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

本論文を作成にあたり指導を賜りました、指導教官の首藤先生に深謝致します。また、実験、文章構成について活発な議論にお付き合い頂いた中村さん、長尾さんに感謝の意を表します。同期の宮尾さん、建部さん、互いに支え合いながら卒業論文を完成させることができました。ありがとうございます。

目 次

第 1 章 序論 **6**
1.1 本研究の背景 6
1.2 本研究の目的 7
1.3 本研究の成果 7
1.4 本論文の構成 7

第 2 章 研究背景 **9**
2.1 分散システムにおけるスケーラビリティの確保のためのスケールアウト戦略 9
2.2 故障検知の代表的プロトコル gossip プロトコルとその問題点 10

第 3 章 関連研究 **11**
3.1 CLON[7] 11
3.2 JetStream[9] 11
3.3 CREW: A Gossip-based Flash-Dissemination System[5] . 12

第 4 章 Gossip protocol **13**
4.1 故障検知とは 13
4.2 Scuttlebutt 18

第 5 章 Cassandra について **21**
5.1 Cassandra の概要 21
5.1.1 メンバーシップ管理 21

第 6 章 Cassandra の軽量化と測定手法 **22**
6.1 Cassandra の軽量化 22
6.1.1 プログラムの改変 22
6.1.2 設定ファイルのパラメータ調整 22
6.2 測定手法 23
6.2.1 実験 1:gossip の通信量の測定 23
6.2.2 実験 2:データ読み書きの通信量の測定 24
6.2.3 計測方法について 24
6.2.4 通信量の推定 29

6.3 通信量測定のためのプログラムの作成 34
6.3.1 実験環境 34

第 7 章 実験・評価 **37**
7.1 予備実験 37
7.1.1 予備実験 (1) gossip 通信量推定の妥当性 37
7.1.2 予備実験 (3):データ通信量推定の妥当性 38
7.1.3 予備実験 (4):データ量に応じた Cassandra 内部での総通信量の変化 39
7.2 本実験 40
7.2.1 実験 1:gossip の通信量 40
7.2.2 実験 2:データ読み書きの通信量の測定 40
7.3 評価 44
7.3.1 gossip 通信量の見積もり 44
7.3.2 1 ノードあたりの通信量 44
7.3.3 複数のデータセンターをまたぐクラスタにおける gossip プロトコルの問題点 44

第 8 章 結論 **48**
8.1 まとめ 48
8.2 今後の課題 48

図 目 次

4.1 gossip プロトコル 15

6.1 リソース制限変更前 24
6.2 リソース制限変更後 25
6.3 ifconfig の実行結果 26
6.4 Cassandra クラスタを構成したときの様子 27
6.5 tcpdump の実行結果 28
6.6 gossip 通信量の推定 30
6.7 データ通信量の推定 -書き込み時 1- 32
6.8 データ通信量の推定 -書き込み時 2- 32
6.9 データ通信量の推定 -読み込み時 1- 35
6.10 データ通信量の推定 -読み込み時 2- 35
6.11 実行スクリプト 36

7.1 マシン数に応じたトラフィックの時間変化 38
7.2 マシン台数に応じたデータ読み書きの通信量推定値の変化 39
7.3 データ量に応じた Cassandra 内部での総通信量の変化 . . . 40
7.4 infer-traffic の実行結果 41
7.5 infer-traffic の実行結果 42
7.6 ノード台数に応じたデータ読み書きの通信量の変化 43
7.7 データセンター間をまたぐクラスタの構成 45

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が隠蔽され、正常に動作しているように振舞わなければならない。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が必要である。

そこで、特に注目を集めているのが、Amazon Dynamo[4]、Cassandra[6]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。単一故障点がないとは、ある部位が故障するとシステム全体が故障してしまうような場所が存在しないことである。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が行う必要がある。

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つのパターンがある。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、後者を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。つまり、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、ストレージのフロントエントの処理である 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 $O(n^2)$ 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　本論文の構成

本稿の残りは、次のような構成からなっている。

第 2 章は 研究背景として分散システムのアベイラビリティを確保するための故障検知の重要性について説明する。

第 3 章では、関連研究としてクラウド環境で gossip プロトコルベースで

のマルチキャストの評価を行った CLON について紹介する。
第 4 章では、Gossip プロトコルとその応用 Scuttlebutt プロトコルについて説明する。
第 5 章では、実験で使用する Cassandra について説明する。
第 6 章では、Cassandra の軽量化と通信測定の手法を説明する。
第 7 章では、実験とその評価を行う。
第 7 章では、結論としてまとめと今後の課題を述べる。

第2章　研究背景

2.1　分散システムにおけるスケーラビリティの確保のためのスケールアウト戦略

分散システムには高い可用性が必要とされる。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、常に稼動し続けるような安定性・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 [?] 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) 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を図ってきたが、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、費用を押さえて高い性能を引き出すことができる。しかし、スケールアウトの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、システムでマシンやスイッチなどのハードウェアの故障が発生する確率は増加する。つまりシステム全体での平均故障間隔 (MTBF) が非常に短くなってしまう。そこで、ハードウェアは常に故障するということを前提とした耐故障性や故障検知といったシステムの仕組みづくりが必要となる。

2.2　故障検知の代表的プロトコル gossip プロトコルとその問題点

故障検知とは、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。また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、特にクラウドストレージにおいては、故障検知によるネットワーク、CPU などのリソース消費は、控えばければならない。なぜならばクラウドストレージでは、データの読み書きがメインの処理であるからである。

故障検知を行う代表的な方法に Gossip Protocol がある。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。Gossip Protocol 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。しかしながら、gossip を利用した故障検知はメインタスクではない。特にクラウドストレージにおいてのメインタスクは、クライアントからのデータの読み書きリクエストを処理することであり、故障検知はバックグラウンド、サブタスクとして扱われなければならない。すなわち、CPU サイクル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故障検知が実現されることが求められているのである。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。

