平成22年度 学士論文

非集中型クラウドストレージの スケーラビリティ評価

東京工業大学 理学部 情報科学科
学籍番号 07-0615-4
奥寺 昇平

指導教員
首藤 一幸 准教授

平成23年2月7日

概要

AmazonDynamo、Cassandra をはじめとした単一故障点がなく、負荷が自動的に分散される非集中型のクラウドストレージが普及しつつある。このような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、整合性を保つことが難しい。

そこで、GossipProtocol をベースとしたメンバーシップ管理を行うプロトコルが取り入れられ、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、GossipProtocol 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・考察する。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、$O(n^2)$ でスケールすることを確認した。

謝辞

本稿は以下の方々なくして、存在しえなかったでしょう。Addistant の開発、Addistant 2 の提案および本稿の編集になにかと心を砕いていただいた千葉滋講師、東京大学の光来健一氏、筑波大学の立堀道昭氏、横田大輔氏そして研究室のみなさん。心より感謝しています。

(具体的に何をしてもらったか書く)

目 次

図 目 次

表 目 次

第1章　序論

1.1　本研究の背景

近年、ネットワークを通じて計算資源を利用するクラウドコンピューティングが流行している。その中でも、ペタバイト級の大量のデータを保存するストレージタイプのクラウドに注目が集まっている。クラウドストレージの必要要件として、1.) サービスを安定的に提供すること、2.) 増加し続ける大量のデータを効率よく処理することの２つが挙げられる。1.) サービスを安定的に継続するのためには、機器の一部が故障しても、システムの外側からは、故障していないように見えなければならない。一方、クラウドのように大量のノードで構成されるシステムにおいて故障は常である。そこで、故障が起きている状況をあらかじめ想定したシステムが必要である。2.) 増加し続ける大量のデータを効率よく処理するためには、ノードの台数に題してスループットがスケールアウトできるアーキテクチャを採用することである。

そこで、特に注目を集めているのが、 Amazon Dynamo、Cassandra をはじめとした非集中型クラウドストレージである。非集中型クラウドストレージとは、構成するすべてのノードが対等の機能をもつクラウドストレージのことである。非集中型クラウドストレージの一つ目の大きな利点は、単一故障点がないことである。単一故障点とは、故障するとシステム全体が故障してしまう部位のことである。非集中型にはこのような部位はなく、安定したサービスを提供することにつながる。二点目は、負荷が自動的に分散されることである。これは、スケールアウトできるアーキテクチャであることを指している。その一方、メンバーシップ管理などを各ノードで行う必要がある。

このような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 (以後、ルーティングと呼ぶ) ためには、各ノードが他のノードを把握する必要がある。ルーティングの方針として大きく分けて、２つのバリエーションがある。担当ノードにリクエストを届けるまでに、別のノードを経由することを認めるか認めないかである。前者のルーティング方式をマルチホップ、シングルホップと呼ぶ。 特に、シングルホップ方式の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、整合性を

保つことが難しい。例えば、もし古い情報をもとにルーティングを行い、誤って別の担当ノードにリクエストを送信しまった場合、リクエストは適切に処理されないことになる。そこで、Gossip Protocol をベースとしたメンバーシップ管理を行うプロトコルが取り入れられ、全ノードがシステム全体の最新の状態を保持することが可能である。Gossip Protocol 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。

1.2　本研究の目的

非集中型のクラウドストレージにて Gossip Protocol をベースとしたメンバーシップ管理を行うプロトコルが取り入れられ、全ノードがシステム全体の最新の状態を保持することが可能である。

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、ストレージのメインタスクである read/write 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 Protocol 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・考察する。

1.3　本研究の成果

Gossip Protocol を用いる Cassandra を対象として、ノード台数に応じたシステム全体の通信量を計測した。その結果、ノード台数を n として、通信量は O(nΠe2) でスケールすることがわかった。また、結果から、クラスタ設計時に、`Gossip Protocol` ベースのメンバーシップ管理による通信量を見積もることができる。

1.4　本研究の構成

本稿の残りは、次のような構成からなっている。

第 2 章は 研究背景としディペンダブルシステムのための故障検知の重要性について説明する。

第 3 章では、関連研究として、故障検知についてのサーベイ論文と、具

体的なソリューションである。gossip protocol を利用した故障検知について言及する
第 4 章では、Cassandra について説明する。
第 5 章では、通信負荷の測定手法を説明する。
第 6 章では、クラスタ上での通信負荷の測定実験とその評価を行う。
第 7 章では、結論を述べる。

第2章　研究背景

2.1　大規模環境で dependable システムを構築するのは難しい

いろいろあるよね。そのなかでも、

2.2　大規模環境で dependable システムを構築するには、故障検知が欠かせない

あああ。

第3章　関連研究

本研究の関連研究の一つ目として、「大規模環境における故障検知についてのサーベイ」を挙げる。さらに、一つ目の関連研究で、大規模環境での有用な故障検知のソリューションとしてあげられた Gossiip protocol に関して、応用した研究を紹介する。

3.1　大規模環境における故障検知についてのサーベイ

まず、故障検知の定義について言及する。故障検知とは故障しているノードについての情報を集めるプロセスのことであり、(故障が) 疑わしきノードと、モニターしているノードのリストを管理している。リストは、スタティック、ダイナミックの二つがあり得る。ダイナミックなリストは、絶えず変化するノード群を管理していていることであり、これは、ネットワークの激しい変化に対しての対応能力を考慮すると、もっと現実的なモデルといえる。

さらに故障検知は、ハートビートと Ping という二つのタイプのキープアライブメッセージを使って、故障しているノードを判断する。

- ハートビート
  ハートビートは、モニターしているプロセスから故障検知を行うプロセスに定期的に送られてくるメッセージのことである。このメッセージで、対象ノードが故障していないことを知らせる。もし、ハートビートが制限時間内に届かなければ、故障検知を行うプロセスは、このノードは故障していると判断する。

- Ping
  Ping は、故障検知を行うプロセスからモニターしているノードに継続的に送られるメッセージのことである。故障検知を行うプロセスは、その応答として、Ack を受け取るを取ることを準備している。もしキープアライブメッセージが届かなければ、精密な調査 (タイムインターバルで区切られた一連のメッセージなど。) が行われ、実際にプロセスが故障しているかどうかが確かめられる。

実際のところメッセージ到達の遅れは予測できないので、非同期的な環境下で故障しているノードと健全なノードを見分けるのは困難である。よって、故障検知の問い合わせに対して返答がないプロセスは、「故障が疑わしき物」として扱われるのである。

次に論文では、大規模環境での故障検知を比較するための分類基準一式を提案している。図は、その要約である。

- A. 集中型 VS 分散型
  集中型の故障検知は、単体で一枚岩的なモジュールで様々なプロセスをモニターすることが可能である。集中型の長所は管理コストが低いことで、欠点は単一故障点が存在すること、潜在的なボトルネックとなりやすいことである。一方、分散型の特徴は、これらの欠点がない。分散型は、一連の故障検知モジュール群とみなされていて、各モジュールにシステムの中で異なるプロセスが割り当てられている。リクエストが来るとすぐに、それぞれ各モジュールが故障が疑わしいノードのリストを提供する。

