非集中型クラウドストレージのスケーラビリティ評価

奥寺昇平[†] 中村俊介[†]
長尾洋也[†] 首藤一幸[†]

非集中なクラウドストレージにおいて、任意のノードからデータを保持する担当ノードにリクエストを到達させるためには、各ノードが他のノードを把握する必要がある。特に、クライアントが接続したノードから担当ノードに直接リクエストを送るクラウドストレージでは、全ノードがシステム全体の最新の状態を保持する必要があり、システムの整合性を保つことが難しい。そこで、gossip プロトコルをベースとしたメンバーシップ管理により効率よく通信を行うことが可能である。一方、このようなメンバーシップ管理もスケーラビリティを制約する要因の一つとなりうる。この管理方法では、すべてのノードで定期的に通信が発生するので、ノード台数が増えるにつれ、総通信量が増えるのである。よって、フロントエンド (例えばストレージであれば、データの読み書き処理。) の処理効率つまり、アベイラビリティを下げると考えられる。しかしながら、非集中型のクラウドストレージにおいて、この管理を行う処理がどれくらいの通信負荷をもたらすのかといったことは知られていない。そこで本研究では、gossip プロトコルを用いるクラウドストレージ Cassandra を対象として、ノード台数に応じてシステム全体の通信負荷がどのように変化するのかを計測・考察する。計測の結果、システム全体で発生する gossip の通信量は、ノード台数を n としたとき、$O(n^2)$ となることを確認した。定量的にクラウドストレージの gossip の通信量を計測できた。

A Scalability Study of A Decentralized Cloud Storage

Shohei Okudera,[†] Shunsuke Nakamura,[†] Hiroya Nagao[†]
and Kazuyuki Shudo[†]

Abstract in English

1. はじめに

分散システムには高い可用性が必要とされる。つまり、システムの可動性、つまり正常なサービスを停止することなく提供できなければならない。特にクラウドのような大規模な環境においては、急激に増加する負荷やネットワーク分断や部分故障といった問題に対しても迅速に対応し、常に稼動し続けるな安定性・信頼性が求められる。

分散システムの可用性を確保する戦略[1)] としてスケールアップ、スケールアウトの二つの戦略がある。スケールアップとはシステムを構成する各マシン (の CPU やメモリ) をの高機能化、高性能化を図ることで、システム全体の性能をスケールさせることを目的とした戦略である。従来はこの戦略でシステムの性能向上が図られてきたが、費用に見合った性能向上には限界がある。

一方、スケールアウトとは、マシン一台あたりの性能を上げるのではなく、安価な商用マシンを並べて並列処理を行うことで、可用性を確保するというと戦略である。スケールアウトでは、一台あたりのマシンが安いため、スケールアップ戦略をとった場合と比べて、総合的にコストパフォーマンスがよい。その一方で、システム全体のノードのどれかが常に故障頻度が高いという問題がある。スケールアウトによるスケーラビリティ確保の戦略では、マシンの数が増加するほど、システムでマシンの故障が発生する確率は増加する。つまり平均故障間隔が非常に短くなってしまうのである。そこで、ハードは常に故障するということを前提としたシステムの仕組みづくりが必要となる。

故障を前提とした分散システムで特に重要な機能は、故障したマシンに接続する他のマシンから故障を検知する仕組みである。システムの整合性を保つためには、各マシンが検知するだけでなく、互いにその故障検知の情報を通信しあうことで、システム全体としての合意を取らなければならない。一方で、特に非集中なクラウドストレージにおいては、故障検知にネットワーク、CPU などのリソース消費を控えばければならない。非集中なクラウドストレージでは、データの読み書きがメインのタスクであるからである。非集中なクラウドストレージの代表的な故障検知に gossip

† 東京工業大学
Tokyo Institute of Technology

プロトコルが挙げられる。しかしながら、非集中なクラウドストレージにおいて、この管理を行う処理がどれくらいの通信負荷をもたらすのかといったことは知られていない。そこで本研究では、gossip がクラウドストレージでもたらす通信負荷を評価した。

本章の構成は以下のとおりである。２章で研究背景として、故障の伝播について説明し、３章で関連研究として、クラウド環境で gossip プロトコルの評価[2] について触れる４章で故障検知の代表的なアルゴリズムである gossip プロトコルに焦点を当てる。５章で非集中なクラウドストレージの一つである Cassandra[3] について説明し、６章で Cassandra の軽量化、測定法について説明する。７章で通信量測定の実験、評価を行い、８章で論文をまとめる。