第3章　関連研究

3.1　CLON[7]

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、評価した論文を紹介する。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、信頼性のあるデータ伝搬に対してスケールするアプローチである一方、このようなクラウド環境においてはその冗長性から、リンクとノードに高いリソース消費を引き起こしている。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。本研究との違いは、我々は非集中なクラウドストレージに焦点を当てていること、gossip プロトコルの応用用途が故障検知である点である。

3.2　JetStream[9]

この論文では、オリジナルの gossip-protocol はランダムにノードを選択して通信するためさまざまなノードでメッセージの受信でオーバーヘッドを起こすことにつながると指摘している。そこで、著者は単純なソーシャルネットワークの規律を埋め込むことで、各ノードが gossip する相手を賢く選択できるようにした。その結果、メッセージ受信のオーバーヘッドをアーカイブできると同時に、gossip 伝搬の遅延の削減、システム全体のネットワーク通信量を減少を実現した。

CLON と同様に、システム全体のトラフィックを評価しているところが本研究との関連である。本研究との違いは、我々は非集中なクラウドストレージ、故障検知に焦点を当てていることである。

3.3 CREW: A Gossip-based Flash-Dissemination System[5]

この論文では、伝搬遅延が少なく、効率が良い gossip プロトコルベースのブロードキャストを提案している。

本研究との関連は、ノード台数に応じた gossip 通信量のスケーラビリティについて評価しているところである。本研究との主な違いは、この論文が一般の P2P アーキテクチャに焦点を当てて議論をしている一方、我々は非集中なクラウドストレージに焦点を当てていることである。

第4章　Gossip protocol

本研究の主対象であるGossipプロトコルについて詳しく説明する。まず、故障検知におけるgossipプロトコルの位置づけを整理する。次に、具体的にgossipプロトコルをベースにした情報共有プロトコルであるScuttlebuttについて説明する。

4.1　故障検知とは

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、「故障が疑わしきもの」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。さらに Pull&Push 型というのもある。これは pull 型の gossip と似ている。違いは、ノード q がノード p に更新情報を送るときに、ノード q のダイジェストも同時に送るところである。その後ノード p はダイジェストを見て、ノード q の情報も更新させる。

- 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できる。アクティブなプロトコルは、キープアライブメッセージを継続して送信、受信する。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

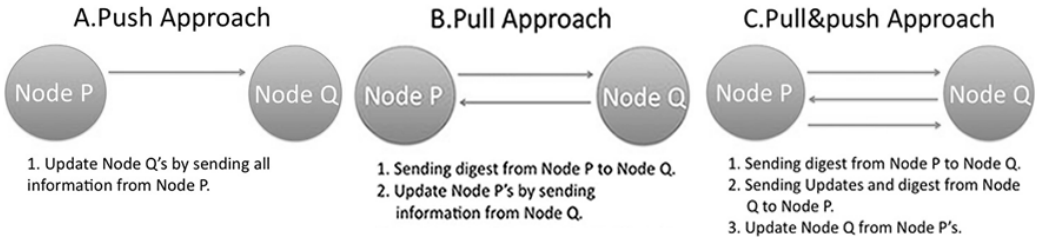

図 4.1: gossip プロトコル

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な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 3 つの大きなオペレーション「正常」、「伝搬」、「再設定」を説明する。

- 正常
  「正常」フェーズにて、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」フェーズがスタートし、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」フェーズは「伝搬」フェーズが終了したときにスタートする。「再構成」フェーズは、ローカルの「再構成」とグローバルの「再構成」の二つフェ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」フェ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。

次に、Pull&Push 型の gossip プロトコルを応用した情報共有プロトコル、Scuttlebutt[10] について詳しく説明する。Scuttlebutt ではメンバー同士でやりとりする情報に制約をかけることで、システム全体の整合性を保ちつつ、CPU やメモリーなどのリソースの消費を押さえることができる。

4.2　Scuttlebutt

Scuttlebutt とは、分散システムでイベンチュアルコンシステント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。イベンチュアルコンシステントなシステムとは、システムに一時的にコンシステントでない状態が生まれても、ある期間の後にはコンシステントな状態になるシステムのことである。例えば、DNS システムなどである。[?] 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 のサイズ内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 $\Delta_{scuttle}^{p \to q}$ をメンバー q に送信する。

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

$$\Delta_{scuttle}^{p \to q} = \{(r, k, v, n)|u_p(r)(k) = (v, n)$$

$$\wedge n > max(u_q(r)))\}$$

すなわち、$\Delta_{scuttle}^{p \to q} = \{(r, k, v, n) | u_p(r)(k) = (v, n)$ 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。

第5章　Cassandraについて

5.1　Cassandraの概要

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。[1] 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短く、故障検知が重要である。

5.1.1　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、追加するノードに予めシステム内にある最初にコンタクトを取るノード (これを seed と呼ぶ) 情報を設定しておき、そのノード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 を使用したノード同士の情報交換を 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: 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 gossip 通信を行う。
2. STEP2: 生存ノードと到達できないノード数に応じたある確率で、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 gossip 通信を行ったノードが seed でないとき、あるいは seed ノード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、到達できないノード数に応じたある確率で、ランダムで seed に gossip 通信を行う。

スタンドアロン形式で Cassandra ノードを立ち上げる場合を除き、この方式により毎秒 1-3 回の gossip 通信を行っている。

第6章　Cassandraの軽量化と測定手法

6.1　Cassandraの軽量化

通信量のスケーラビリティを評価するためには、Cassandra ノードを多数台起動する必要がある。しかしながら、物理リソースの都合上 10 台のマシンのみが利用可能であった。そこで、マシン 1 台あたりに複数の Cassandra ノード を多数起動することが必要になった。

そもそもデフォルトでは、Cassandra は 1 マシンで複数ノード起動できるようになっていない。そこで、複数台起動ができるように通信のためにポート番号の変更する。さらに多数台起動させるスクリプトを作成した。