- Pull 型 VS Push 型
  キープアライブメッセージに関して Pull 型と Push 型の二つの切り口がある。Push 型アプローチは、ハートビートを用いており、制御フローと情報フローの方向は同じである。Pull 型の故障検知は、そのフローは逆である。Pull 型の故障検知においては、Ping メッセージが使われる。ハートビートは、Ping 故障検知と比較して半分のメッセージのやりとりしか必要ないこと、タイムアウト遅延の見積もりが片道のメッセージで判断ができるので、アドバンテージがあると述べる著者を紹介している。Ping 故障検知の一つのアドバンテージは、時間の制御が故障検知をおこなうプロセスにおいてだけ実行されることである。

- C. アクティブ VS パッシブ
  アプリケーションを利用するかどうかで故障検知を分類している。アクティブプロトコルは、キープアライブメッセージを継続して送信、受信する。レイジーな、あるいはパッシブなプロトコルは、アプリケーションメッセージを利用して故障検知をおこなう。もしデータ通信が頻繁に発生するならば、故障検知は十分であると言える。一方でパッシブプロトコルが適切でない状況もあり、そうした状況ではアクティブプロトコルが必要になる。

- D. ベースライン VS シェアリング
  ノードの生存情報について共有するかどうかに関して、ベースライ

ンとシェアリングの二つに分類することができる。シェアリングアプローチでは、故障検知するモジュールはモニターしているノードの生存情報について他のモジュールと共有する。ネットワークトポロージを考慮して、概して近隣ノード群が使われる。シェアリングアルゴリズムでは、交換される情報の種類、これらが維持するキープアライブ状態の分量において異なる。
土台アプローチでは、それぞれのモジュールは、独断で故障が疑われるノードについて判断を下す。

- E. 適応可能な値 VS 一定の値
  キープアライブの頻度、タイムアウト、その他の時間は、適応可能 (Adaptive) な値にするか、一定の値にするかで分類できる。一定の値を採用する利点は、例えば、ノードがシステムに参加したときなどにそれぞれのノードにおいて一回計算されるだけなので実装しやすい点である。欠点は、ネットワークの激しい変化状況では、効率性は制限されてしまうことである。
  適応型の故障検知では例えば、送らせたタイムアウトを使うことで、新しいハートビートの到着時間を予想する。懸念は、こういった値やタイムアウトを計算することは、単純な仕事ではないことである、またいくつかのネットワーク情報を考慮しなければならない。

- F. グローバル時刻 VS ローカル時刻
  時間に関するその他の事柄は、個々でおこなうか、グローバルに行うかである。単純なアプローチは、すべてのノードに対して、グローバルにキープアライブメッセージを出す頻度を利用することである。もしすべてのノードが同質で、同じセッションの存在時間を設けているならばこのアプローチは効果的である。一方、もしノードが同質でなかったら、個々のノードがそれぞれがローカル時刻を計算することになる。

他の基準を紹介する前に、我々は、大規模環境での故障検知のオペレーションのフレーズ「正常」、「伝搬」、「再設定」について説明する。

- 正常
  「正常」フレーズにて、故障検知モジュールはキープアライブメッセージをモニターするノードに送信する。

- 伝搬
  故障が検知されたときに「伝搬」フレーズがスタートし、故障情報がその他のモジュールに伝えられる。

- 再構成
  「再構成」フレーズは「伝搬」フレーズが終了したときにスタートする。「再構成」フレーズは、ローカルの「再構成」とグローバルの「再構成」の二つフレーズから構成される。ローカルの「再構成」は、現存のモジュールがその故障を修理するときに行われる。例えば故障検知は、グループからノードを取り除くことができる。グローバルの「再構成」は、故障部位についての情報が他の故障検知モジュールに伝搬されるときに行われる。このオペレーションにより、他のノードはシステム情報の変化を反映させることができる。

大規模環境での故障検知では、二つのパフォーマス問題が存在する。モニタリングと、伝搬パターンである。迅速な伝搬は、システムの整合性 (一貫性) を維持するのを補助し、効率的なモニターリングは検知時間を短縮する。

- G. モニターパターン
  モニタリングのパターンは、「正常」フレーズにおける故障検知モジュールとモニターされているノードにおけるコミュニケーションに関連がある。モニタリングのパターンは、全対全型、ランダム型、近接ベース型の 3 つがある。

  – 全対全型
    全対全型の故障検知では、各モジュールがキープアライブメッセージをすべてのモニターしているプロセスに送信する。構成するグループのメンバーが少ない時、このアプローチは、効率的に機能する。一方で、スケーラビリティは制約される。プロセス数が膨れ上がった時には、大規模ネットワークトラフィックが発生するからである。
  – ランダム型
    ランダム型の故障検知では、各メンバごとに故障検知に使用するためのアドレスと時間のリストを管理している。グループ内の各ノードは、ランダムに他の k ノードを選択して、キープアライブメッセージを送信する。
    Gossiping protocol はハートビートをもとにランダムで故障検知を行うの故障検知の一つである。ゴシッピングはグループ数が少ないときはとても効率的であり、グループ数が多いときには複数のバリエーションがある。モニターしているノード同士でランダムであるいは定期的なコミュニケーションを通じて行われるランダム型の故障検知は、スケーラビリティを改善、

検知時間の短縮ができる。故障検知時間は、ランダムに選択されることの確率に依存する。

- 近接ベース型
  各プロセスはキープアライブメッセージを近隣ノードに送る。コミュニケーションを制限し、ローカリティ性を考慮することで、パフォーマンスが改善される。近隣ノードは、故障を検知しない限り、時間がたっても静的に変化しない。故障を検知した場合は、故障したノードをのぞいたり新しい近隣ノードを選択するために「再構成」フェーズが必要になる。またネットワーク情報は考慮されなければならない。

- H. 故障の伝搬
  あるノードが故障していると判断されたとき、この情報は他のモジュールに伝搬されなければならない。故障の伝搬は、大規模環境では非常に時間がかかるもので、伝搬にかかる時間を短縮するためにさまざまなアプローチが提案されている。全対全, ランダム,(リングあるいは) 円形状の空間、階層構造がある。

  - 全対全型の伝搬パターン
    全対全型の伝搬パターンの場合、故障は瞬時にすべての故障検知モジュールに伝わる。もし故障、チャーンが頻繁に起こるならば、ネットワーク通信量が一気に跳ね上がってしまう。この場合、故障検知の実装は、パフォーマンスの理由から IP マルチキャストを利用するのがよいだろう。
  - ランダム型のな伝搬パターン
    ランダム型のな伝搬パターンでは、モジュール、モジュール群が選択されて、現存する故障検知モジュールから故障についての情報を受け取る。利点は、コミュニケーションは全対全伝搬パターンにくらべて、少なくて十分なことである。一方欠点は、伝搬にかかる時間は、別のノードに選択される割合に依存していることである。
  - 円形状の空間の伝搬パターン
    円形状の空間の故障検知では、仮想的なリングにノード群を配列する。コミュニケーションは、近接するノード群だけで行われる。欠点としては、新しいノードが追加されたり、リングからノードが離脱したときには、近接ノードは、再構成されなければならないことである。また、仮想的なネットワークトポロジに対して、仮想的なリングをマッピングしなければならない