2. 研究背景

故障した情報を他のノードに伝搬する方法には、すべてのノードに直接伝搬を行う全対全型、近接するノードにだけに伝える近傍型、ランダムで選んだノードだけ伝達するランダム型がある。全対型は、故障を伝搬する時間は早いが、すべてのノードと通信が発生するのでスケーラビリティを確保できない。また近傍型では、ネットワークの局所性を意識するので、通信のパフォーマンスは改善される一方で、新規ノードの追加、離脱時に近傍ノードを再設定しなければならないことやネットワーク情報を管理しなければならないという欠点がある。ランダム型の代表的な伝搬アルゴリズムには、gossip プロトコルが挙げられる。gossip プロトコルとは、ソーシャルネットワークで見られる噂 (ゴシップ) の伝搬をモデルとしたアルゴリズムである。gossip プロトコルを利用して、検知した故障の情報を他のマシンに効率よく伝達することが可能である。近年 Cassandra をはじめとしたクラウドストレージの故障検知に gossip プロトコルを応用したアルゴリズムが採用されており、メンバーシップ管理が行われている。しかしながら、故障検知はメインのプロセスではない。特にクラウドストレージにおいてのメインのプロセスは、データの読み書きを行うことであり、故障検知はあくまでバックグラウンドのプロセスとして扱い、メインのプロセスに影響を与えてはならない。すなわち、故障検知は計算資源や使用するネットワークバンド幅が限られている環境で実現されることが求められる。これらの問題を解決するために、gossip を応用したさまざまなプロトコルが提案されている。例えば [参照 1] では、クラウド環境を想定して、MTBF が非常に短いアンダーレイネットワークを考慮したオーバレイネットワークを構築し、局所的な通信が多く発生するように工夫した。ネットワークリソース資源の消費を押さえて故障検知を行っている。しかしながら、非集中な環境については、○○である。

我々は、クラウドストレージに適用されている gossip プロトコルにおいて通信量がノード数に対してどのように増加していくのかを実測し、特性について評価と考察を行った。また、実アプリである YCSB を起動してデータ読み書きを行った時の通信量のスケーラビリティについても実験・考察を行った。

3. 関連研究

関連研究として、クラウド環境で gossip プロトコルをベースにしたマルチキャストを提案し評価した論文について紹介する。一般にグローバルで接続されたデータセンターから構成されるクラウドコンピューティングのインフラで起きているようなコストが高くリソース制約があるリンクが存在する。基本的な gossip プロトコルは、剛健で信頼性のあるデータ伝搬に対してスケールするアプロ地である一方、このようなクラウド環境においては、その冗長性がから、リンクとノードに高いリソース消費を引き起こす。そこで著者は、局所的なリンクを優先するオーバレイを構築し、局所を考慮して伝搬を行うことで、制約のあるリンク間でのトラフィックを減少させることができた。

本研究との関連は、分散システムで実ノードを立ち上げて gossip プロトコルベースのトラフィックを評価しているところである。しかしながら、我々は非集中なクラウドストレージに焦点を当てること、またマルチキャストではなく我々は故障検知を目的にしている点が異なる。

4. gossip プロトコル

gossip プロトコルとは、ソーシャルネットワークで見られる噂 (ゴシップ) の伝搬をモデルとしたアルゴリズムである。gossip プロトコルを利用して、検知した故障の情報を他のマシンに効率よく伝達することが可能である。ノード p、q が存在し両者で情報交換を行うときと考える。gossip プロトコルを情報の伝搬方法によって三つのカテゴリーに分類することができる。[4]

- A. Pull 型のアプローチ
  ノード p が持っている情報をノード q に送信し、ノード q の情報を更新する。
- Push 型のアプローチ
  ノード p はノード q に情報のダイジェストを送信する。ダイジェストを受け取ったノード q は必要な更新箇所だけをノード p に送り返し、ノード p は更新は行う。
- Pull&Push 型のアプローチ
  pull 型の gossip と似ている。違いは、ノード q がノード p に更新箇所を送るときに、ノード q のダイジェストを知らせることである。ノード p はダイジェストを見て、ノード q の情報も更新さ

