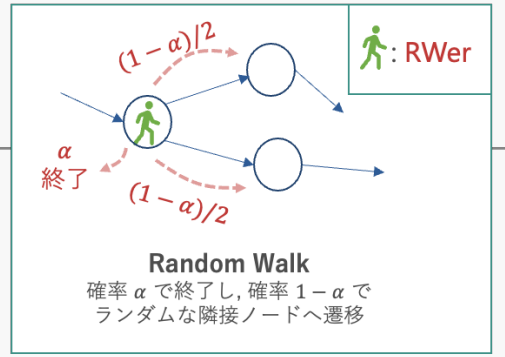
2022/8/5

Random Walker 単位の非同期処理における Random Early Detection を用いた 自律的輻輳制御機構

滝沢 駿

慶應義塾大学

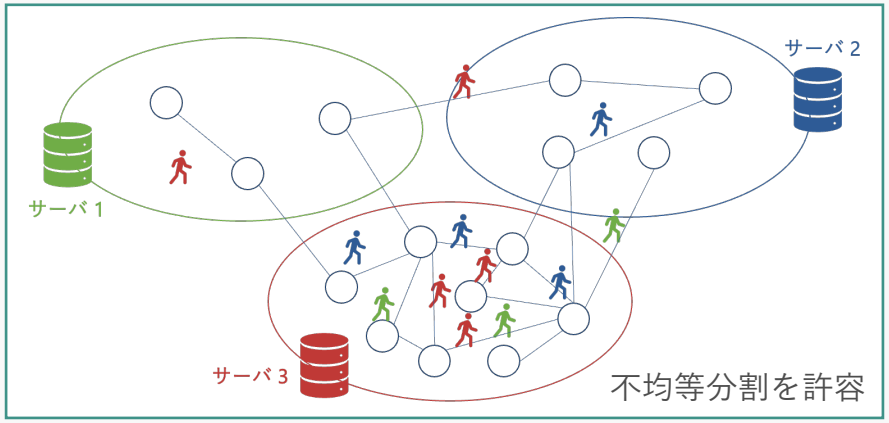
イントロダクション



- **グラフ上での Random Walk (RW) の利便性**
  - e.g., 推薦システム, コミュニティ検出, 類似性推定
- **既存手法：グラフの均等分割 & 同期時にデータ (RWer) をまとめて送信**
  - 動的グラフを想定した場合, グラフ分割は不均等になっていく（負荷分散 ×）
  - 複数ユーザによる多様なアプリケーション実行に不向き（同期タイミング ×）
- **本研究では RWer 単位での非同期処理を採用**
  - 同期待ちの解消により同期処理の弱点を克服