ことでありこれは容易なことではない。また大きなリングにおいて、故障の伝搬に非常に時間がかかる。

– 階層構造型の伝搬パターン
階層的な故障検知では、ノード群をマルチ階層に配列し、少ないグループのモニタリングに分断する。利点としては、ツリーにそって、故障が報告されるためにスケーラビリティが改善されることである。また、ネットワークインフラを考慮に入れることができるので効率的である。階層的な故障検知は、一般的に大規模分散システムにおいて用いられていおり、他の伝搬パターンで動作しているスモールグループ同士を連結している。

さらに論文では、故障検知の QoS 指標について述べている。大規模環境における故障検知の QoS は、一貫性、パフォーマンス、ダイナミックな環境に対する適応、スケーラビリティと関連している。

- 一貫性についての指標
(TODO:意味ブー) 故障検知は正確性と完全性という二つの主な分類可能な性質がある。正確性は、故障検知が犯すミスに言及していて、完全性は故障検知が最終的にノードが故障していると疑うことに言及している。故障検知の Qos 仕様のための主な指標と推論の正確性の指標を提案している。完全性に関連している主な指標は、検知するまでの時間と故障検知にかかる時間である。正解性と関連している二つの指標は、ミスの再発性、ミスが発生し続けている期間である。推論される正解性の指標は、平均ミス率、クエリーの正確性の確率、よいレスポンスタイム、良い転送タイム？

- パフォーマンスについての指標
パフォーマンスの指標は、故障検知によって使われるリソース (ノード、ネットワークなど) と関連がある。主な指標は、CPU ロード, メモリー消費、ネットワークバンド幅である。故障検知は、これらのリソースの消費を制限するかもしれないし、増加させるかもしれない。例えば、バンド幅の消費を減らすために、故障検知は、ネットワーク越しに送るメッセージ数を制限するかもしれない。パフォーマンス指標は、故障検知が影響をあたえることができない (つまり、間接的に影響を与える) 事柄もふくんでいる。これらの指標には、ノード故障率、ノード故障までの時間、平均復帰時間、平均故障間隔、平均ノード生存時間、また、遅延、メッセージロス、リンク故障率、リンク故障までの時間などのネットワークの振る舞いにも関連している。

- ダイナミックな環境に対する適応についての指標
  このカテゴリーは、故障検知がダイナミックなシステムの振る舞い (変化) に対応する能力に関連する指標を包含している。大規模環境の故障検知では、高いチャーン率、爆発的なメッセージ数の増加、メッセージロス、ヘテロな環境、柔軟性に適応できなければならない。適応性に関する指標は、ダイナミズム関連と、柔軟性に関連する指標の二つのグループに分類される。ネットワークダイナミズムに関連する指標は、故障検知が直接影響を及ぼせないようなパフォーマンス関連の指標に記述されている。フレキシビリティについての指標は、グローバルなあるいはローカルの `keep-alive` メッセージの割合、一定 `or` 変化する `keep-alive` メッセージの割合である。

- スケーラビリティについての指標
  スケーラビリティは、少なくとも以下に挙げる指標のうちの一つから定義される。サイズスケーラビリティ、地理的なスケーラビリティ。スケーラビリティについての指標には、グループサイズや遅延が含まれている。

さらにこの指標と分類に基づいて、大規模分散環境のための主な故障検知ソリューションをいくつか挙げている。

- 生存情報を共有して、検知時間を改善する
  故障検知時間は、近隣ノード間で生存情報を共有することで改善される。つまり、最初に故障を検知したノードが他のノードにアナウンスするのである。ある論文では、ネットワーク上の 2000 ノードで実験していみたところ、近隣のノードの数は増加するにもかかわらず、検知時間はほとんど変わらなかった。そこで、著者は、情報共有することで、高いチャーン率に適応するのを助けることができると主張する。

- 適応性についてのソリューション

-

-

-

-

3.2　関連研究２:Efficient Reconciliation and Flow Control for Anti-Entropy Protocols

（TODO1:この論文、このアルゴリズムの立ち位置・位置づけを認識すること。）（TODO: 具体的に、Cassandra とどう関わっているかを認識しろ。故障検知なのか？メンバーシップアルゴリズムなのか？その違いは？同時に？Cassandra の場合の gossip の動作は？）

第4章　Cassandraについて

4.1　Cassandraの概要

Cassndraは大量の構造化データを管理する分散ストレージシステムである。地理的に離れて分散された数百ノードからなる商用サーバー上で運用されることが想定されていて、可用性の高いサービスを提供し、単一故障点を持たない。このようなスケールでは大小の故障はひっきりなしに起こる。このように故障がひっきりなしに起きる状況においても、カサンドラは一貫した状態を保つのよう設計されているので、信頼性、スケーラビリティも高い。また、カサンドラはリレーショナルデータモデルを採用していない。そのかわりに動的にデータのレイアウトとフォーマットを制御できるシンプルなデータモデルを提供する。よって、読み書き効率を下げることなく高い書き込みスループットを実現している。

Cassandraは、2008年にFacebookによってオープンソースとなり、Avinash Lakshman（Amazon Dynamoの作者のうちの一人）と、Prashant Malik（Facebookエンジニア）によって設計されました。Cassandraは、Amazon DynamoとGoogle BigTableの融合によって生まれたものであり、Dynamo 2.0とも考えることができます。Cassandraは、Facebookにおけるプロダクション用途にあるが、まだ発展途上のプロダクトです。（wikiより。）

以下、本研究の関連がある点について述べる。

4.1.1　Cassandraのメンバーシップ管理について

（seedの説明も必要か。）

4.1.2　Cassandraの故障検知について

（もしかしたらリングの図とかはいるかもね。）

4.1.3　Cassandra のデータ保存部分

（データ保存も改良した。関連する部分を解説。）

4.1.4　Cassandra ノードの判別

・cassandra は,IP アドレスのみでノードを判定している。
(もしやこのセクションではない？)

4.2　Cassandra の軽量化

（TODO1:具体的に数値を細かくだそうか。）（TODO2:具体的に数値は環境依存だが、どう書く？）実験を行う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 起動するためには、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、150M 程度消費する。1 台あたり多数のノードを立ち上げるために、データ保持部分のプログラムの改変と、設定ファイルのパラメータの調整を行った。

4.2.1　プログラムの改変

（TODO1:具体的に数値を細かく?)
Cassandra では、何も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している。また、時間が経つと徐々に Onmemory 上に memtable とかが増えてしまことで、メモリー使用量域がかさむ。以下の点を踏まえて、メモリー使用領域を減らすために改変を加える。
私の実験では、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。その結果メモリー使用量域が減った。また、時間が経過しても増加する量が抑えられた。

4.2.2　設定ファイルのパラメータ調整

（TODO2:現状は、非常に怪しい [column family の定義を減らしたのが聞いてるだけってのもありうる。])
Cassandra 1 ノードで使用するスレッドは非常に多い。できるだけスレッド数を減らすために設定ファイルを変更した。具体的には、Cassandra が起動時に呼び込む storage.conf というファイルである。変更したパラ

メータは、

- $concurrent_reads$ ：　いくつまで同時読み込みを許すしきい値
- $concurrent_writes$：　いくつまで同時書き込みを許すしきい値