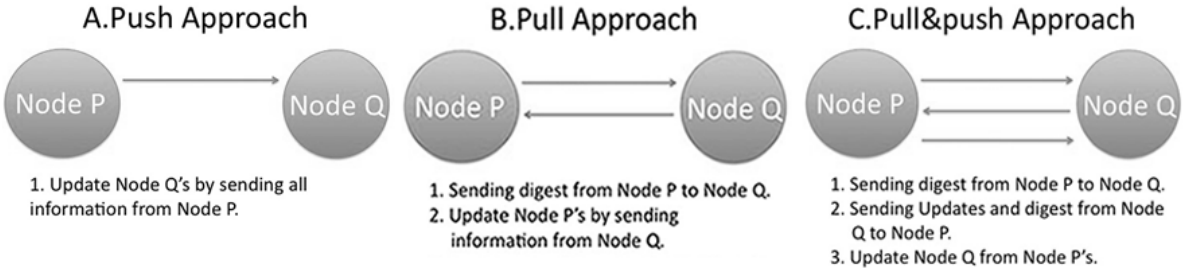


図 1 gossip プロトコル

せる。

ここでは、Pull&Push 型を応用したプロトコル、Scuttlebutt[5] について詳しく説明する。

Scuttlebutt とは、分散システムでイベンチュアリーコンシステンシーに各メンバーが保持する情報を融合をするためのプロトコルの一つである。Pull&Push 型アプローチを採用している。特に、使用できるネットワーク帯域幅、CPU サイクルが限定されているときを想定している。つまり、使用するネットワーク帯域幅を抑えるためにある MTU(Message Transfer Unit) が決められていると仮定して、MTU のサイズの中で情報を融合できるように工夫し送信を行うのである。具体的には、更新する情報に優先順位をつけて必要な情報だけを相手に伝搬させることで、システムの整合性を保ちつづ、ネットワーク帯域幅の消費を抑えた情報融合を可能にする。通信手順について説明する。メンバーの集合を $P = \{p, q, ...\}$ とし、各メンバーがマッピング $\sigma \in S = K \to (V \times N)$ にモデル化された状態を保持している。ここで、K はキーのセット、V は値のセット、N は有限の順序付けられたバージョン番号の集合を表している。$\sigma(k) = (v, n)$ は、キー k が値 v と、バージョン番号 v にマップすることを意味している。　また最近のマッピングほどバージョン番号が大きい。メンバーの状態は変更があり、またすべてのノードで複製化され保持している状況を考える。そこでノード p が保持する各メンバーのマッピングを u_p:P → S とモデル化する。

メンバーの状態の更新を考える。メンバー p は、$u_p(p)$ に対する更新のみが行えて、$u_q(p), p \neq q$ に対しては gossip を通じて間接的にしか更新することができない。Scuttlebutt においての $u_p(p)$ に対する更新では、一度に一つのキーに対する更新のみが可能であり、更新したキーに対するバージョンは、 p が保持するマッピングの中のバージョン $(max(\sigma_p))$ よりも大きな値に更新する規則がある。メンバー p とメンバー q における更新では、$u_p(r)$ に対する更新つまり融合では、バージョンが大きいキーの値で更新される。$\sigma_1(k) = (v_1, n_1)$ と $\sigma_2(k) = (v_2, n_2)$ が融合し、σ が生成されるとする。$n_1 > n_2$ であれば $\sigma = (v_1, n_1)$、そうでなければ $\sigma = (v_2, n_2)$ となる。ここで、ある特定のメンバー p が持つキー k, 値 v, バージョン n のタプルを $delta$ を名付ける。

Scuttlebutt では以下のように通信が行われる。

(1) メンバー p が保持する各メンバーについて mapping の中で一番新しいバージョンだけ $(\{(r, max(u_p(r))|r \in P)\})$ をメンバー q に送信する。

(2) 同様に、メンバー q が保持する各メンバーについて mapping の中で一番新しいバージョンだけ $(\{(r, max(u_q(r))|r \in Q)\})$ をメンバー p に送信する。

(3) 上のようなダイジェストを受信したメンバー q は、下で定義している $\Delta^{p \to q}_{scuttle}$ をメンバー q に送信する。