しかしながら、1 ノードの Cassandra 起動するために、デフォルトの設定で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。またクラスタを構成した際には、さらに 1 ノードあたりのリソースが消費量が増える。多数第起動するには、1 ノード起動するためのリソースの消費を抑える必要があった。そこで、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、メモリー使用領域がかさむ。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、データを読み書きするオペレーションはない。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メモリー消費を削減した。

6.1.2　設定ファイルのパラメータ調整

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デフォルトの 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。

d また、これらのパラメータの調整も、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を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１０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[2] 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として、Yahoo! Cloud Serving Benchmark(YCSB) を使用した。YCSB[3] 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。

```
[okudera@lime01 ~]$ ulimit  -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 257704
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 1024
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

図 6.1: リソース制限変更前

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

```
[okudera@lime20 ~]$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 257704
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) unlimited
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

図 6.2: リソース制限変更後

るとさらに増加する。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。(図 6.1)

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。(図 6.2) その結果、多数台の起動が可能になった。

NFS を利用した書き込みをできるだけ控える　概して一度に多数ノードを起動するときは、実行ファイルを共通にして起動することが多い。すなわち、NFS 上に共通の実行スクリプトを保存し、実行時に読み込むことが多い。しかしながら、一度に非常に多数の NFS を通じた読み込みが発生すると、上手くアクセス出来ないことがある。そこで、各マシンのローカル上に実行ファイルを保管し、起動時にそのファイルを読み込むようにした。

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

```
eth0:1    Link encap:Ethernet  HWaddr 84:2B:2B:64:86:88
          inet addr:10.20.20.1  Bcast:10.20.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Interrupt:36 Memory:da000000-da012800

eth0:2    Link encap:Ethernet  HWaddr 84:2B:2B:64:86:88
          inet addr:10.20.20.2  Bcast:10.20.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Interrupt:36 Memory:da000000-da012800

eth0:3    Link encap:Ethernet  HWaddr 84:2B:2B:64:86:88
          inet addr:10.20.20.3  Bcast:10.20.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Interrupt:36 Memory:da000000-da012800

eth0:4    Link encap:Ethernet  HWaddr 84:2B:2B:64:86:88
          inet addr:10.20.20.4  Bcast:10.20.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Interrupt:36 Memory:da000000-da012800

eth0:5    Link encap:Ethernet  HWaddr 84:2B:2B:64:86:88
          inet addr:10.20.20.5  Bcast:10.20.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Interrupt:36 Memory:da000000-da012800

eth0:6    Link encap:Ethernet  HWaddr 84:2B:2B:64:86:88
          inet addr:10.20.20.6  Bcast:10.20.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Interrupt:36 Memory:da000000-da012800

eth0:7    Link encap:Ethernet  HWaddr 84:2B:2B:64:86:88
          inet addr:10.20.20.7  Bcast:10.20.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Interrupt:36 Memory:da000000-da012800

eth0:8    Link encap:Ethernet  HWaddr 84:2B:2B:64:86:88
          inet addr:10.20.20.8  Bcast:10.20.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Interrupt:36 Memory:da000000-da012800

eth0:9    Link encap:Ethernet  HWaddr 84:2B:2B:64:86:88
          inet addr:10.20.20.9  Bcast:10.20.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Interrupt:36 Memory:da000000-da012800

eth0:10   Link encap:Ethernet  HWaddr 84:2B:2B:64:86:88
          inet addr:10.20.20.10  Bcast:10.20.255.255  Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          Interrupt:36 Memory:da000000-da012800
```

図 6.3: ifconfig の実行結果

```
[okudera@lime19 bin]$ ./nodetool -h 10.20.20.1 -p 8081 ring
Address         Status     Load          Range                                      Ring
                                         169180376881269628403480787642570762075
10.20.18.6      Up         495 bytes     111137601887734927713944131253046931 87    |<--|
10.20.17.2      Up         495 bytes     148157107098467755162692802994318830 97    |   ^
10.20.17.1      Up         495 bytes     165902533493069610392417511102557390 27    v   |
10.20.17.6      Up         495 bytes     222948074890851451252305901334991175 98    |   ^
10.20.20.4      Up         495 bytes     260737714644069958792007755207994050 43    v   |
10.20.20.1      Up         495 bytes     268611722632005521728221236274434328 45    |   ^
10.20.16.4      Up         495 bytes     354214317199396727755129829150930972 19    v   |
10.20.20.2      Up         495 bytes     450342405595057589845999355708307319 37    |   ^
10.20.16.6      Up         495 bytes     452244743267908136695677157726651826 94    v   |
10.20.16.2      Up         495 bytes     573324237089598026526087705051933134 95    |   ^
10.20.18.5      Up         495 bytes     576082964356735723593756216323910687 17    v   |
10.20.20.3      Up         495 bytes     580488888457134846468146543846027556 59    |   ^
10.20.18.4      Up         495 bytes     605026404397668153244550088317240018 78    v   |
10.20.19.3      Up         495 bytes     654019063576531534936820234697014458 31    |   ^
10.20.19.4      Up         495 bytes     689166259749524832391671387072873295 25    v   |
10.20.16.1      Up         495 bytes     795254593413857388052220196156524558 75    |   ^
10.20.16.5      Up         495 bytes     816485947613841724314439841337413124 11    v   |
10.20.17.3      Up         495 bytes     905340957587054851338092243594173277 58    |   ^
10.20.19.6      Up         495 bytes     942760638233101145737682401029955119 34    v   |
10.20.18.3      Up         495 bytes     964280130032801110157048653304313196 87    |   ^
10.20.20.5      Up         495 bytes     115536254213812968781603144480528659 407   v   |
10.20.17.4      Up         495 bytes     116057395662412152539005385433006110 363   |   ^
10.20.18.1      Up         495 bytes     118331362875818037407837981329128097 759   v   |
10.20.20.6      Up         495 bytes     120177157666681819044802242041320586 570   |   ^
10.20.17.5      Up         495 bytes     140038670664330997718516444687165924 160   v   |
10.20.19.1      Up         495 bytes     140746655400214165971187825325530326 086   |   ^
10.20.19.2      Up         495 bytes     149494985536248049937854533104206081 972   v   |
10.20.18.2      Up         495 bytes     151768813669109044693976960569868080 469   |   ^
10.20.16.3      Up         495 bytes     164425287327808289139031510855023468 502   v   |
10.20.19.5      Up         495 bytes     169180376881269628403480787642570762 075   |-->|
```