But...
- ✓ 一つのサーバに RWer が集中
- ✓ RWer キューのキューイング遅延が増加

  RWer キュー：サーバが保持する RWer の待ち行列

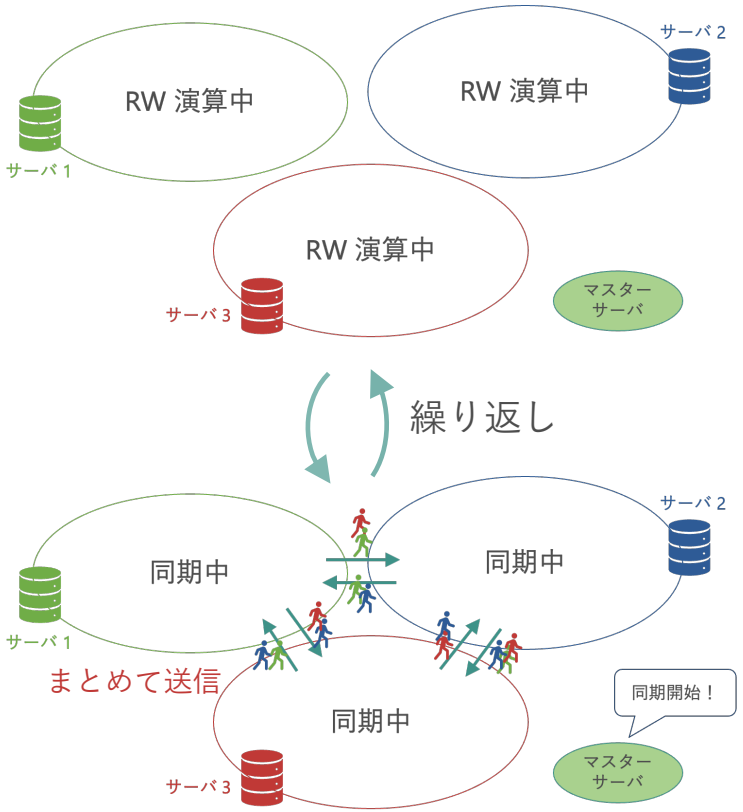


- **自律的 RWer 輻輳制御機構：各サーバが RWer の流量を自律的に調整**
  - Random Early Detection[1] を利用し, RWer キューを管理 → 輻輳の初期検知が可能に
  - RWer ロスを検知し, RWer 生成スピードを落とす

[1] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. IEEE/ACM Trans. Netw., 1(4):397–413, aug 1993.

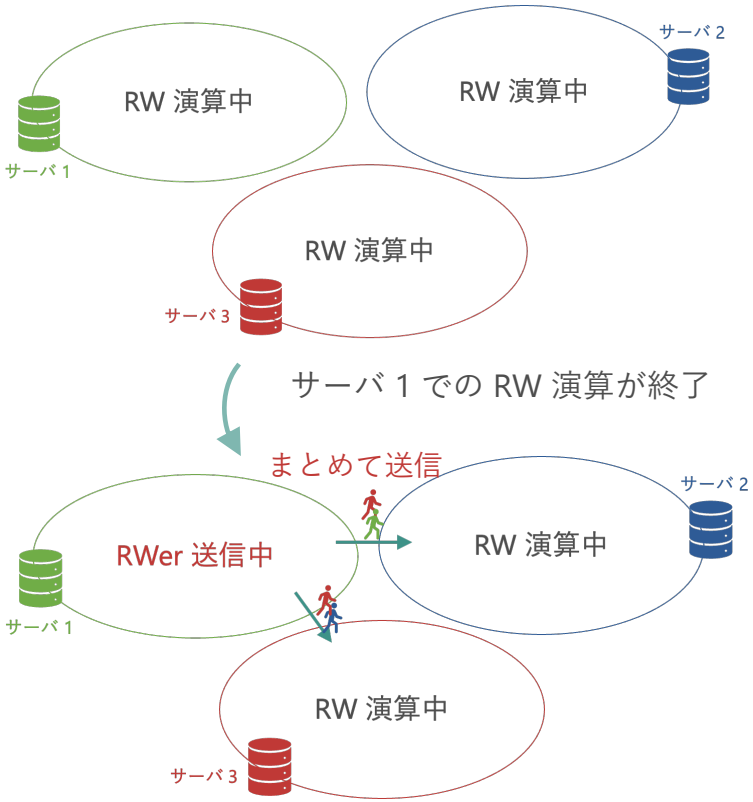
既存手法との比較

同期型[2]



- グラフの均等分割が前提
- マスターサーバが必要
- 早く演算が終わったサーバは同期まで遅いサーバを待つ必要あり

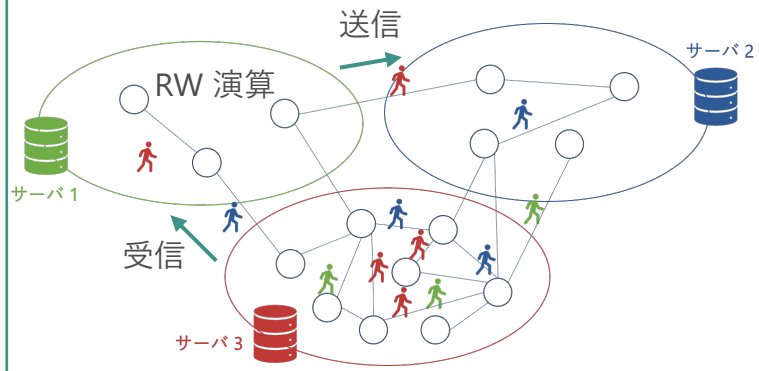
非同期型[3]



- グラフの均等分割が前提
- 早く演算が終わったサーバは他のサーバを待つことなく, 次のステップへ
  - ✓ 同期型の待ち時間の発生を防止
- 依然としてサーバ内での待ちは発生