このパラメータは,Cassandra(の？) で使用するスレッド数に直結するので、この数を 32 - > 2 ,132 - > 2 と減らすことで、全体のスレッド数ををを 130 -> 100 に落とした。

第5章　測定手法

5.1　実験シナリオ

実験では、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10台を用意した。ワーカーマシンを lime11,lime12,...lime20 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 を利用して制御した。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。

5.1.1　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
- Java 仮想マシン: Java SE 6 Update 21
- メモリー: 32GB RAM
- ネットワーク: 1000BASE-T

5.1.2　計測方法について

計測にあたっていくつか工夫した点を紹介する。

- 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。(img:左図がデフォルト,img 右図設定ファイル変更後)

  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。特に上の環境では、スレッド数が 130 程度であった。(Cassandra の設定はデフォルト。)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、(左図参照)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。(参照:右図)

- 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 とか、PING などのリクエストである。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。この作業で、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ればよく、ノイズが減らすことができる。
  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 (lime11,lim12,...,lime20) 上の番号 n を使用し、

lime[n] 上で起動するマシンは、仮想的に 10.20.n.0/24 なるサブネットを設けた。例えば、n=19 のとき、10.20.19.1 〜10.20.19.254 までの仮想アドレスを作成し Cassandra ノードに割り当てた。以下がプライベートネットワークを構築後に ifconfig コマンドを実行したときの実行結果と、このネットワーク内で Cassandra クラスタが構成されている様子である。

図 5.1:　ifconfig の実行結果

- 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を取得することになるのである。

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。以下、tcpdump を実行したときの実行結果の例である。

- 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。

5.2　通信量の測定について

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。我々が立てた仮定である。(TODO:もっとうまい言い方で。)

- 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 T とし、Cassandra ノードで発生する通信量 TT とく。このとき、

$$T = ((n-1) * m \text{ 台の } cassandra \text{ ノードで発生する通信量}) \tag{5.1}$$

である。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 TT は、

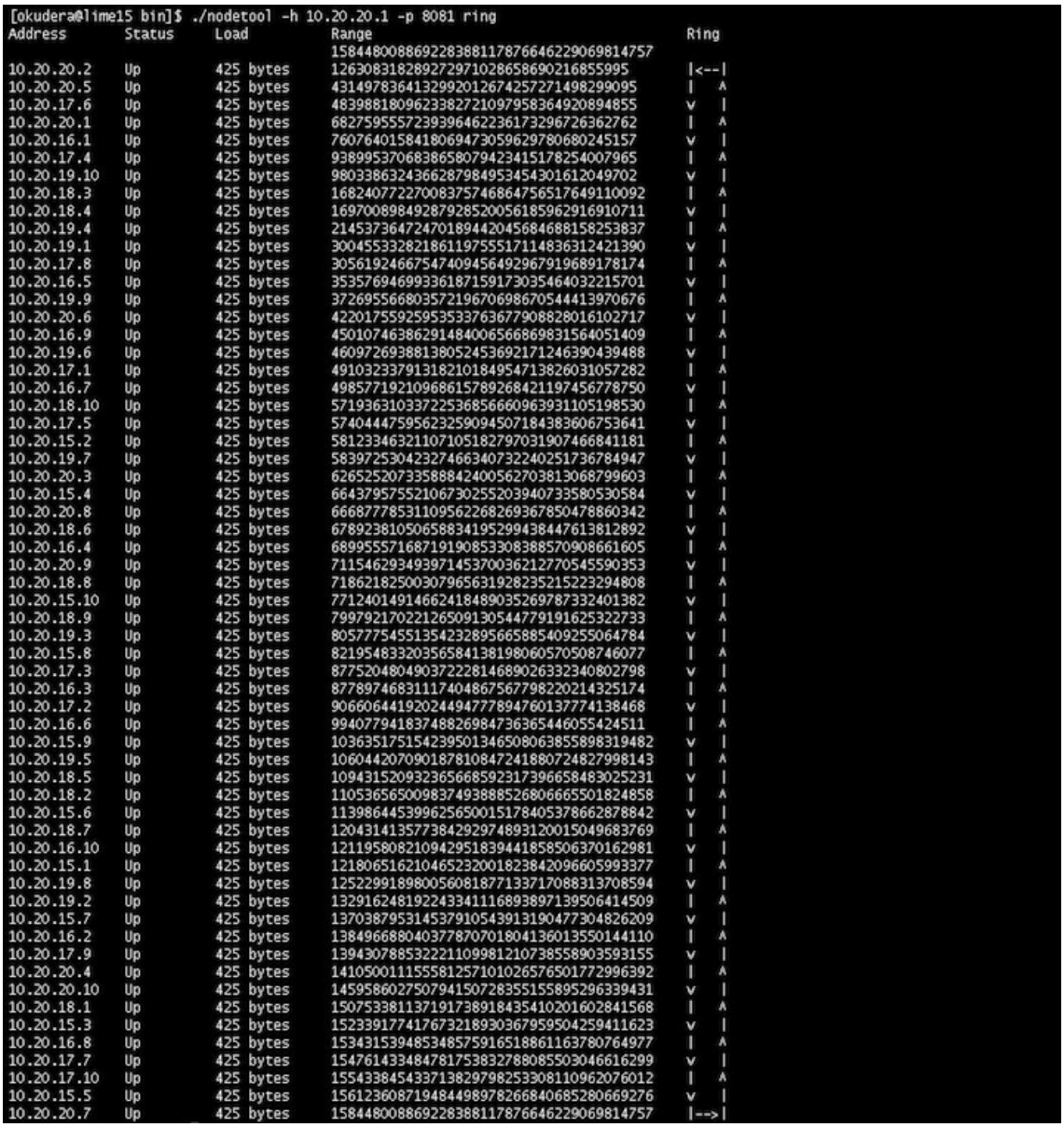
$$TT = T * (n * m)/((n-1) * m) = T * n/(n-1) \tag{5.2}$$

と Cassandra ノードで発生する総通信量を推測することができた。

具体的に、n = 4 台,m=2 の時について、図を使いながら解説する。(図)