(4) 3. と同様に、メンバー p から $\Delta^{q \to p}_{scuttle}$ をメンバー p に送信する。

$$\Delta^{p \to q}_{scuttle} = \{(r, k, v, n)|u_p(r)(k) = (v, n) \wedge n > max(u_q(r)))\}$$

すなわち、$\Delta^{p \to q}_{scuttle} = \{(r, k, v, n)|u_p(r)(k) = (v, n)$ とは、あるメンバ r についてメンバー q がまだ更新していないキーの $delta$ ということである。

このようにはじめに、各メンバーに対する最大のバージョン番号をダイジェストとして通信することで、無駄な通信を省略することができる。さらに MTU が存在するときには、手順 3. 手順 4. ですべての更新情報を送信することはできない。そこでバージョンが小さい $deltas$ に優先順位をつけて更新の選択を決定することになる。この選択により、システムの整合性を保ちつつ効率的な通信が可能になる。

5. Cassandra

Apache Cassndra は、Facebook 社によって開発され、Apache Project としてオープンソース化されたクラウドストレージである。[6] 複数のデータセンタ上に分散して配置された数百ノードで構成されることを想定しており、高い可用性と単一故障点を持たない非集中な分散モデルが大きな特徴である。このような大規模な環境では平均故障間隔は非常に短く、故障検知

が重要である。

5.1 メンバーシップ管理

Cassandra を構成する各ノードは、リング上 ID 空間に配置される。各ノードは、クラスタに参加しているすべてのメンバーを gossip により把握する。また、新規ノード追加時には、指揮するノードに予めシステム内の最初にコンタクトを取るノード (これを seed と呼ぶ) の情報を設定しておき、そのメンバーに新規に参加することを伝えた後で、クラスタに加わる。メンバーシップ管理は、上述した gossip プロトコルである Scuttlebutt をベースに行っている。ここでは、Scuttlebutt を使用した情報交換を gossip 通信とする。具体的には、以下の手順に沿って毎秒情報交換が行われる。

(1) STEP1:もし他の生存しているノードが存在したら、ランダムで生存しているノードを選択してゴシップ通信を行う。
(2) STEP2:生存ノードと到達できないノードの数に応じたある確率によって、ランダムで到達できないノードに gossip 通信を行う。
(3) STEP3:STEP1 でゴシップ通信を行ったノードが seed ではなかったとき、あるいは seed ノードの数より生存しているノードが少ないときには、生存ノード、seed, 到達不可のノードの数に応じたある確率でランダムで seed に gossip 通信を行う。

Cassandra ノードはこの方式で毎秒 1-3 回の gossip 通信を行っている。

6. Cassandra の軽量化と測定手法

6.1 Cassandra の軽量化

実験にあたって物理リソースの都合上、1 台あたり複数の Cassandra ノード を起動する必要があった。デフォルトの設定では、1 ノードの Cassandra 起動するためにデータを全く保持していない状態で、スレッド数が 130、メモリー使用領域が 120M 程度とリソースを多く消費する。また複数ノードでクラスタを構成した時にさらにリソースが消費される。リソースの消費を抑えるために、Cassandra データ保持部分のプログラムの改変と、設定パラメータのチューニングを行った。

6.1.1 プログラムの改変

Cassandra は、データを保持していない状態であってもシステム管理のためのテーブルを保持する必要があり、メモリー使用量域がかさむ。この点を踏まえメモリー使用領域を減らすためにプログラムに改変を加える。

gossip プロトコルの通信量を測定する実験では、実データがどのようなものかは関係がない。そこで、実際のデータを保存するのではなくデータサイズだけを保管するように変更した。

6.1.2 パラメータの調整

- JVM 最大ヒープサイズの制限を変更
  Cassandra は JVM 上で動作する。多数台起動するために、JVM 最大ヒープサイズの制限を 1G から 160M に変更した。
- 設定ファイルのパラメータ調整
  設定ファイルでにて、同時読み込みを許す最大値、同時書き込みを許す最大値を制限することでスレッド数を減らした。

また、これらのパラメータの調整は、gossip プロトコルの通信量を測定する実験では、直接関わらないパラメータであることに留意する。この調整により、1 マシンあたり Cassandra を最大 65 ノードまで起動することができた。