RWer 単位での非同期型

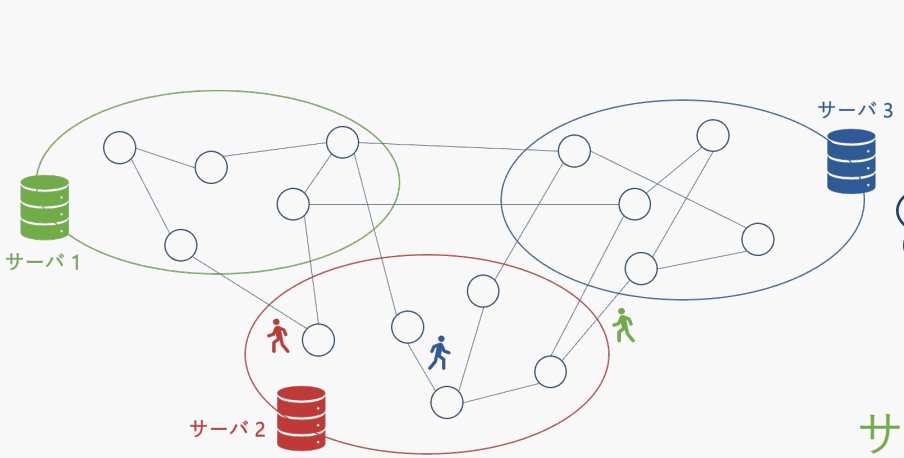
それぞれの RWer について,
独立して演算, 送受信を行う



- グラフの不均等分割を許容
- サーバ内での待ちが発生しない
- 1 RWer = 1 パケットで UDP 通信
  - ✓ RWer の独立性を活かす
  - ✓ RW ではサーバを跨ぐたびに通信が発生 → TCP では無駄が多い
  - ✓ 1 RWer のデータサイズは UDP のペイロードに十分収まる
- RWer が一箇所に集中する場合がある
  - ➢ 本研究が対処する課題

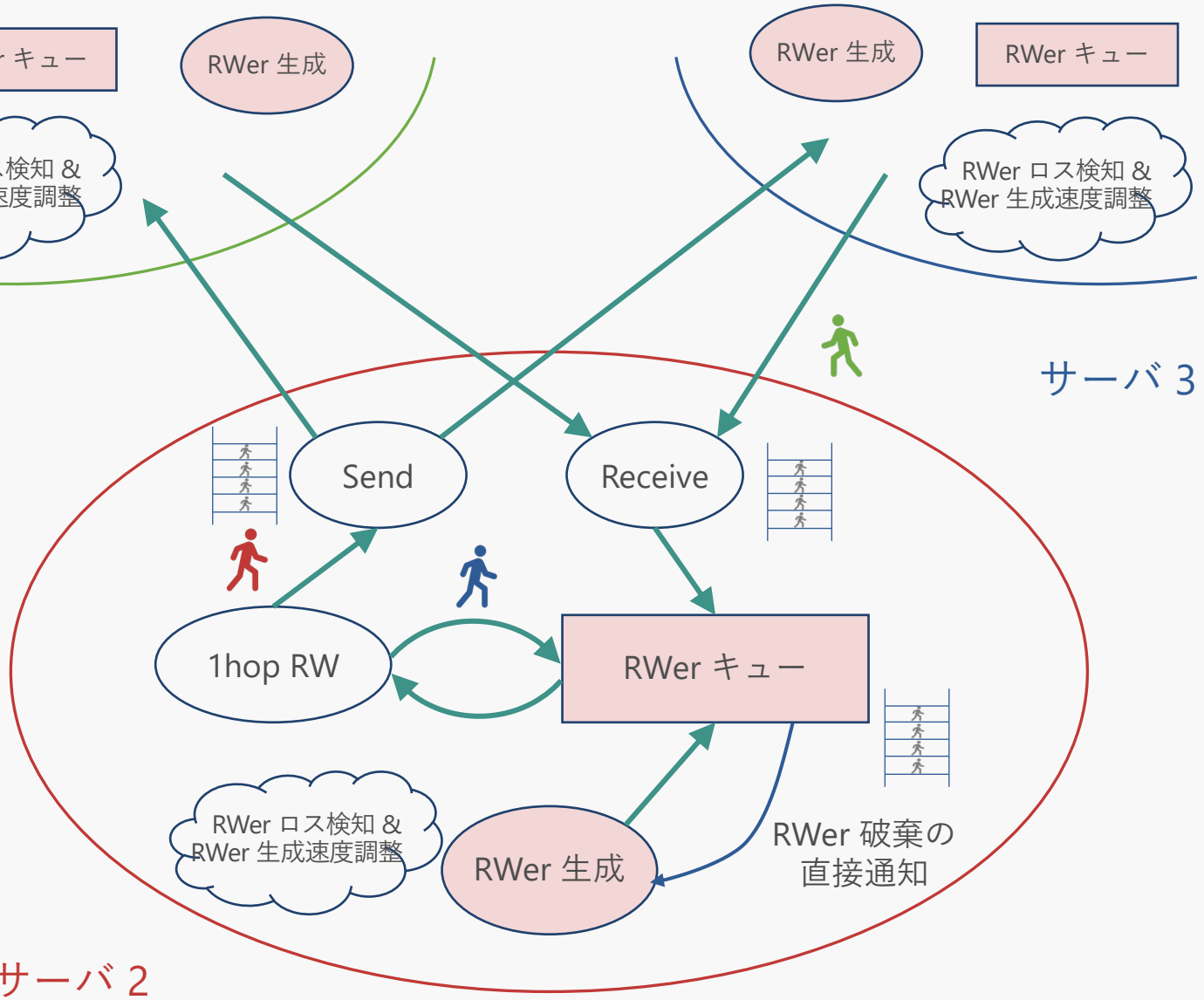
[2]Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. Knightking: A fast distributed graph random walk engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 524–537, New York, NY, USA, 2019. Association for Computing Machinery.
[3]Roshan Dathathri, Gurbinder Gill, Loc Hoang, Vishwesh Jatala, Keshav Pingali, V. Krishna Nandivada, Hoang-Vu Dang, and Marc Snir. Gluon-async: A bulk-asynchronous system for distributed and heterogeneous graph analytics. In *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 15–28, 2019.