図 6.4: Cassandra クラスタを構成したときの様子

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。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 (図 6.3) 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 6.4)

```
15:10:44.202766 IP (tos 0x0, ttl 64, id 42399, offset 0, flags [DF], proto TCP (6), length 60)
    10.20.17.6.34338 > 10.20.19.16.afs3-callback: Flags [S], cksum 0x51b4 (correct), seq 674710703, win 5840, option
s [mss 1460,sackOK,TS val 3081494832 ecr 0,nop,wscale 7], length 0
15:10:44.202800 IP (tos 0x0, ttl 64, id 43887, offset 0, flags [DF], proto TCP (6), length 60)
    10.20.17.4.58813 > 10.20.19.15.afs3-callback: Flags [S], cksum 0xed30 (correct), seq 661801567, win 5840, option
s [mss 1460,sackOK,TS val 3081494832 ecr 0,nop,wscale 7], length 0
15:10:44.202830 IP (tos 0x0, ttl 64, id 10585, offset 0, flags [DF], proto TCP (6), length 60)
    10.20.18.6.52625 > 10.20.19.17.afs3-callback: Flags [S], cksum 0x37b1 (correct), seq 2554030828, win 5840, optio
ns [mss 1460,sackOK,TS val 2423800757 ecr 0,nop,wscale 7], length 0
15:10:44.202992 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 40)
    10.20.17.20.afs3-callback > 10.20.19.5.60264: Flags [R.], cksum 0x1059 (correct), seq 0, ack 666515643, win 0, l
ength 0
15:10:44.203024 IP (tos 0x0, ttl 64, id 27190, offset 0, flags [DF], proto TCP (6), length 60)
    10.20.17.6.57470 > 10.20.19.14.afs3-callback: Flags [S], cksum 0xb190 (correct), seq 665422598, win 5840, option
s [mss 1460,sackOK,TS val 3081494832 ecr 0,nop,wscale 7], length 0
15:10:44.203937 IP (tos 0x0, ttl 64, id 4156, offset 0, flags [DF], proto TCP (6), length 60)
    10.20.17.4.43489 > 10.20.19.18.afs3-callback: Flags [S], cksum 0x81b7 (correct), seq 672526605, win 5840, option
s [mss 1460,sackOK,TS val 3081494833 ecr 0,nop,wscale 7], length 0
15:10:44.203988 IP (tos 0x0, ttl 64, id 9900, offset 0, flags [DF], proto TCP (6), length 60)
    10.20.17.6.46065 > 10.20.19.8.afs3-callback: Flags [S], cksum 0xa0f4 (correct), seq 675334045, win 5840, options
 [mss 1460,sackOK,TS val 3081494833 ecr 0,nop,wscale 7], length 0
15:10:44.204037 IP (tos 0x0, ttl 64, id 50777, offset 0, flags [DF], proto TCP (6), length 60)
    10.20.17.6.50612 > 10.20.19.13.afs3-callback: Flags [S], cksum 0x2fb5 (correct), seq 669853544, win 5840, option
s [mss 1460,sackOK,TS val 3081494833 ecr 0,nop,wscale 7], length 0
15:10:44.204811 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 40)
    10.20.18.16.afs3-callback > 10.20.19.5.60779: Flags [R.], cksum 0x0b35 (correct), seq 0, ack 675035742, win 0, l
ength 0
15:10:44.205000 IP (tos 0x0, ttl 64, id 0, offset 0, flags [DF], proto TCP (6), length 40)
    10.20.16.11.afs3-callback > 10.20.19.2.38144: Flags [R.], cksum 0xc1c4 (correct), seq 0, ack 670097037, win 0, l
ength 0
15:10:44.207107 IP (tos 0x0, ttl 64, id 7951, offset 0, flags [DF], proto TCP (6), length 60)
    10.20.18.1.42578 > 10.20.19.15.afs3-callback: Flags [S], cksum 0x6d08 (correct), seq 2556189878, win 5840, optio
ns [mss 1460,sackOK,TS val 2423800761 ecr 0,nop,wscale 7], length 0
15:10:44.207692 IP (tos 0x0, ttl 64, id 9446, offset 0, flags [DF], proto TCP (6), length 60)
    10.20.17.3.33428 > 10.20.19.16.afs3-callback: Flags [S], cksum 0xd425 (correct), seq 674088403, win 5840, option
s [mss 1460,sackOK,TS val 3081494837 ecr 0,nop,wscale 7], length 0
15:10:44.207810 IP (tos 0x0, ttl 64, id 19334, offset 0, flags [DF], proto TCP (6), length 60)
    10.20.18.3.40223 > 10.20.19.11.afs3-callback: Flags [S], cksum 0xcb99 (correct), seq 2541291580, win 5840, optio
ns [mss 1460,sackOK,TS val 2423800762 ecr 0,nop,wscale 7], length 0
```

図 6.5: tcpdump 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。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。

6.2.4　通信量の推定

6.2.4.1　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。

m 台の各マシンで cassandra$n$’ ノード (ただし、n’ > 2 とする) 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、

$$T \quad = \quad Tt * [(n' * m) - 1/((n' - 1) * m)] \tag{6.1}$$

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。具体的に n=4,m=2 として説明する。マシン２台上でそれぞれ Cassandra を２ノードたちあげてるい。また、各マシンは受信する通信量しか計測していないことに注意する。このとき、測定可能な通信量から、測定不可能な通信量を見積もる。マシン A が測定しているのは、ノード 3,4 からノード 1 への通信とノード 3,4 からノード 2 への通信である。同様にマシン B が測定しているのは、ノード 1,2 からノード 3 への通信とノード 1,2 からノード 4 への通信である。それぞれ、Ta,Tb とおく。また、測定出来てい

