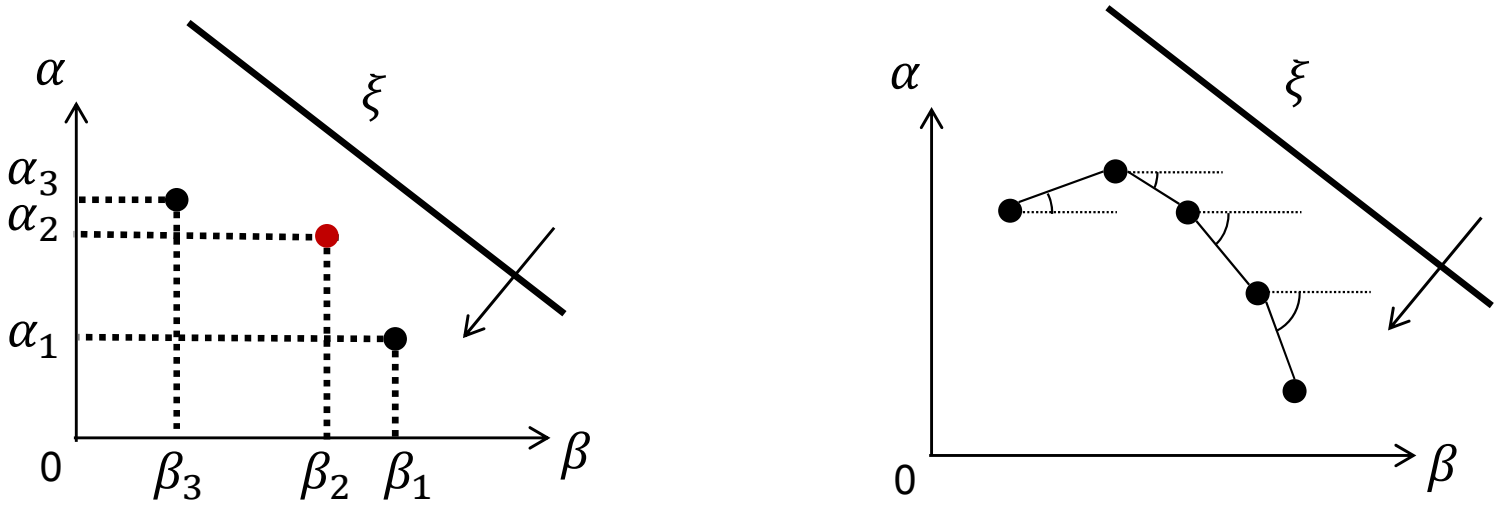
クエリ間の
順序関係が保たれる



- より一般的な場合のスコアは

$$f(\xi) = \alpha + \beta\xi \quad (\xi = e^{at})$$

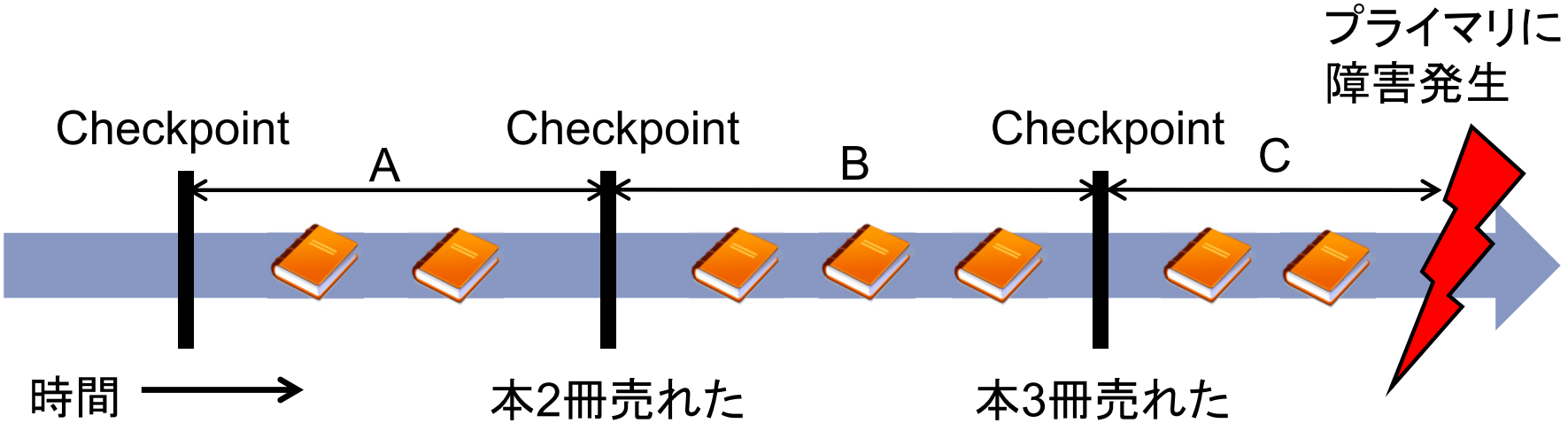
- とあらわすことができ，α, βの平面にプロットすると点になる
- ξ を原点方向に動かして最初にぶつかった点が最も高い値を持つ
- 凸包を求めておけば，効率よく探索できる
  - β 軸と点のなす角（符号付き）が β 方向に進むにつれ単調減少である性質から ξのなす角がわかっていれば二分探索可能