6.2 測 定 手 法

非集中型クラウドストレージのスケーラビリティ評価するために以下の実験を行った。

- 実験 1:gossip プロトコルの通信量の測定
- 実験 2:データ読み書きの通信量の測定

6.2.1 実験 1:gossip の通信量の測定

実験 1 では gossip の通信量の測定を行った。実験シナリオは、マスターとなるマシンを 1 台とワーカーとなるマシンを 1 0 台を用意した。ワーカーマシンを Node1,Node2,...Node20 と名付ける。マスターの役割は、通信量計測の開始・終了、Cassandra ノードの起動, 計測した記録の解析をワーカーに指示すること、最終的な通信量の推定を行うである。一方、ワーカーの役割は、通信量の計測、Cassandra ノー ドを起動すること、通信量の解析である。また、1 台あたり複数の Cassandra ノードを立ち上げる必要がある。Cassandra ノードの立ち上げ方は、30 秒ごとに、1 台あたり 10 ノードの Cassandra を一度に起動し、これを目指す台数に達成するまで続ける。最初の Cassandra ノードを起動した瞬間から各マシンで 10 分間の通信量を計測した。

計測後に各マシンで通信量を解析し、マスターとなるマシンに解析結果を送信する。マスターは、送られて通信量から合計値を出し、Cassandra で発生する通信量の推定を行う。

マスター、ワーカーで実行するプログラムは、シェルスクリプトでプログラムを書き、各ワーカーへの指示は、GXP を使用して制御した。GXP とは、同じコマンドを多数のホストで並列に実行するためのジョブスケジューラのことである。また、パケット情報の解析には、java,R, シェルスクリプトを使った。

6.2.2 実験 2:データ読み書きの通信量の測定

実験 2 ではアプリケーションを用いてデータの読み書きを行ったときの通信量の測定を行った。実験シ

ナリオは、マスターとなるマシンを 1 台とワーカーとなるマシンを 5 台を用意し、実験 1 と同様の手順で Cassandra の起動、通信量の測定を行った。
データを読み書きするアプリケーションには、Yahoo! Cloud Serving Benchmark(YCSB) を使用した。YCSB は Yahoo! Research が開発したクラウドストレージ用のベンチマークのことである。

6.2.3 計測方法について

実験 1 、実験 2 の計測にあたっていくつか工夫した共通の点を紹介する。

- ユーザー, プロセスのシステムリソース制約を外す
  通常のオペレーションでは、1 ユーザに共有のシステムリソースを占有されないように管理されている。具体的には、1 ユーザが同時に実行出来るプロセス数、ファイル・ディスクリプタの数や、ユーザーが実行するプロセスにおいて、仮想メモリーの使用領域、物理メモリーの使用領域などが制限される。
  上述したように Cassandra ノードの実行には、メモリー使用量域が大きく、使用するスレッド数が多い。我々の実験環境では、数台のクラスタを構成するだけで 1 ノードあたりスレッド数が 130 程度必要であり、クラスタを構成するノード数が増加するとさらに比例していく。linux のデフォルトの設定では、1 ユーザが同時に実行出来るプロセス数は 1024 であるので、Cassandra ノードを 7 台までしか起動できなかった。そこで、linux のユーザーリソースを決める設定ファイルを編集し、1 ユーザーのリソース制限、1 プロセスのリソース制限を緩和した。その結果、多数台の起動が可能になった。
- IP エイリアシングを利用し、プライベートネットワークを構築
  Cassandra のメンバー管理では、IP アドレスでメンバーを認識する。つまり、今回の実験ように 1 マシンあたり複数ノードの Cassandra を立ち上げようとすると、不都合が生じる。
  そこで、IP エイリアシングを使用して仮想アドレスを作成し、Cassandra ノードごとに割り振ることにした。その結果、同じマシン上に立ち上がった Cassandra マシン同士の差異が明らかになり、不整合が起きなくなった。
  さらに、通信量の測定の際にはノイズを防がないといけない。ノイズとは、Cassandra ノード以外から要求されるリクエストのことである。具体的には、ARP や ssh などのパケットのことである。これらのパケットを誤って計測してしまうことを避けるために、IP エイリアスリングを行うと同時に、プライベートネットワークを構築した。このネットワークに参加しているのは、Cassandra ノードだけである。よって、プライベートネットワーク内で飛び交うパケットのみを取得することが簡単にできる。
  具体的には、10.20.0.0/16 のネットワークを構築した。さらに Cassandra ノードが物理的にどのマシン上で起動しているかを判別しているために、実マシン (Node11,Node12,...,Node20) 上のノード番号 n を使用し、Node[n] 上で起動するマシンに仮想的に 10.20.n.0/24 なるサブネットを設けた。例えば、実マシン Node19 で起動する Cassandra ノードに割り当てられる仮想アドレスは、10.20.19.x ($1 < x < 254$) となる。
- tcpdump の使用
  通信量の測定は、tcpdump を使用した。上述したように、Cassandra ノード同士のやりとりはプライベートネットワーク上で行われるので、このネットワークをまたぐすべての TCP パケットのサイズを記録すればよい。具体的には、tcpdump に以下のオプションをつけて実行した。取得するパケットは二つの条件でフィルターをかけた。
  - `src net 10.20.0.0/16`
    このフィルターにより、指定したデバイスを経由したパケットのうち、送信元が 10.20.0.0/16 のネットワークであるパケットのみが取得する。つまり、これで、このネットワーク以外のパケットを取得しないことになる。この条件で、余計な ARP クエリなどを弾ける。
  - `dst net 10.20.(マシン番号).0/24`
    このフィルターにより、指定したデバイスを経由したパケットのうち、受信元が 10.20.(マシン番号).0/16 のネットワークであるパケットのみが取得できる。つまり、tcpdump を実行するマシンで実行する Cassandra ノード宛のパケットだけを取得することができる。