ないのは、ノード 1 からノード 2、ノード 2 からノード 1、ノード 4 からノード 3、ノード 3 からノード 4 の通信である。これをまとめて Tc とおく。Ta,Tb,Tc ともに 4 つの Cassandra ノード同士の通信である。

上の仮定を利用すると、Tc は、マシン A、B で測定した通信から見積もることができる。つまり、

$$Tc = (Ta + Tb)/2 \tag{6.2}$$

よって Cassandra ノード間で発生する通信は、

$$\begin{aligned} T &= (Ta + Tb)/2 + Ta + Tb && (6.3)\\ &= (Ta + Tb) * 3/2 && (6.4)\\ &= (Ta + Tb) * (2 * 2 - 1)/[(2 - 1) * 2] && (6.5) \end{aligned}$$

と推測できる。

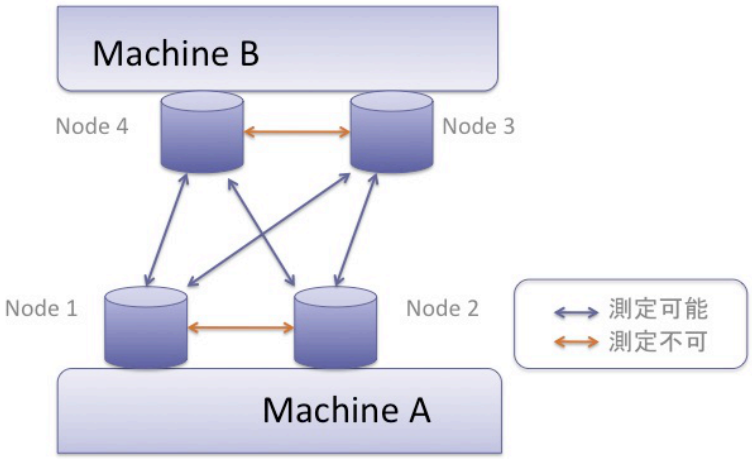

図 6.6: gossip 通信量の推定

6.2.4.2　実験 2:データ読み書きの通信量の推定

非集中な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 2 で gossip プロトコルの通信量は測定しているので、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。また,gossip による通信は、メンバー構成が安定しているときにはデータの読み書きによらず一定であることに留意する。
具体的に $n = 4$、$m = 2$ のケースについて図をもちいて説明していく。まず、YCSB クライアントを解説する。

YCSB のクライアントは、Cassandra 1 ノードだけと接続して (これを proxy ノードという。) データの読み書きを行う。データの書き込み、読み込み時の Cassandra ノード間での通信の流れを説明する。また今回はレプリカを作成していない。データ書き込み時には、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しらせる。いわゆる Ack を返す。データ読込時には、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。データ通信量の推定値 T は、proxy ノードが動作するマシンで取得した通信量を TA,proxy 以外のノードが動作するマシンで取得した通信量の合計を TB として、それぞれ gossip の通信量を TAg,TBg とおく。

$$T = [(TA - TAg) + (TB - TBg)] * [(n-1)/n] * [m/(m-1)] \quad (6.6)$$

で得られる。その理由を説明する。具体的に $n = 4$、$m = 2$ として考える。書き込み時、読み込み時を分けて推定をおこなう。

書き込み時　書き込み時に発生する通信は図 6.7 である。proxy ノードが動作するマシンで取得した ack 通信量を TAa,proxy 以外のノードが動作するマシンで取得した通信量の合計を TBd とおく。すなわち

$$TA = TAg + TAa \quad (6.7)$$

$$TB = TBg + TBd \quad (6.8)$$

が成立している。各マシンで取得した通信量から gossip による通信量を差し引く。これはデータの読み書きに応じて gossip 通信量が変化しないことから問題ない。gossip の通信量は、データの読み書きをする前後で計測可