（論文3）“RemusDB: Transparent High Availability for Database Systems”

- 目標
  - 仮想マシン上で動作しているDBMSの高可用性を効率よく実現
- 本論文の貢献
  - 仮想マシンの Xen で HA を実現する Remus を DBMS 向けに最適化
    - Remusは NSDI 2008 のベストペーパー
  - DBMS の実装変更を最小限に高可用性を実現
  - 3%の性能オーバーヘッドで3秒以内のダウンタイムを実現

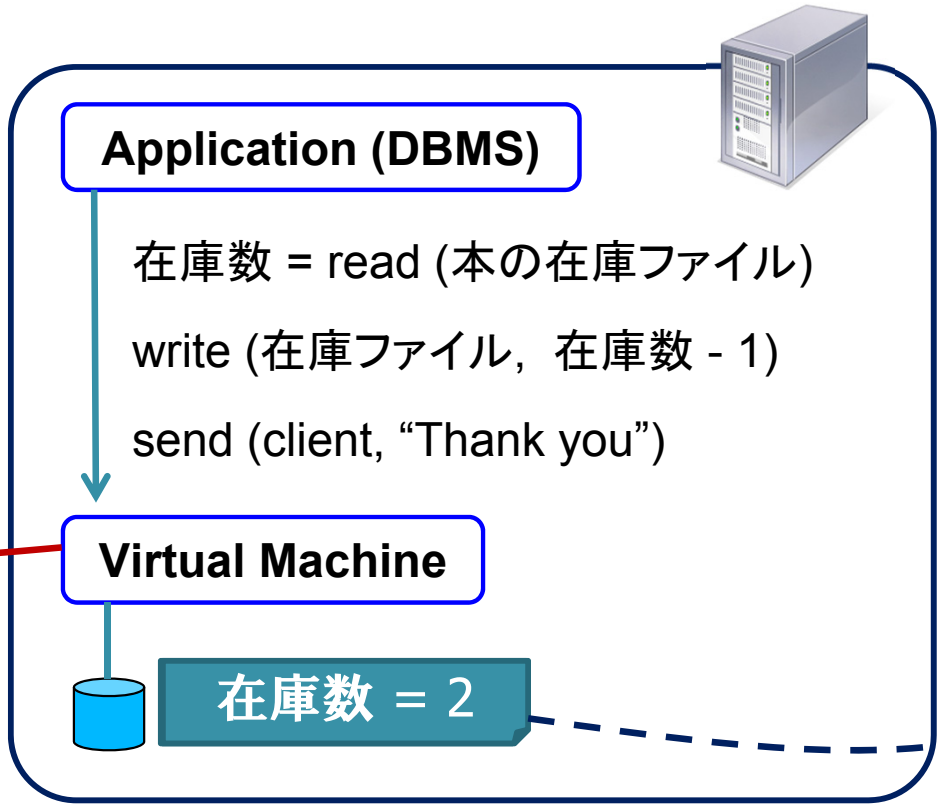
RemusDB の概念図（論文 Figure 1 を参考に作成）



チェックポイント方式の問題点とRemusのアイデア

- バックアップは最新のチェックポイントから再実行される
  - プライマリが チェックポイント後に実行した処理は失われる
  - 上の例では本が2冊売れた区間Cの情報が失われてしまう
- Remus はネットワークバッファリング機能でこの問題に対応
  - チェックポイントの反映が完了するまで，パケットをバッファリングすることで不整合な状態をユーザが観測することを防ぐ