  この 2 つの条件により、他のマシンで起動している Cassandra ノードからこのマシンで起動している Cassandra に送られてくるパケットだけを取得できるのである。
- 同じマシン上で動作している Cassandra ノード同士の通信は取得できない。
  一方、tcpdump で測定する方法では同じマシン上で動作している Cassandra ノード同士の通信は取得できないことに留意したい。

6.3 通信量の推定

6.3.1 実験 1:gossip の通信量の推定

tcpdump を使用した上のような計測方法では、同じマシン上での CassandraNode 同士の通信を計測することはできない。そこで我々は下の仮定をもとに、計測した総通信量から Cassandra ノードで発生する総通信量を推測することにした。

- 仮定:任意の Cassandra Node 同士の通信量は平均すると同じである。

n 台の各マシンで m 台の cassandra ノードを起動したとする。つまり、システム全体で合計 $n*m$ 台の Cassandra ノードが起動されている。各マシンで tcpdump を使用して計測した得られたトラフィックの合計を Tt とし、Cassandra ノードで発生する通信量 T とく。Tt は、$(n-1)*m$ ノードの Cassandra から取得した通信量であるので、上の仮定を用いると、$n*m$ 台の cassandra ノードで発生する通信量 T は、

$$T = Tt*(n*m)/((n-1)*m) = Tt*n/(n-1)(1)$$

と Cassandra ノードで発生する gossip の総通信量を推測することができた。

6.3.2 実験 2:データ読み書きの通信量の推定

非集中のクラウドストレージである Cassandra においてのデータ読み書き時の通信手順を説明する。クライアントと直接データの受け渡しを行う Cassandra ノードを proxy ノードと呼ぶ。書き込み時には、proxy ノードから担当ノードへデータが転送され、担当ノードは proxy にデータ書き込みが成功したことを知らせる、いわゆる Ack を返す。読込時には、proxy ノードが担当ノードに読み込みリクエストを転送し、受信した担当ノードがデータを転送する。

つまり、データ読み書きがある場合に発生する通信の種類は、データの転送、データ書き込み成功の Ack, 読み込みのリクエスト、そして gossip による通信の 4 つである。実験 1 で gossip プロトコルの通信量は測定できているので、計測した通信量から実験 1 の通信量を差し引くことで推定ができる。

6.4 実 験 環 境

以下に実験環境を示す。

- Cassandra 0.6.6
- OS Linux 2.6.35.10 74.fc14.x86_64
- CPU: 2.40 GHz Xeon E5620 × 2
- JVM: Java SE 6 Update 21
- Memory: 32GB RAM
- Network: 1000BASE-T