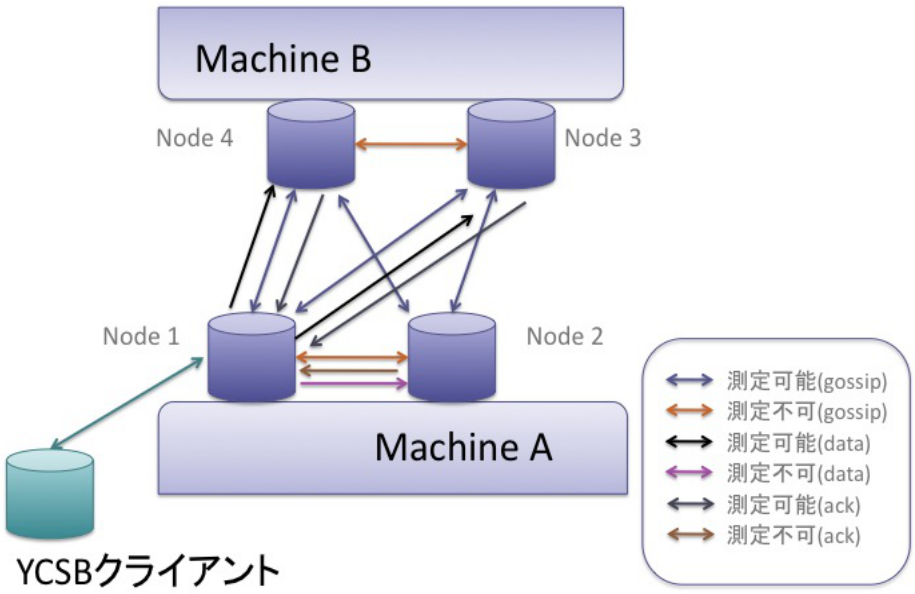

図 6.7: データ通信量の推定 -書き込み時 1-

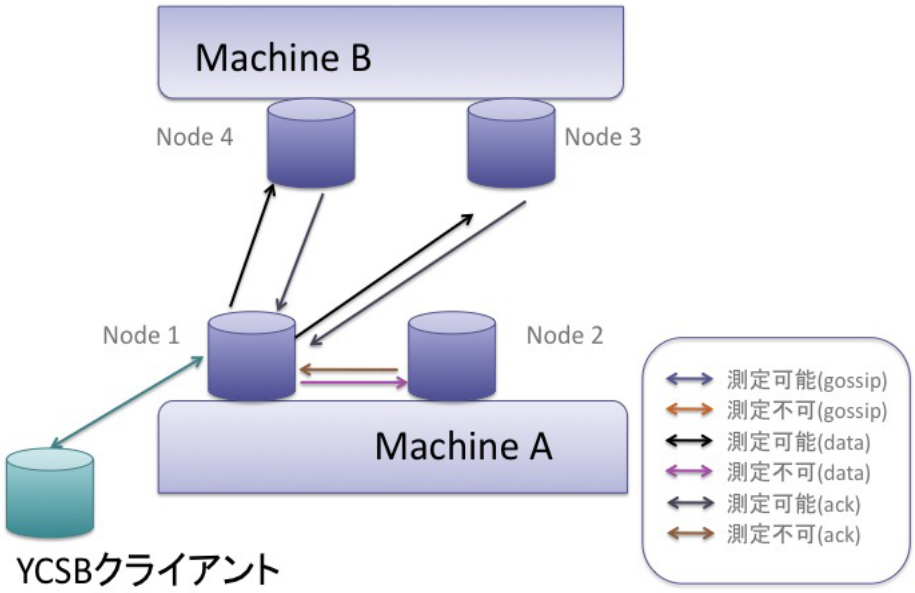

図 6.8: データ通信量の推定 -書き込み時 2-

能である。gossip による通信を除いた通信は、proxy ノードから proxy 以外へデータを転送する通信と、proxy 以外のノードから proxy への *ack* の

通信である。これを表したのが図 6.8 である。このとき、Cassandra ノード全体で発生する ack の通信量を Ta とおけば、

$$Ta = TAa * 3/2 \tag{6.9}$$

Cassandra ノード全体で発生する data の通信量を Td とおけば、

$$Td = TBd * 3/2 \tag{6.10}$$

一方、

$$T = Ta + Td \tag{6.11}$$

であるので、

$$\begin{aligned} T &= (TAa + TBd) * 3/2 && (6.12)\\ &= [(TA - TAg) + (TB - TBg)] * 3/2 && (6.13) \end{aligned}$$

である。

読み込み時　読み込み時も同様である。読み込み時の通信の流れを図 6.9 に表した。gossip による通信を抜いた通信の流れが図 6.10 である。書き込み時におけるデータによる通信をデータリクエストに、ack をデータによる通信に置き変えたのが読み書き時の通信の流れになる。すなわち、proxy ノードが動作するマシンで取得したデータ通信量を TAd,proxy 以外のノードが動作するマシンで取得したリクエストの通信量の合計を TBr とおく。このとき、

Cassandra ノード全体で発生するデータの通信量を Td とおけば、

$$Td = TAd * 3/2 \tag{6.14}$$

Cassandra ノード全体で発生するリクエストの通信量を Tr とおけば、

$$Tr = TBr * 3/2 \tag{6.15}$$

一方、

$$T = Td + Tr \tag{6.16}$$

であるので、

$$\begin{aligned} T &= (TAd + TBr) * 3/2 && (6.17)\\ &= [(TA - TAg) + (TB - TBg)] * 3/2 && (6.18) \end{aligned}$$

である。

以上より、読み書き両方においてデータの通信量は、式 6.6 で推測できることがわかった。

6.3　通信量測定のためのプログラムの作成

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。(図 6.11)

6.3.1　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
- JVM: Java SE 6 Update 21
- Memory: 32GB RAM
- Network: 1000BASE-T