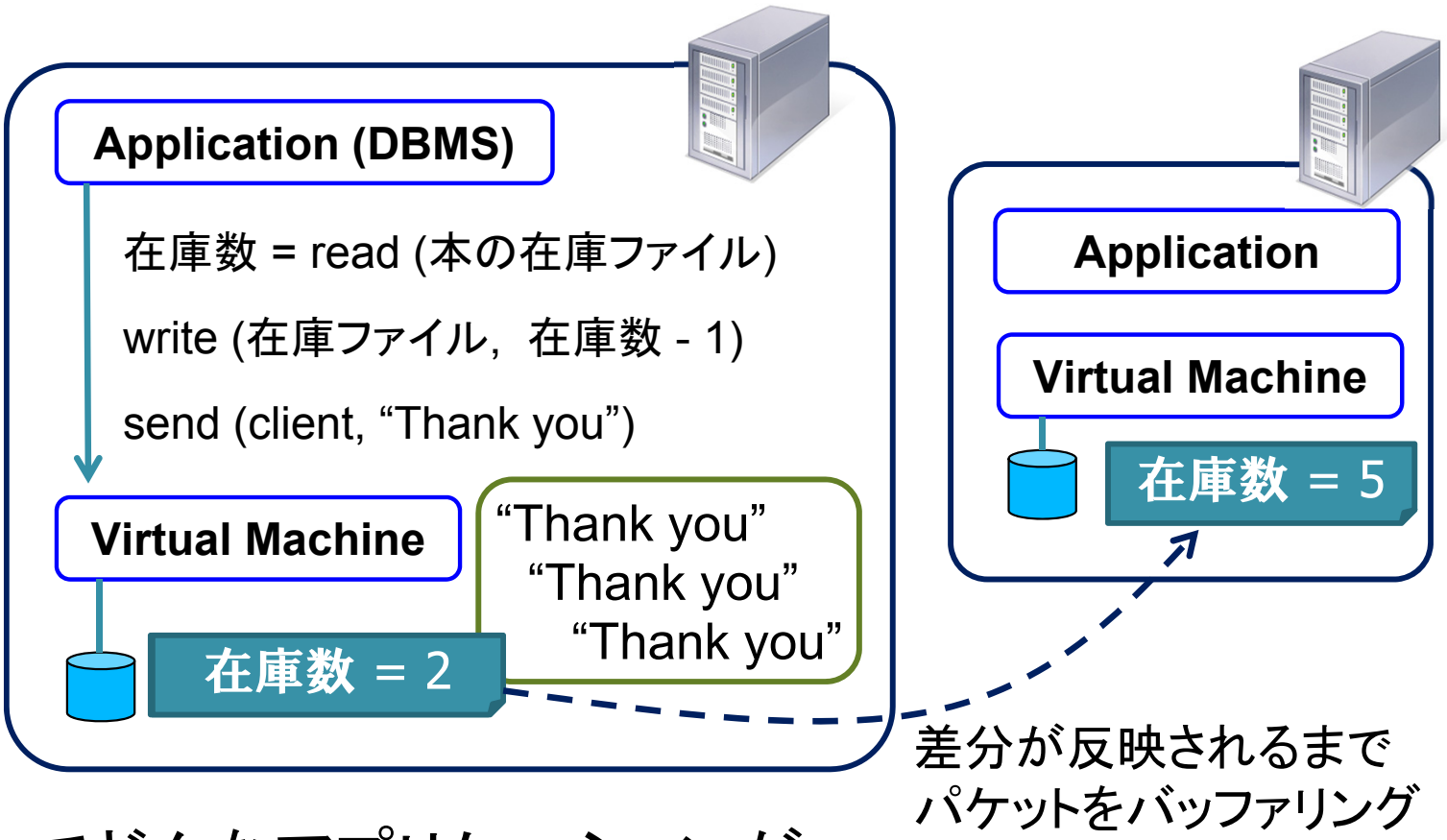
（補足解説）Remusのバッファリング機能の解説



- 仮想マシンは上でどんなアプリケーションが動いているか知らないし仮定もできない
- Remus はチェックポイントが作成されるまで，「外の世界」へのデータ送信を止めておくことで，一貫性を達成する