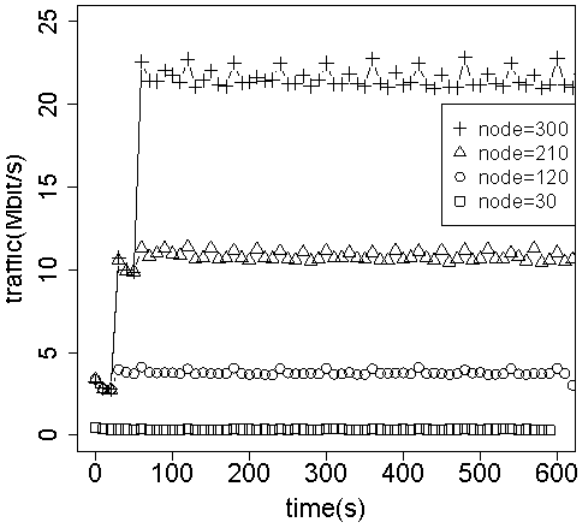
7. 実験・評価

7.1 実験 1:gossip の通信量

図 2 が、10 秒あたりのマシン間の総通信量の変化をノード数別に表したグラフであ る。(ただし、1M =10^6,1K=10^3 とする。) ノードの台数によらず、100 秒以降は通信量 が安定していることがわかる。図 3 は、ノード数と通信量が安定して いる時の (ここでは、実験開始から 200-300 秒 後とした)1 秒あたりの通信量の平均をプロッ トしたものである。

7.2 実験 2:データ読み書きの通信量の測定

図 4 は、YCSB を実行したときにデータ読み書きで発生した通信量をノード台数別に表したグラフである。ただし、QPS($queries/s$) は 1000 で一定であり、更新時のデータサイズは、1kbyte で行為してある。



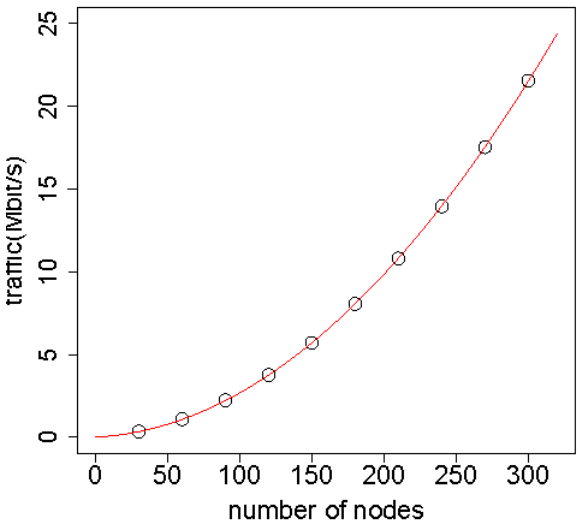


図 3 ノード台数に応じた通信量の増加

7.3 評　　価

7.3.1 総通信量の見積もり

図 1 の曲線は、プロット した点から二次関数でフィッティングしたも のである。n をノードの台数として得られた関数は、通信量 (bit/s) を T として

$$T = 224.6 * n^2 + 4314.8 * n \quad (2)$$

である。よって、通信量は O(n^2) でスケール することがわかった。 この関数から、ノード台数をパラメータとし て Cassandra の gossip プロトコル で発生しうる全体の通信量を推測することができる。例えば、$n = 1000$ のとき、[通信量]= $229Mbps$ と なる。こ

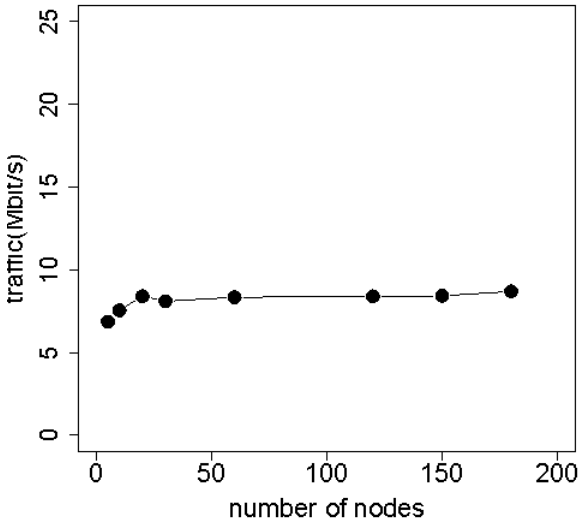


図 4 ノード台数に応じたデータ読み書きの通信量の変化

のように、この関数を使って総通信量 が見積もることができる。また、クラスタの設 計時にも活かすことができる。

7.3.2 1 ノードあたりの通信量

また同様に、1 ノードあたりの通信量を見積もることも可能である。総通信量を Cassandra ノードの台数 n で割った値が、1 ノードあたりの通信量 T' となる。つまり、

$$T' = (224.6 * n^2 + 4314.8 * n)/n \quad (3)$$
$$= 224.6 * n + 4314.8 \quad (4)$$