5.3　自動で通信量の測定を行い、その後解析を行うプログラムを作成した

図のような自動で通信量の測定を行い、その後解析を行うプログラムを作成した。



```
[okudera@lime15 bin]$ ./nodetool -h 10.20.20.1 -p 8081 ring
Address         Status     Load          Range                                      Ring
                                         158448008869228388117876646229069814757
10.20.20.2      Up         425 bytes     126308318289272971028658690216855995       |<--|
10.20.20.5      Up         425 bytes     431497836413299201267425727149829909 5     |   ^
10.20.17.6      Up         425 bytes     483988180962338272109795836492089485 5     v   |
10.20.20.1      Up         425 bytes     682759555723939646223617329672636276 2     |   ^
10.20.16.1      Up         425 bytes     760764015841806947305962978068024515 7     v   |
10.20.17.4      Up         425 bytes     938995370683865807942341517825400796 5     |   ^
10.20.19.10     Up         425 bytes     980338632436628798495345430161204970 2     v   |
10.20.18.3      Up         425 bytes     168240772270083757468647565176491100 92    |   ^
10.20.18.4      Up         425 bytes     169700898492879285200561859629169107 11    v   |
10.20.19.4      Up         425 bytes     214537364724701894420456846881582538 37    |   ^
10.20.19.1      Up         425 bytes     300455332821861197555171148363124213 90    v   |
10.20.17.8      Up         425 bytes     305619246675474094564929679196891781 74    |   ^
10.20.16.5      Up         425 bytes     353576946993361871591730354640322157 01    v   |
10.20.19.9      Up         425 bytes     372695566803572196706986705444139706 76    |   ^
10.20.20.6      Up         425 bytes     422017559259535337636779088280161027 17    v   |
10.20.16.9      Up         425 bytes     450107463862914840065668698315640514 09    |   ^
10.20.19.6      Up         425 bytes     460972693881380524536921712463904394 88    v   |
10.20.17.1      Up         425 bytes     491032337913182101849547138260310572 82    |   ^
10.20.16.7      Up         425 bytes     498577192109686157892684211974567787 50    v   |
10.20.18.10     Up         425 bytes     571936310337225368566609639311051985 30    |   ^
10.20.17.5      Up         425 bytes     574044475956232590945071843836067536 41    v   |
10.20.15.2      Up         425 bytes     581233463211071051827970319074668411 81    |   ^
10.20.19.7      Up         425 bytes     583972530423274663407322402517367849 47    v   |
10.20.20.3      Up         425 bytes     626525207335888424005627038130687996 03    |   ^
10.20.15.4      Up         425 bytes     664379575521067302552039407335805305 84    v   |
10.20.20.8      Up         425 bytes     666877785311095622682693678504788603 42    |   ^
10.20.18.6      Up         425 bytes     678923810506588341952994384476138289 2     v   |
10.20.16.4      Up         425 bytes     689955571687191908533083885709086616 05    |   ^
10.20.20.9      Up         425 bytes     711546293493971453700362127705455903 53    v   |
10.20.18.8      Up         425 bytes     718621825003079656319282352152232948 08    |   ^
10.20.15.10     Up         425 bytes     771240149146624184890352697873324013 82    v   |
10.20.18.9      Up         425 bytes     799792170221265091305447791916253227 33    |   ^
10.20.19.3      Up         425 bytes     805777545513542328956658854092550647 84    v   |
10.20.15.8      Up         425 bytes     821954833203565841381980605705087460 77    |   ^
10.20.17.3      Up         425 bytes     877520480490372228146890263323408027 98    v   |
10.20.16.3      Up         425 bytes     877897468311174048675677982202143251 74    |   ^
10.20.17.2      Up         425 bytes     906606441920244947778947601377741384 68    v   |
10.20.16.6      Up         425 bytes     994077941837488269847363654460554245 11    |   ^
10.20.15.9      Up         425 bytes     103635175154239501346508063855898319 482   v   |
10.20.19.5      Up         425 bytes     106044207090187810847241880724827998 143   |   ^
10.20.18.5      Up         425 bytes     109431520932365668592317396658483025 231   v   |
10.20.18.2      Up         425 bytes     110536565009837493888526806665501824 858   |   ^
10.20.15.6      Up         425 bytes     113986445399625650015178405378662878 842   v   |
10.20.18.7      Up         425 bytes     120431413577384292974893120015049683 769   |   ^
10.20.16.10     Up         425 bytes     121195808210942951839441858506370162 981   v   |
10.20.15.1      Up         425 bytes     121806516210465232001823842096605993 377   |   ^
10.20.19.8      Up         425 bytes     125229918980056081877133717088313708 594   v   |
10.20.19.2      Up         425 bytes     132916248192243341116893897139506414 509   |   ^
10.20.15.7      Up         425 bytes     137038795314537910543913190477304826 209   v   |
10.20.16.2      Up         425 bytes     138496688040377870701804136013550144 110   |   ^
10.20.17.9      Up         425 bytes     139430788532221109981210738558903593 155   v   |
10.20.20.4      Up         425 bytes     141050011155581257101026576501772996 392   |   ^
10.20.20.10     Up         425 bytes     145958602750794150728355155895296339 431   v   |
10.20.18.1      Up         425 bytes     150753381137191738918435410201602841 568   |   ^
10.20.15.3      Up         425 bytes     152339177417673218930367959504259411 623   v   |
10.20.16.8      Up         425 bytes     153431539485348575916518861163780764 977   |   ^
10.20.17.7      Up         425 bytes     154761433484781753832788085503046616 299   v   |
10.20.17.10     Up         425 bytes     155433845433713829798253308110962076 012   |   ^
10.20.15.5      Up         425 bytes     156123608719484498978266840685280669 276   v   |
10.20.20.7      Up         425 bytes     158448008869228388117876646229069814757   |-->|
```