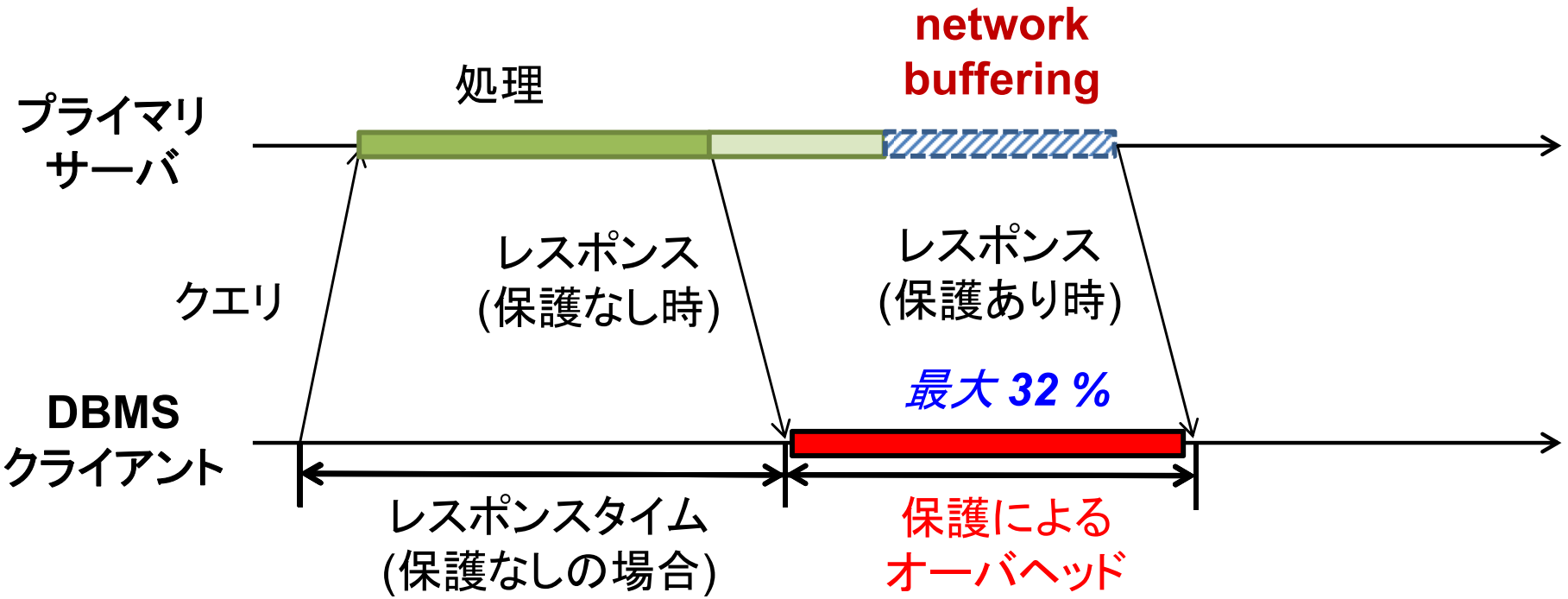
（論文3）“RemusDB: Transparent High Availability for Database Systems”

（補足解説）Remusのバッファリング機能の解説



- 仮想マシンは上でどんなアプリケーションが動いているか知らないし仮定もできない
- Remus はチェックポイントが作成されるまで，「外の世界」へのデータ送信を止めておくことで，一貫性を達成する

（論文3）“RemusDB: Transparent High Availability for Database Systems”



本論文の貢献

- データベースのワークロードのために Reumus のオーバヘッドを削減し 3% の性能オーバヘッドで 3秒以下の復旧時間を実現
- バッファも転送するのでワームアップ済みの状態から再開可能

論文で行っている最適化

メモリ最適化

- Asynchronous Checkpoint Compression (ASC)
  - LRUとランレングス圧縮を利用して転送データを圧縮
- Disk Read Tracking (RT)
  - ディスクから読み込んだデータが更新されるまで実データを送信しない

ネットワーク最適化

- Commit Protection (CP)
  - setsockopt() にオプションを追加して，DBMSがソケットのモードを選択できるようにする
    - DBMSの修正が必要なため透過性は薄れるが，100行程度の修正でよい
  - COMMITかABORTが発行されてから，Ackをクライアントに返すまでの間は保護ありに
    - それ以外は保護なし．障害が発生しても，COMMIT前のトランザクションが失われるだけだから大丈夫

Disk Read Tracking (RT)

- DBMSはディスクからのデータを Buffer Pool (BP) に読み込む
- メモリへの書き込みなため，Remusにとっては Dirtyページ
  - チェックポイントごとに転送されてしまう
- ディスクから読み出したメモリ上のデータが更新されない限り破棄可能
  - ディスクから復元するようにアノテーションしておけばよい