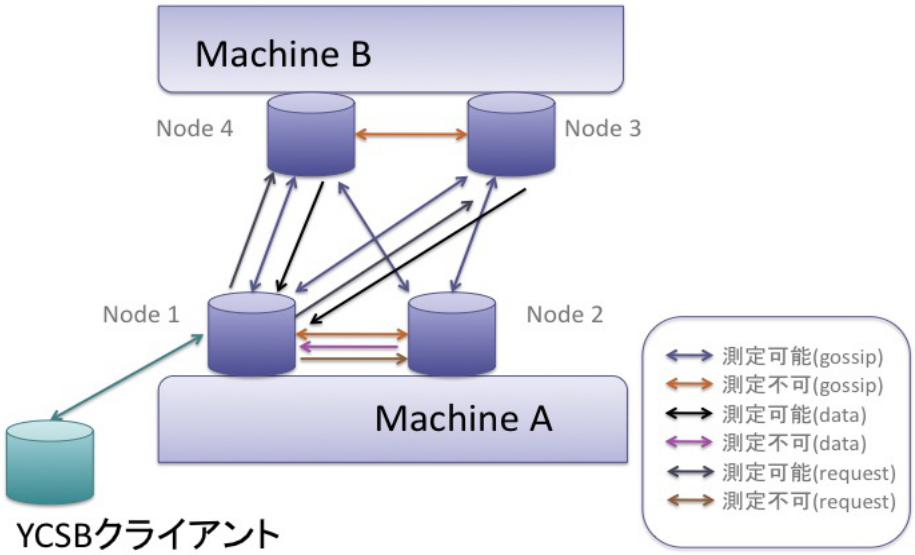

図 6.9: データ通信量の推定 -読み込み時 1-

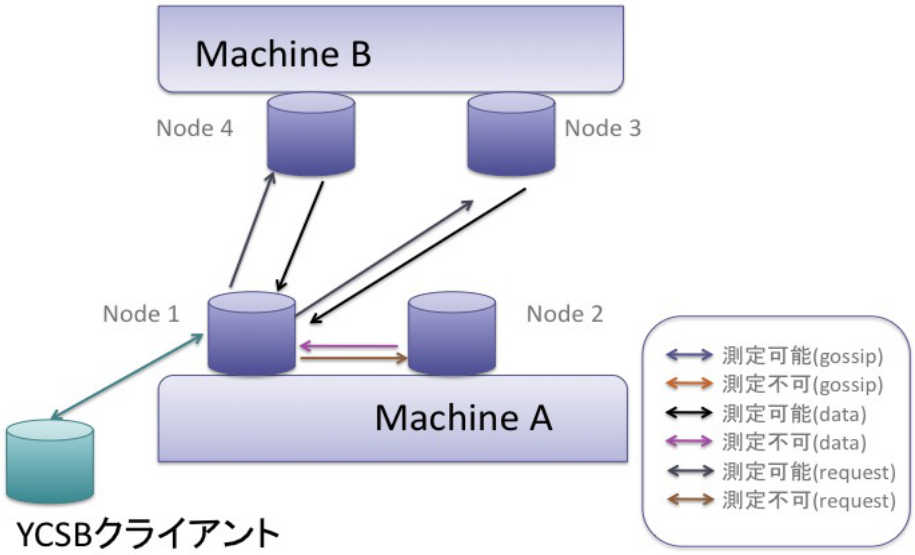

図 6.10: データ通信量の推定 -読み込み時 2-

```
#! /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。
#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。
    . $E_PARA_INCLUDE
#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。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、以下が行われる。
#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.計測した通信量情報から必要な情報だけを取得。各ノードで解析を行う ===
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

#データを集約してから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 6.11: 実行スクリプト

第7章　実験・評価

7.1　予備実験

本実験に入る前に、予備実験を行った。

7.1.1　予備実験 (1) gossip 通信量推定の妥当性

この実験によって、Cassandra ノードで発生する総通信量の推定が妥当であることを裏付ける。異なるマシン数で同数の Cassandra ノードを立ち上げて通信量の推定を行い比較する。予備実験では、120 台の Cassandra ノードを、TypeA,TypeB の 2 パターンで立ち上げて 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１２ノードの Cassandra を立ち上げたマシン 10 台でのクラスタを構成
- typeB
  一台あたり６０ノードの Cassandra ノードを立ち上げたマシン 2 台でのクラスタを構成

Type A の場合は、時間 t の時の通信量を A(t) とおくと、

$$[\text{推定される通信量}](t) = A(t) * 10/9 \tag{7.1}$$

Type B の場合は、時間 t の時の通信量を B(t) とおくと、

$$[\text{推定される通信量}](t) = B(t) * 2/1 \tag{7.2}$$

となる。各通信量の推定値をグラフに表したのが図 7.1 である。

先ほどと同様に、横軸は時間軸を示していて、縦軸は推定される通信量 (単位は M bit) である。図 7.1 からマシン数が 2 台、10 台のときともに、200 秒以降は通信量が安定していることがわかる。安定したときには、推定される通信量がほぼ等しいこともわかる。よって、この推定が確からしいことが裏付けられる。