図 5.2:　Cassandra クラスタを構成したときの様子

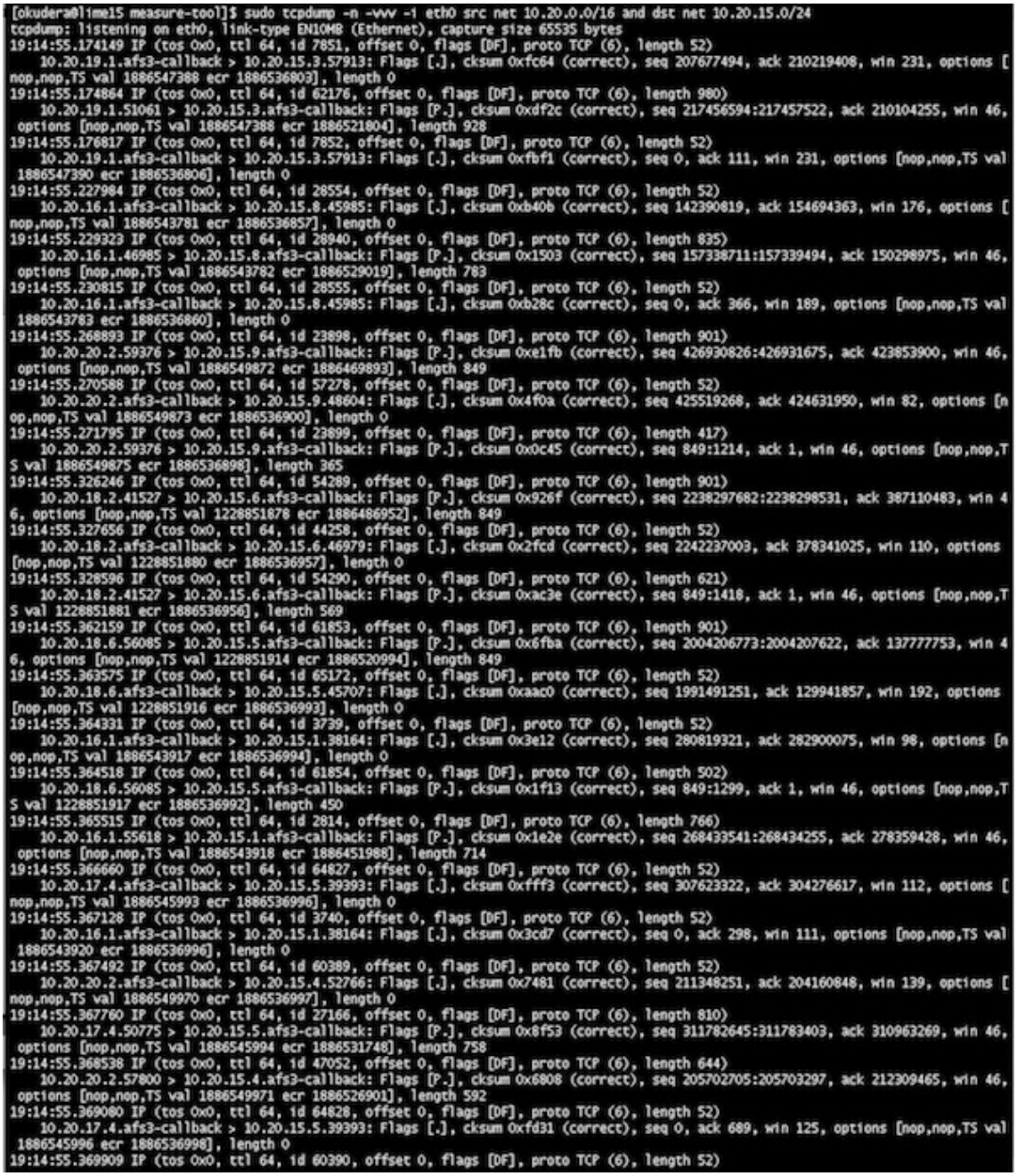


図 5.3: tcpdump の実行結果

```
#! /bin/sh


#============== 0.実験の初期化 ========================
#設定ファイル(実験パラメータ)の読み込み。
#設定ファイルには、起動するCassandraのノード数や、使用する実マシン、データ保存のためのホームディレクトリ、
#Cassadra実行ファイルのホームディレクトリが記述されている。
if [ "x$E_PARA" = "x" ]; then
    # Locations (in order) to use when searching for an include file.
    for include in `dirname $0`/e-para.conf; do
        if [ -r $include ]; then
            . $include
            break
        fi
    done
#otherwise, source the specified include.
elif [ -r $E_PARA_INCLUDE ]; then
    . $E_PARA_INCLUDE
fi

#現在時刻を取得、これがそれぞれの実験を識別するものになる。
date=`date +%Y%m%d%H%M%S `

#データ保存フォルダの作成
final_result_data_directory="${cassandra_final_result_home}/node${node_number}"
if [ ! -e $final_result_data_directory ];then
        mkdir $final_result_data_directory
fi
mkdir $final_result_data_directory/$date

#メタ情報(実験パラメータ)の保存
echo "nodes=${node_number},\
span=${measuring_span}\
worker_ids=${worker_node_ids[@]}
" >$final_result_data_directory/$date/meta-info

cat /home/okudera/work/cassandra-lite/measure-tool/dynamic_seeds_conf >> $final_result_data_directory/$date/meta-info
#leave worker id in meta-info
echo  ${worker_node_ids[@]} > $final_result_data_directory/$date/worker_node_ids


#============ 1.GXPCジョブスケジューラを使用して各ノードでジョブを実行する。以下はその初期設定 ==================
admin_node_name=`hostname`
gxpc use ssh $admin_node_name $cluster_name
for i in ${worker_node_ids[@]}
do
        gxpc explore $cluster_name[[$i]]
done


#============ 2.計測を開始する  ====================================================
#各ノードでmeasureTCPPacketスクリプトを実行する
#measureTCPPacketスクリプトでは、如何が行われる。
#1.計測の開始
#2.Cassandraの起動,
#3.計測時間経過後にCassandraの停止
#4.計測の終了
gxpc e ${measure_tool_directory}/measureTCPPacket  $date $measuring_span $each_node_number $final_result_data_directory/$date


#すべてのノードで計測が終了した後次のステップに進む
#========== 3.計測した通信量の情報から必要な情報だけを取得する。解析スクリプトを各ノードで実行 ==================
gxpc e ${measure_tool_directory}/analizeTraffic $date $final_result_data_directory

#各ノードで解析したデータを集約する。
R --vanilla --slave --args $final_result_data_directory/$date < ${measure_tool_directory}/R/reduceData.R

#最後に、集約して得られたデータからCassandraノード同士で発生した通信量を推定する。
R --vanilla --slave --args $final_result_data_directory/$date < ${measure_tool_directory}/R/calculateWholeThroughput.R

gxpc quit
```

図 5.4: 実行スクリプト

第6章　実験・評価

6.1　予備実験

本実験に入る前に、予備実験を行った。

6.1.1　予備実験 (1) マシン数が異なる時の通信量の推定値は変わらない。

この実験はによる、上述の Cassandra ノードで発生する総通信量の推測が妥当であることを強調する。以下の TypeA,TypeB の 2 パターンで 120 台の Cassandra ノードを立ち上げて 10 分間計測を行い、推定される通信量が一致することを確認する。

- TypeA
  一台あたり Cassandra ノード 12 個を立ち上げたマシン 10 台でクラスタを構成する。

- typeB
  一台あたり Cassandra ノード 60 個を立ち上げたマシン 2 台でクラスタを構成する。

Type A の場合は、時間 t の時の通信量を A(t) とおくと、

$$[\text{推定される通信量}](t) = A(t) * 10/9 \tag{6.1}$$

Type B の場合は、時間 t の時の通信量を B(t) とおくと、

$$\text{推定される通信量 } (t) = B(t) * 2/1 \tag{6.2}$$

となる。この推定量をグラフに重ねて書いてみる。

先ほどと同様に、X 軸は時間軸を示していて、Y 軸は通信量 (単位は M bit) である。平均は、??でありほぞ一致することが確認できましたと。

TypeA:120 cassandra nodes on 10 machines

TypeB:120 cassandra nodes on 2 machines

6.1.2　予備実験 (2)

また、Cassandra 特有の seed の数が変化されたときに、通信量がどのように変わるのかも調べた。実験では、以下のように、seed 数を 1,2,4,10,60,120 と変化させながら、２台のマシン上で cassandra120 台を起動し通信量の変化を観察した（グラフ）を一枚書いて終了！

6.2　本実験

図??は、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?2 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。図中の曲線は、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、

$$[\text{通信量 } (bit)] = 224.6 \times n^2 + 4314.8 \times n \tag{6.3}$$

である。

6.3　評価

6.3.1　総通信量の見積もり

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 Protocol 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。これは後述する。

6.3.2　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 node 台数 n で割った値が、1 ノードあたりの通信量となる。つまり、

$$[\,1 \text{ ノードあたりの通信量}] = (224.6 \times n^2 + 4314.8 \times n)/n = 224.6 \times n + 4314.8 \tag{6.4}$$

と O(n) でスケールすることがわかる。グラフに示すと以下になる。X 軸がノード台数、Y 軸が通信量である。また、この通信量はメンバーシップ管理で受信、送信する通信量のそれぞれの値である。

6.3.3　システム全体の限界とは？

上より総通信量はある関数に沿って、スケールしていることがわかった。この結果は、クラスタ設計時 に活かすことができる。つまり、クラスタを構成するノード数に応じて、どの機器でどれくらいの通信量が発生するかを見積もることができる。一般的には議論するのは難しいので、クラスタの構成ごとに具体的なケースに商店を当てて見ていく。