自律的 RWer 輻輳制御機構の概要



表面上の動作

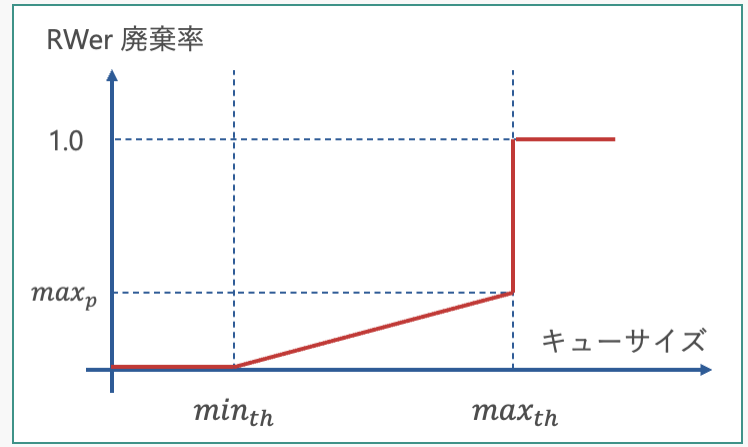
内部実装



→ : RWer の流れ

※ 終了した RWer は起点サーバへ通知

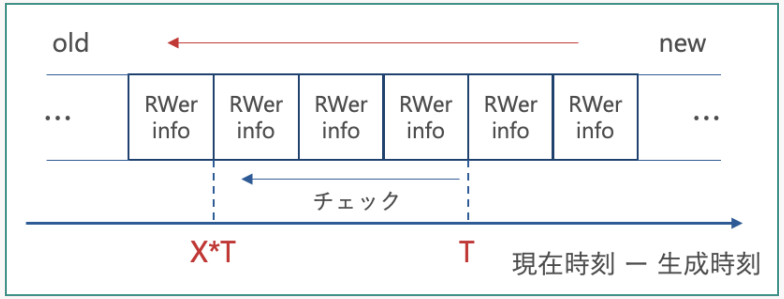
キュー長に応じて RWer の破棄確率を決定



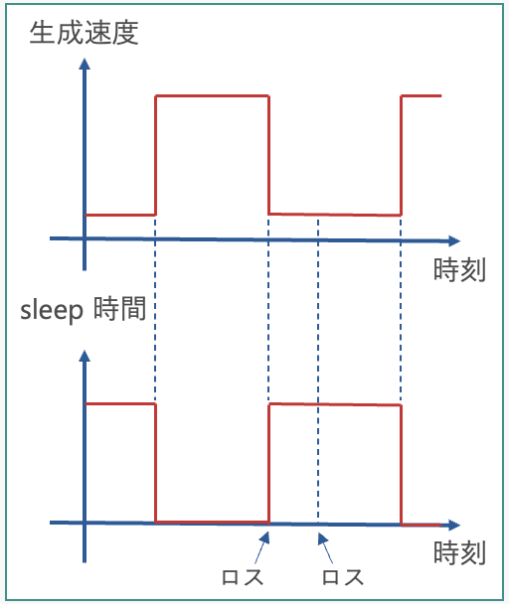
- **Random Early Detection (RED)[1] を参考に実装**
  - ルータなどで採用されているキュー管理アルゴリズム
- **テールドロップに比べ公平な制御**
  - テールドロップ：キューが満杯のときに到着した RWer のみを破棄
  - RQ では輻輳の原因となる送信者の RWer 破棄率が自ずと上昇
- **ランダムな RWer 破棄により<u>グローバル同期を防ぐ</u>**
  - グローバル同期：複数のサーバが同時に RWer の生成スピードを下げることで，再び同時に生成スピードが上昇し，輻輳が発生
- **キューが満杯になるより前に RWer を破棄し始め，RWer 生成速度調整を誘発させることにより，<u>輻輳を事前に防ぐ</u>**

[1] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. IEEE/ACM Trans. Netw., 1(4):397–413, aug 1993.

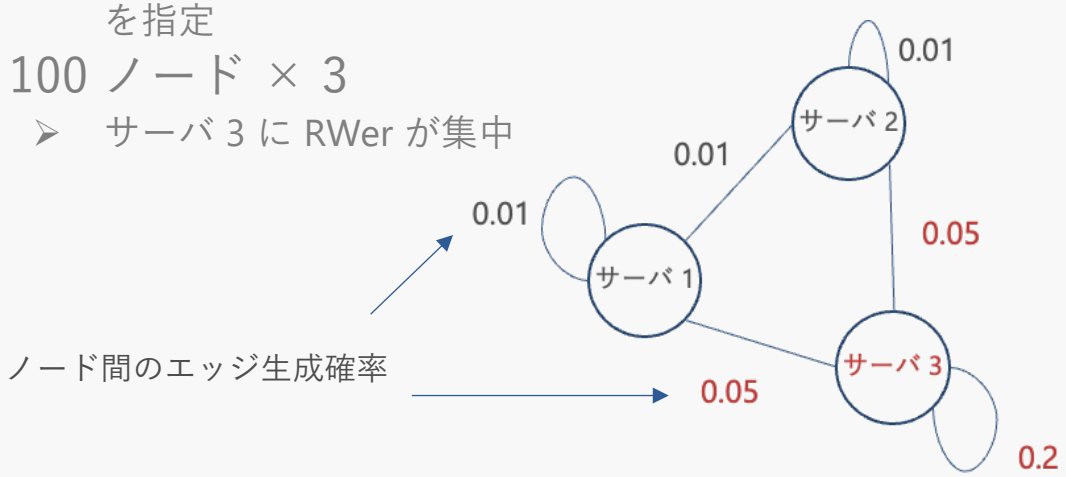
RWer ロスを検知し, RWer 生成速度を落とす



- **タイムアウト検知と直接通知による検知**
  - 生成から経った時間が一定範囲内にある RWer のうち, まだ終了確認が取れていないものをロスしたものとみなす
  - RQ で破棄した RWer の起点サーバが今いるサーバの場合, 起点ノードに直接通知し, ロスを検知させる
    - ➢ RTT の時間差なしでロスの通知が可能
- **RWer 生成速度制御**
  - x 回以上連続で ロスを検知しなかったら $sleep = 0\,ns$ それ以外は $sleep$ あり
    - ➢ 1~1000ns のスリープは制御できないので $sleep = 1\,ns$ と設定している
  - ロスを検知したらリセット