と O(n) でスケールすることがわかる。

7.3.3 gossip の通信量が $O(n^2)$ でスケールしていく理由

gossip の通信量が $O(n^2)$ でスケールしていく理由について考察する。総通信量 $T(bit/s)$ は、1 ノードあたりの通信量 (bit/s) を T'、ノード台数を n として、

$$T = T' * n \quad (5)$$

である。さらに 1 秒毎の gossip 通信が行われる回数を s, 1 回の gossip 通信にかかる通信量 $tg(bit)$ として、1 ノードあたりの通信量 $T'(bit/s)$ は、

$$T' = s * tg \quad (6)$$

となる。メンバー構成が安定した時を考える。Cassandra のメンバー管理シップでは、メンバー構成が安定したときは、毎秒 1-2 回の gossip 通信が行われる。よって 1 秒毎の gossip 通信する回数 s は、

$$s < 2 = Order(Const) \quad (7)$$

一方、1 回あたりの gossip 通信の通信量 $tg(bit)$ は Scuttlebutt のアルゴリズムから安定時にはノード台数に依存するので、

$$tg = Order(n) \quad (8)$$

よって式 (5) より、1 台あたりの通信量 $T'(bit/s)$、総通信量 $T(bit/s)$ は、

$$T' = O(n) \quad (9)$$
$$T = Order(n^2) \quad (10)$$

となることが証明できた。また上記に Cassandra 固有の方式はほとんどない。よって、メンバー構成が安定した時の gossip base のメンバーシップ管理アルゴリズムの通信量のスケーラビリティが評価できたことになる。

7.3.4 データ読み書きの通信量はノード数

グラフ図 4 から、QPS を一定にしたときに、構成する Cassandra ノード台数 n に応じて、増加しないことが確認できた。これは直感的に明らかである。ノード台数が n 台の時、クライアントとやり取りを行う proxy ノードがデータを保持する担当のノードである確率は、$1/n$ である。つまり、クライアントが要求するデータに対して、$(n-1)/n$ の確率で通信が発生するのである。ns が大きくなると、その確率は 1 に近づいていくため、通信量はノード台数 n に対して増加しないのである。

8. ま と め

本論では、gossip プロトコル を用いる Cassandra を対象として、 ノード台数に応じてシステム全体の通信負荷がどのように変化するのかを計測・考察した。計測の結果、システム全体で発生する通信量は、ノード 台数を n としたとき $O(n^2)$ となることが確認でき、定量的にクラウドストレージの gossip ベースのメンバーシップに要する通信量を計測することができた。今後の課題について 2 つ記す。

1 つ目は、クラウドストレージにおける gossip プロトコルを別の切り口から検討していくことである。故障検知アルゴリズム評価は、通信量という切り口のほかにも CPU 占有率や故障の伝搬スピードなどが挙げられる。それらの評価を非集中なクラウドストレージで行う。

2 つ目は、gossip プロトコル以外の故障検知アルゴリズムを評価し、比較することである。gossip 以外の故障検知アルゴリズムには、情報を更新するとすべてのメンバーに全対全で情報を伝搬していくアルゴリズムや、近傍にだけ更新を伝えるアルゴリズムなどがある。これらを評価し、比較することで gossip の性能を他のアルゴリズムと比較しながら評価できる。

謝辞 本研究は科研費（22680005）の助成を受けたものである.

参 考 文 献

1) 繧ュモ☒ラ縺☒橿茵, Vol. pp.6–13, ASCII.
2) CLON: Overlay Networks and Gossip Protocols for Cloud Environments, (2008).
3) Cassandra -A Decentralized Structured Storage System (2009).

4) Failure Detection in Large Scale Systems: a Survey (2008).
5) Efficient Reconciliation and Flow Control for Anti-Entropy (2008).
6) : Cassandra Wiki.