TYPEA:データセンター３つでクラスタを構成する場合

TYPE B:データセンター１つでクラスタを構成する場合

TYPE C:ネットワーク 1000base-T のような切り口

6.3.4　通信量が $O(n^2)$ でスケールしていく理由

最後に、メンバーシップ管理の通信量が $O(n^2)$ でスケールしていく理由について考察する。Cassandra が採用する Gossip Protocol に依存するところもあるため、まず Cassandra のメンバーシップ管理の実装し、その後数式から考察していく。

6.3.5　Cassandra のメンバーシップ管理の実装

・gossip 通信の定義 (往復何回あるかとかいう必要があるか？) ・毎秒どのようにどのノードと gossip 通信を行うかのロジックについて。

6.3.6　数式から説明

ノード台数を n として、安定時に発生する通信量が $O(n^2)$ であることをしめす。

$$(\text{ノード台数}) = n \tag{6.5}$$

$$(\text{総通信量}/s) = (\text{1 台あたりの通信量}/s) * (\text{ノード台数}) \tag{6.6}$$

さらに、(一台あたりの通信量/s) を分解すると、

$$(1\text{台あたりの通信量}) = (1\text{秒あたりの}\mathit{gossip}\text{通信する回数})*(\text{一回あたりの}\mathit{gossip}\text{通信にかかる通信量} \tag{6.7}$$

ここで、Cassandra のメンバーシップ管理では、メンバー構成が安定時 (TODO:どーやってもりこもうか。) には、

$$(1\text{秒あたりの}\mathit{gossip}\text{通信する回数}) = 1, \tag{6.8}$$

一方、

$$(\text{一回あたりの}\mathit{gossip}\text{通信にかかる通信量}) = Order(n) \tag{6.9}$$

よって、

$$(1\text{台あたりの通信量}) = O(n) \tag{6.10}$$

$$(\text{総通信量}) = Order(n^2) \tag{6.11}$$

となることが証明できた。また、安定時を考えると、1 秒あたりの gossip 通信する回数は、1 回であるので Cassandra 独自という意味合いはうすれ、より一般的な Gossip Protocol base のメンバーシップ管理アルゴリズムの結果といえる。(TODO:言い方うまく！)