- **実装環境**
  - ① 3.5 GHz Intel Xeon 24 コア × 2 (サーバ 1, 2)
  - ② 3.4 GHz Intel Core 8 コア × 1 (サーバ 3)
  - C++ で実装
- **データセット**
  - SBM[4] で生成
    - partition 数, ノード数,
      partition 内のノード間のエッジ生成確率,
      partition 間のノード間のエッジ生成確率
      を指定
  - 100 ノード × 3
    - サーバ 3 に RWer が集中



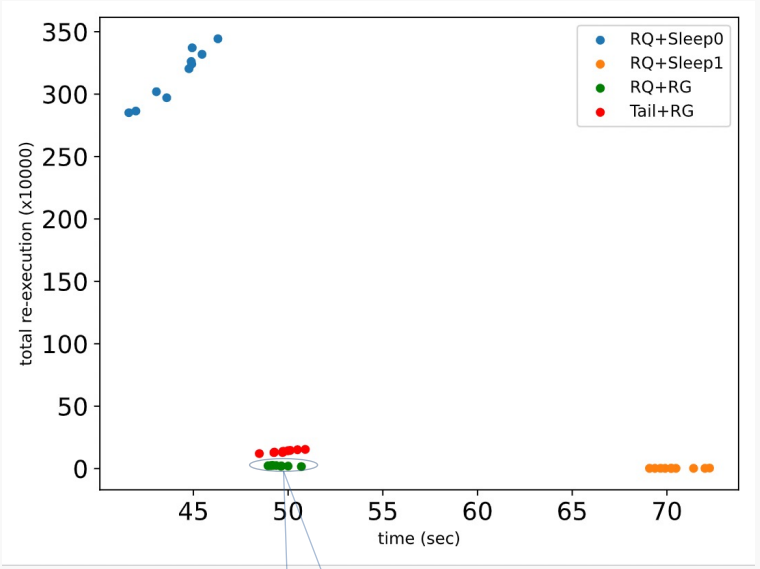
- **実験**
  1. 全ノードから 10000 回分の RW が終了するまで実行を継続 (全サーバ同時開始)
     - (1 番遅いサーバでの) 実行時間, 総追加実行回数
       - ✓ RQ + sleep0 (スリープなし)
       - ✓ RQ + sleep1 (毎回 1ns のスリープ)
       - ✓ Tail (テールドロップ) + RG
       - ✓ RQ + RG
       - ✓ RQ + RG (起点サーバでの直接通知なし)
  2. 全ノードから 100000 回分の RWer を生成 (〃)
     - キューの状態とドロップした場所 (サーバ 3)
       - ✓ RQ + RG, Tail (テールドロップ) + RG
  3. グラフトポロジを変化させながら実行
     - 総追加実行回数
       - ✓ RQ + RG, Tail (テールドロップ) + RG
  - ◆ パラメタ
    - RQ : $max_p = 0.1, min_{th} = 5, max_{th} = 100$
    - RG : x = 3
    - テールドロップ : $max_{th} = 100$

[4] Paul W. Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. Stochastic blockmodels: First steps. Vol. 5, pp. 109–137, 1983.