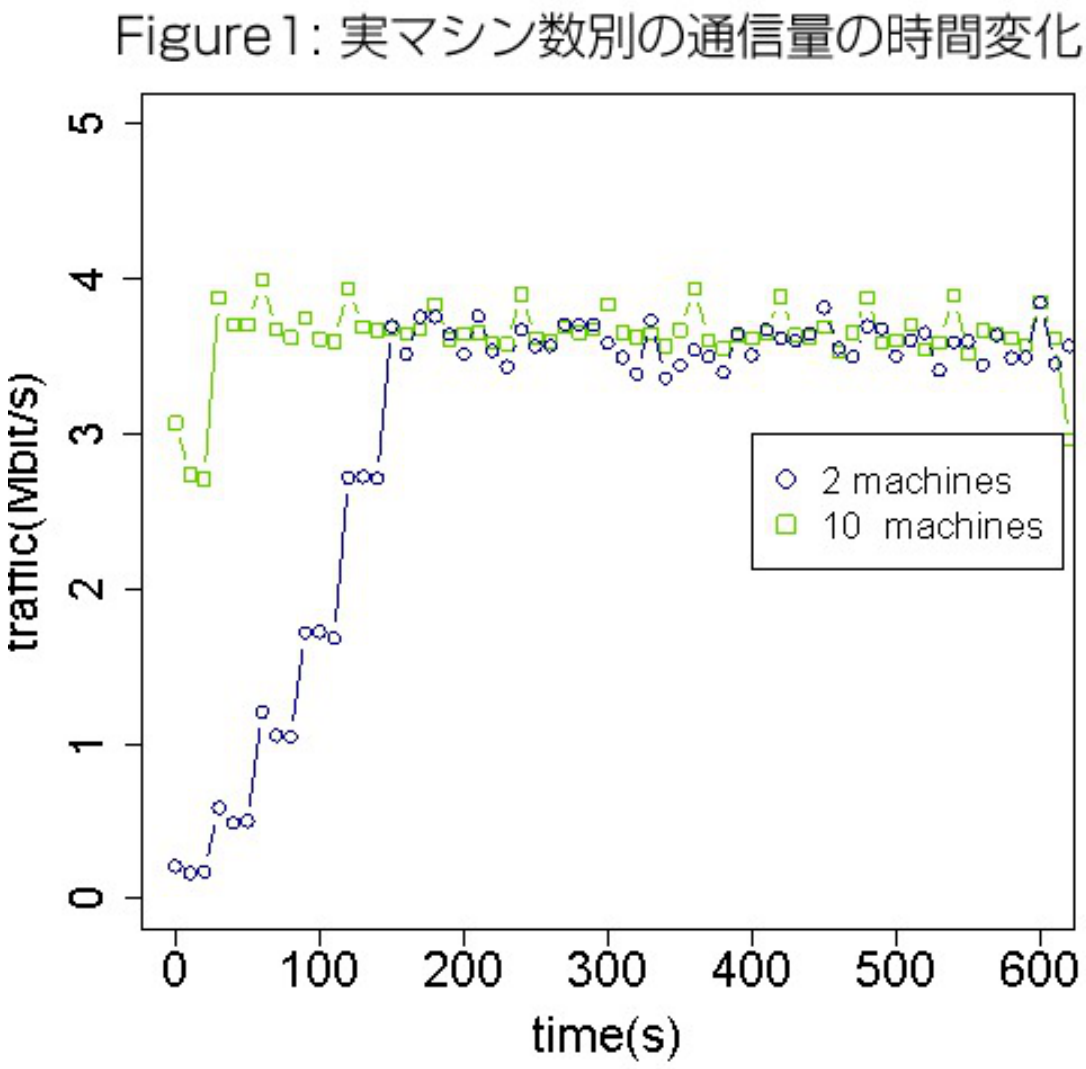


図 6.1: infer-traffic の実行結果

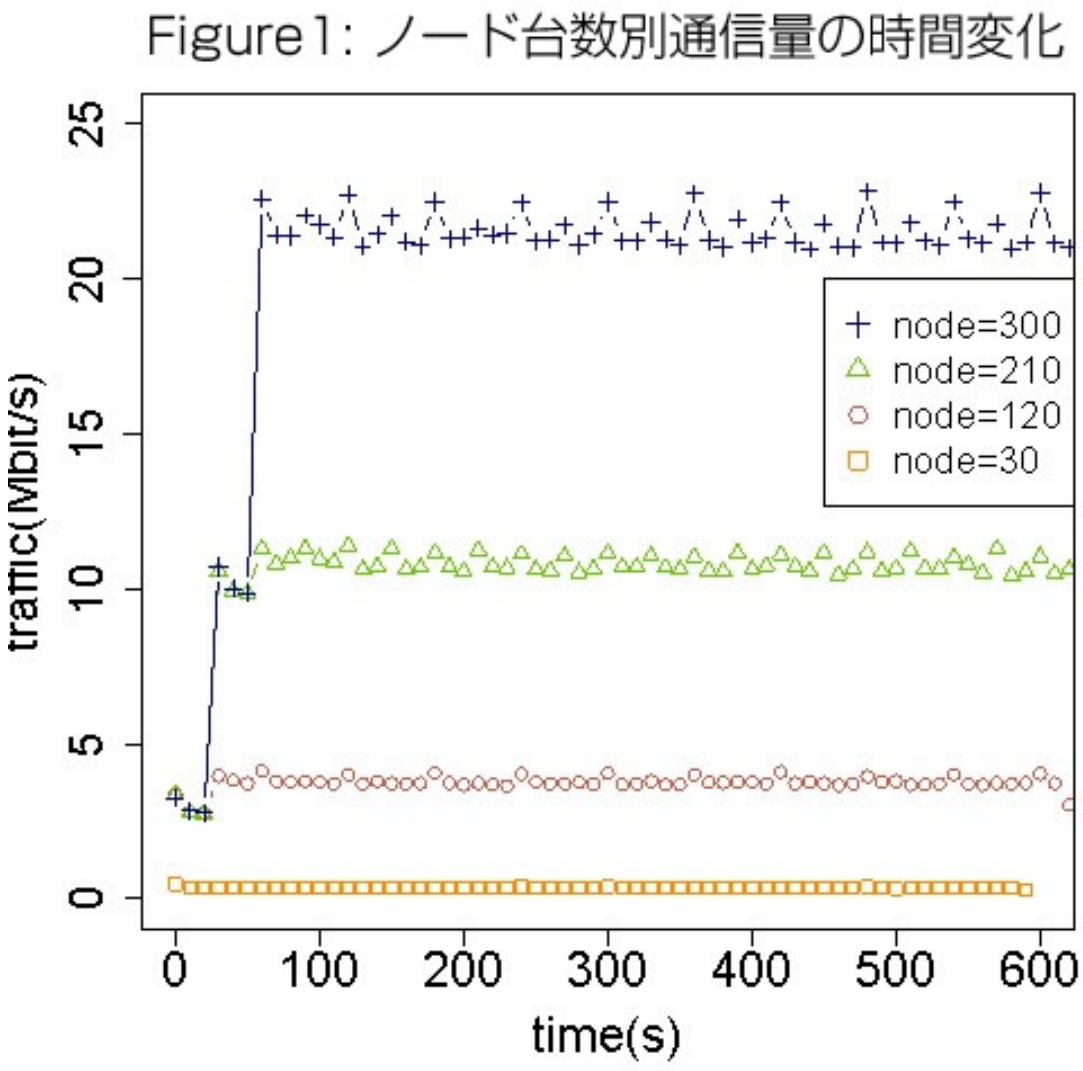


図 6.2: infer-traffic の実行結果

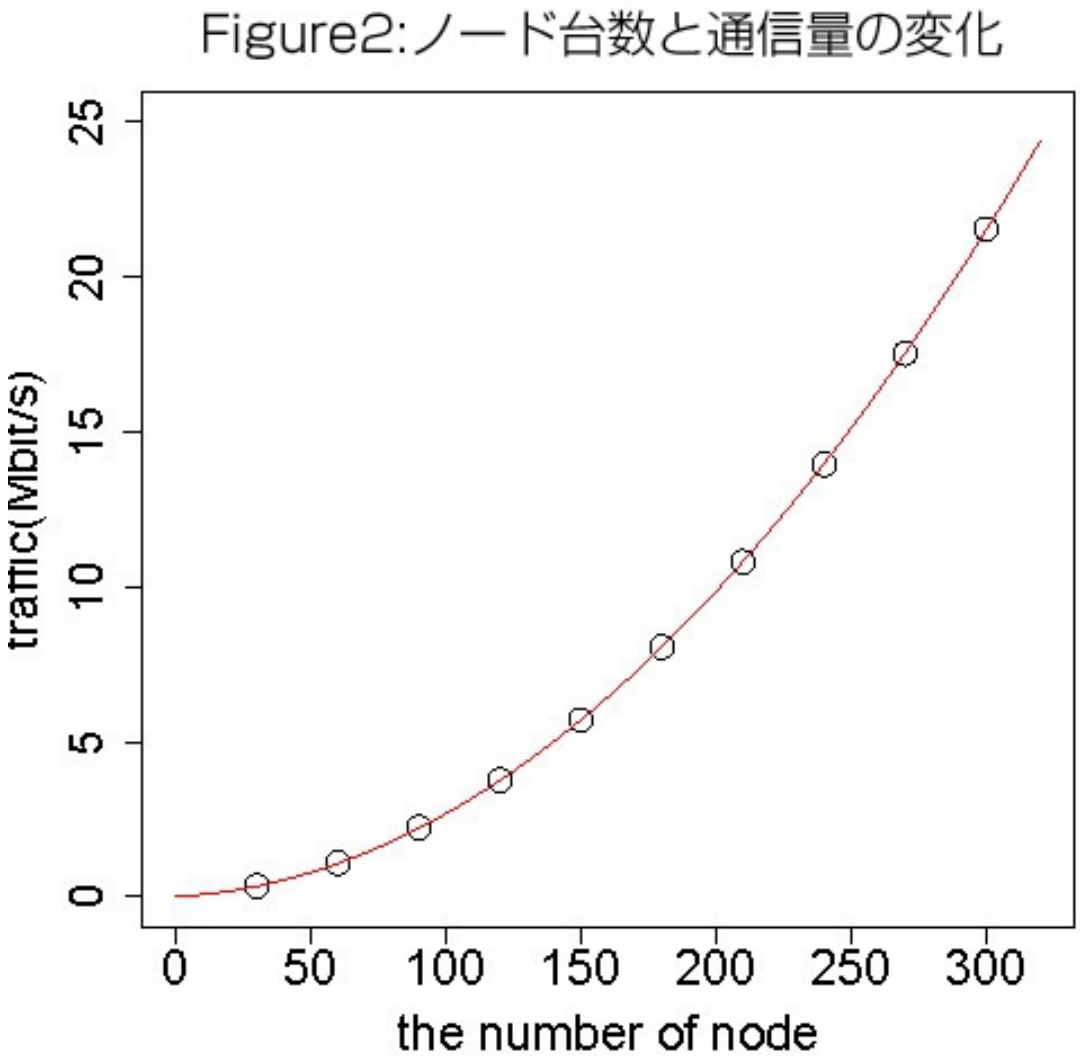


図 6.3: infer-traffic の実行結果

第7章　結論

7.1　まとめ

1．通信量の測定方法を紹介した。

2．gossip-base のプロトコルで発生する通信量は O(n^2) でスケールする。

3．クラスタ設計時の通信量を見積もることができる。

7.2　今後の課題

7.2.1　.gossip protocol を別の切り口から評価する

ルーターのフラップやチャーン状態の時の通信量を計測。通信量と適切な経路情報が伝搬しているのかという二つの切り口で gossip を評価する。

7.2.2　Gossip Protocol 以外のメンバーシップ管理の評価

Gossip Protocol 以外のメンバーシップ管理を行うプロトコルの通信量を調べること

参考文献

[1] : .

[2] : .

[3] : Load-time Structural Reflection in Java, pp. 313--336 (2000).

付 録A　プログラム例