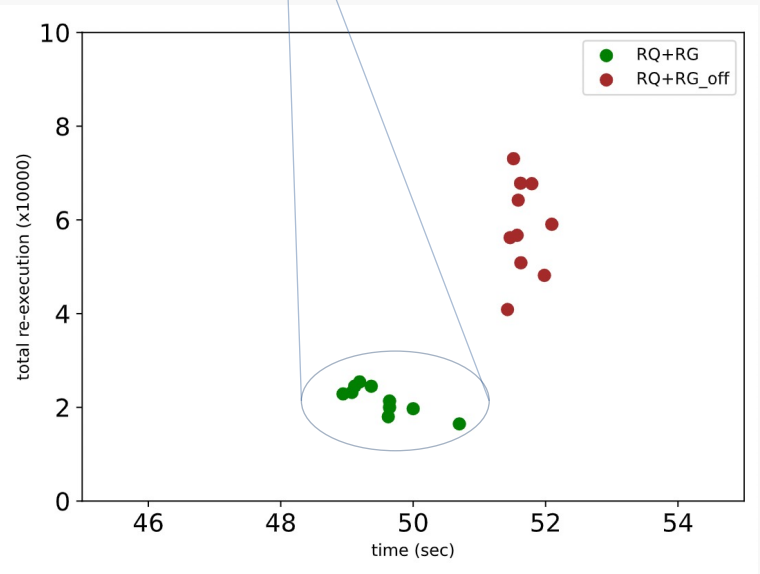
実験 : 全ノードから 10000 回分の RW が終了するまで実行を継続

RQ + RG : 高スループット / 低 RWer ロス **を達成**

手法 (RQ + RG との比較)	実行時間	総追加実行数
RQ + RG	49 秒 (基準値)	20000 回 (基準値)
RQ + sleep0 (と比べて)	+ 5 秒	- 3,130,000 回
RQ + sleep1 (〃)	- 20 秒	+ 20,000 回
テールドロップ + RG (〃)	- 1 秒	- 120,000 回
RQ + RG (直接通知なし) (〃)	- 2 秒	- 40,000 回



直接通知ありでの比較

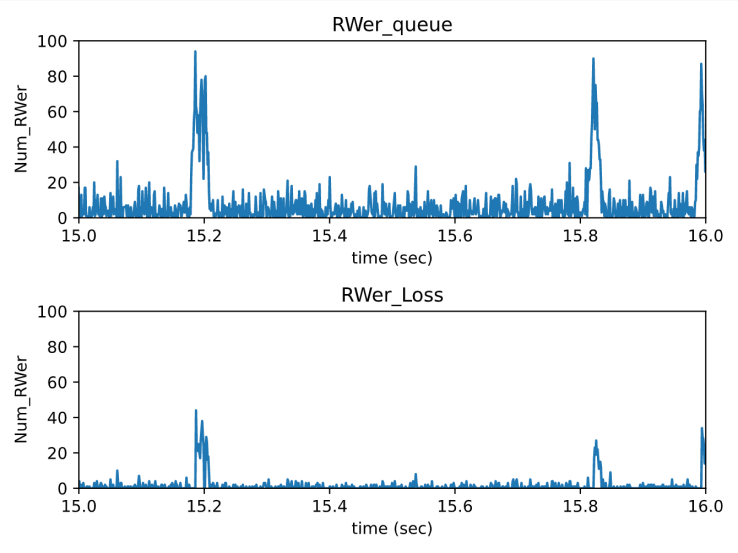


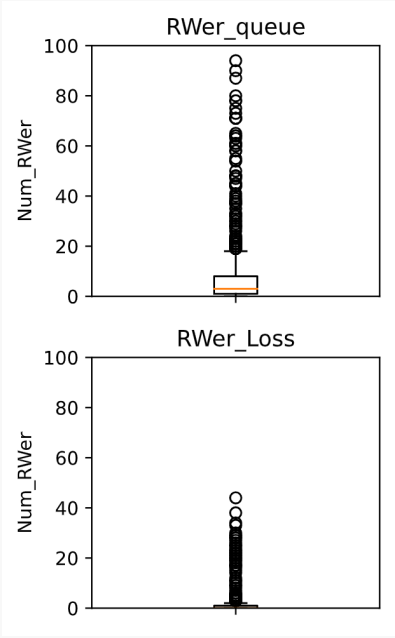
直接通知の有無の比較

実験 : 全ノードから 100000 回分の RWer を生成

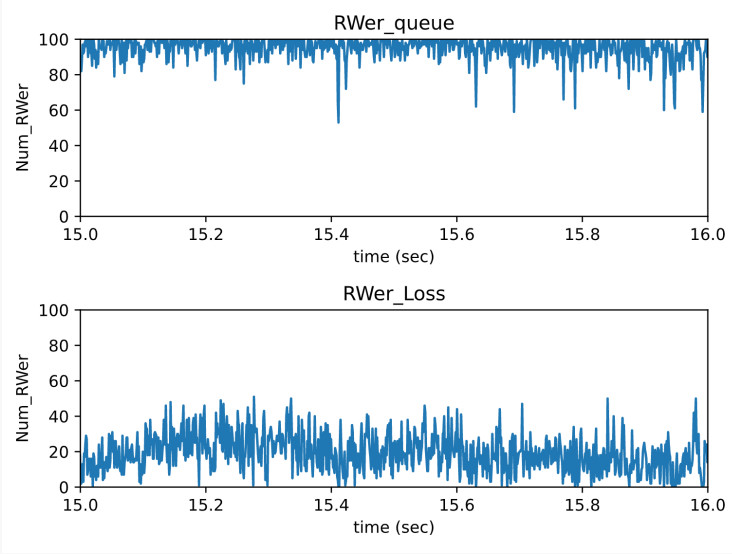
RQ はテールドロップに比べ,
- **キューの平均長が小さい**
- **RWerロスが少ない**

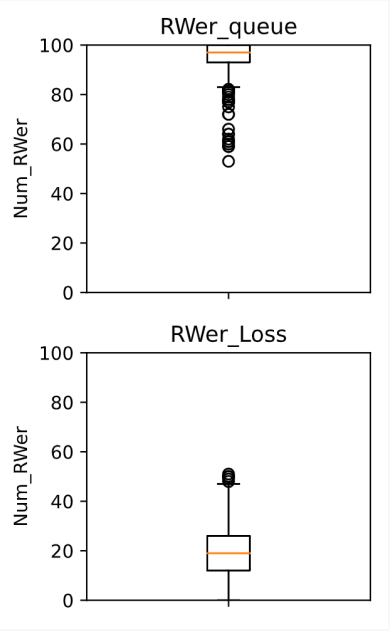
- **RQ はキューが満杯になる前から RWer を破棄し始めるので輻輳を未然に検知し, 防ぐことができている**





RQ + RG



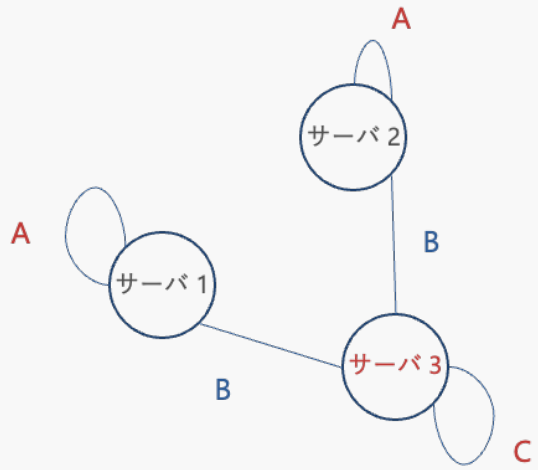


テールドロップ + RG

RQ はテールドロップよりも負荷耐性が高い

- **RQ はテールドロップに比べグラフトポロジーに対する柔軟性が高い**
  - B に対し C が大きくなるとロスが減るのは, サーバ 3 に存在する
    サーバ 3 で生成された RWer が増えることにより,
    直接通知による RWer ロス検知の効果が上昇するから

それぞれのサーバが 1000 ノードを保持



RQ + RG 「追加実行回数 (×10000)」 テールドロップ + RG

A : B \ B : C	1 : 2	1 : 5	1 : 10
1 : 2	2.0	3.2	2.3
1 : 5	8.9	7.1	3.3
1 : 10	14.6	6.2	3.7

A : B \ B : C	1 : 2	1 : 5	1 : 10
1 : 2	13.6	17.6	17.3
1 : 5	21.8	22.2	22.5
1 : 10	25.6	24.1	26.1

自律的 RWer 輻輳制御機構：RWer 単位の非同期処理における輻輳回避システム

- **RQ：キューが満杯になる前から RWer をキュー長に基づいて確率的に破棄**
  - 輻輳を未然に検知
  - キューの安定化
- **RG：RWer ロスを検知し，RWer 生成速度を調整**
  - タイムアウトによる検知と直接通知による検知
  - RWer 生成間のスリープによって RWer 生成速度を調整
- **高スループットかつ低 RWer ロスを達成**
  - RWer 生成間のスリープをなくした場合に比べ，約 5 秒遅いが，総追加実行数が約 3,130,000 回少ない
  - RWer キュー管理をテールドロップにした場合と比べ，約 1 秒早く，総追加実行数が約 120,000 回少ない