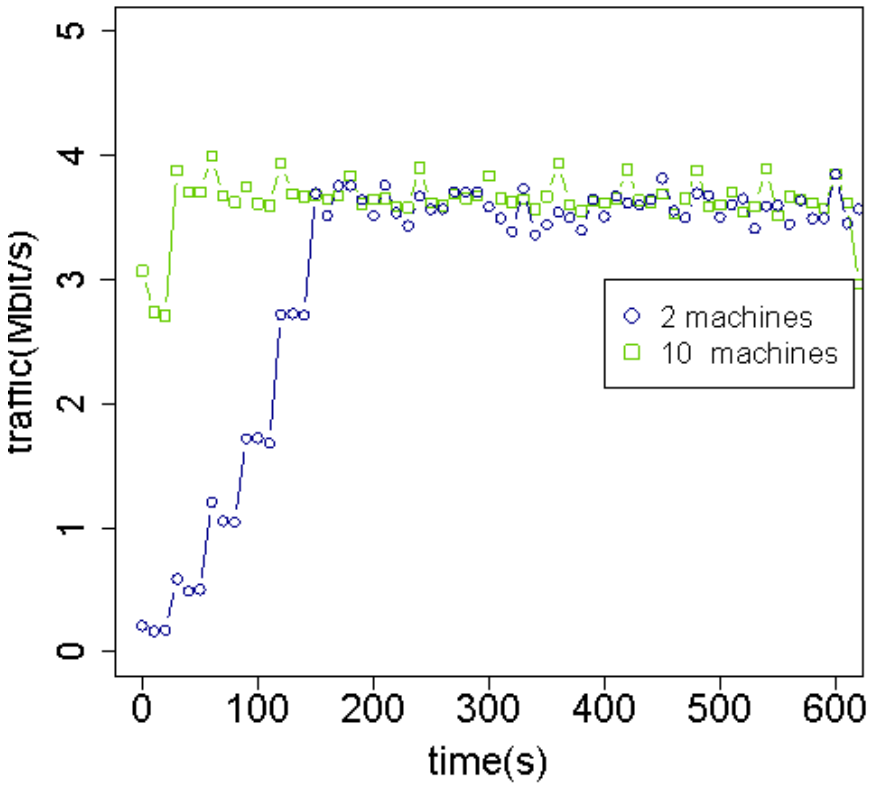

図 7.1: マシン数に応じたトラフィックの時間変化

7.1.2　予備実験 (3):データ通信量推定の妥当性

予備実験 (1) gossip 通信量推定の妥当性と同様に、Cassandra ノードで発生するデータ通信量の推測が妥当であることを裏付ける。そのために、異なるマシン数で同数の Cassandra ノードを立ち上げてデータ通信量の推定を行い比較する。ノード台数は 120 台と固定して、マシン数を 3,4,5 と変化させて通信量の推定を行った。QPS=5000 とした。その結果が図 7.2 である。

構成するマシンが 3 台の時、通信量が 42.3Mbit/s、マシンが 4 台の時 42.6Mbit/s、マシンが 5 台の時 43.6Mbit/である。マシン台数が増えるにつれ、わずかに通信量も増加しているが、最大 3.1%の誤差を含み、推定値がほぼ同じであることが確かめられる。よってこの推定が確からしいことが裏付けられた。

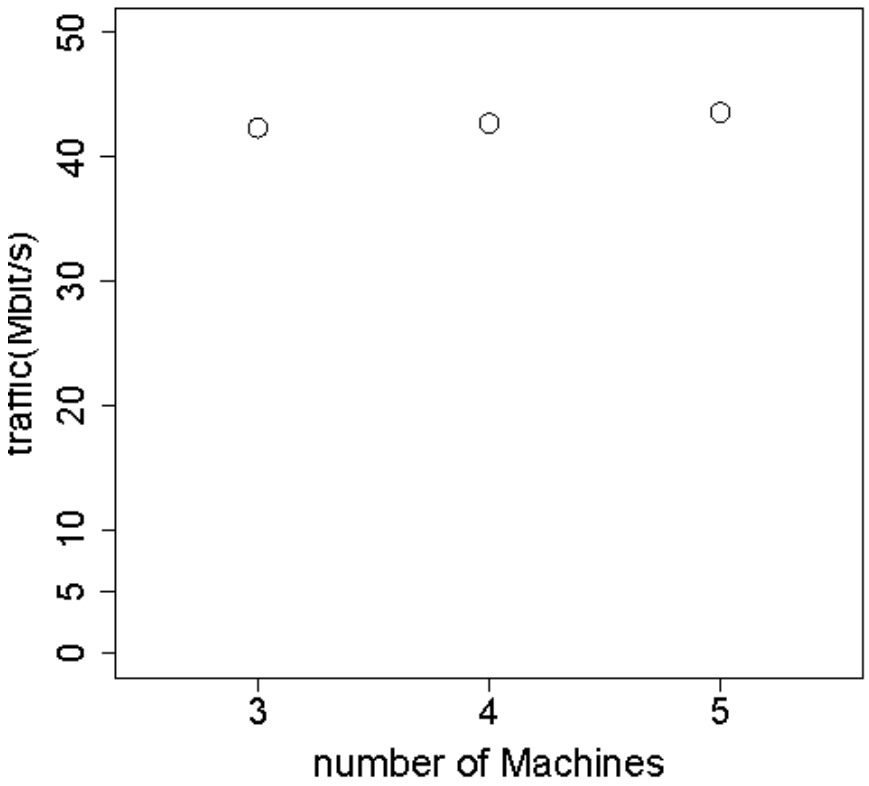

図 7.2: マシン台数に応じたデータ読み書きの通信量推定値の変化

7.1.3 予備実験 (4):データ量に応じた Cassandra 内部での総通信量の変化

要求されるリクエストの量 (Queries/s) を変化させたときに、Cassandra 内部で発生する総通信量 (bit/s) の変化を測定する。データ量に応じて線形に通信量が増加していくことが予想される。以下がその測定結果である。ノード台数は 120 台と固定し、Q = 500, 1000,2000,3000,4000 と変化させてデータ通信料を測定した。(図 7.3) 縦軸が 1 秒あたりのクエリー発行数 (Queries/s) であり、横軸が総通信量である。

図中の曲線は、プロットした点から 1 次関数でフィッティングしたものである。q を 1 秒あたりのクエリー発行数として得られる関数は、

$$[\text{通信量 } (bit/s)] = 8727.9 \times q - 348939 \tag{7.3}$$

である。通信量は 1 秒あたりのクエリー数に応じて O(n) で増加していくことが確認できる。予想通りの結果が得られた。

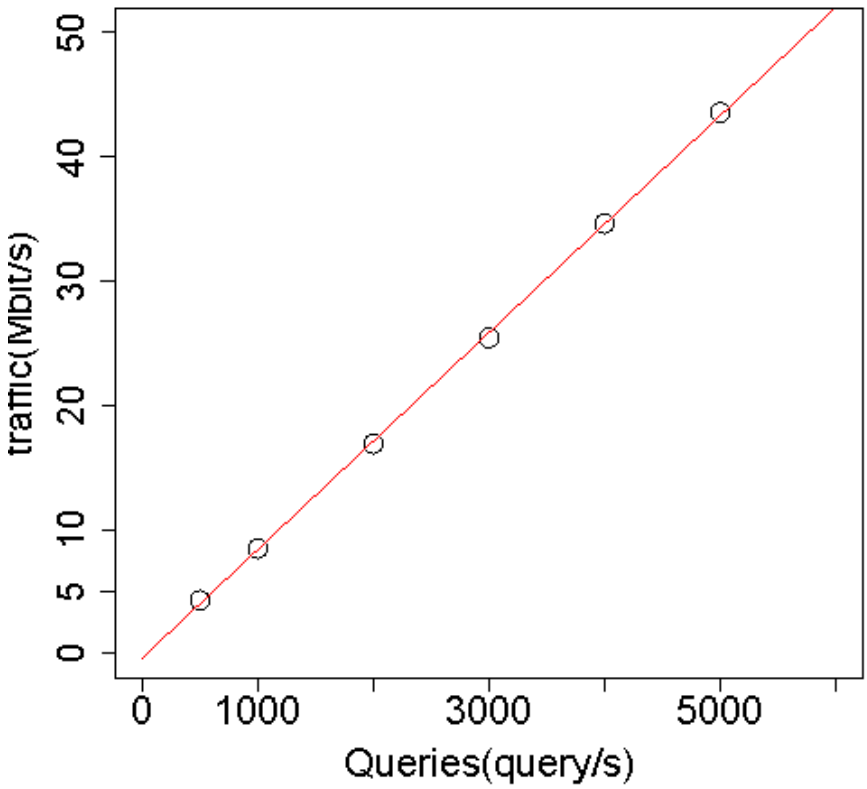

図 7.3: データ量に応じた Cassandra 内部での総通信量の変化

7.2 本実験

7.2.1 実験 1:gossip の通信量

図 7.4 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 7.5 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。

7.2.2 実験 2:データ読み書きの通信量の測定

図 7.6 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 で固定してある。

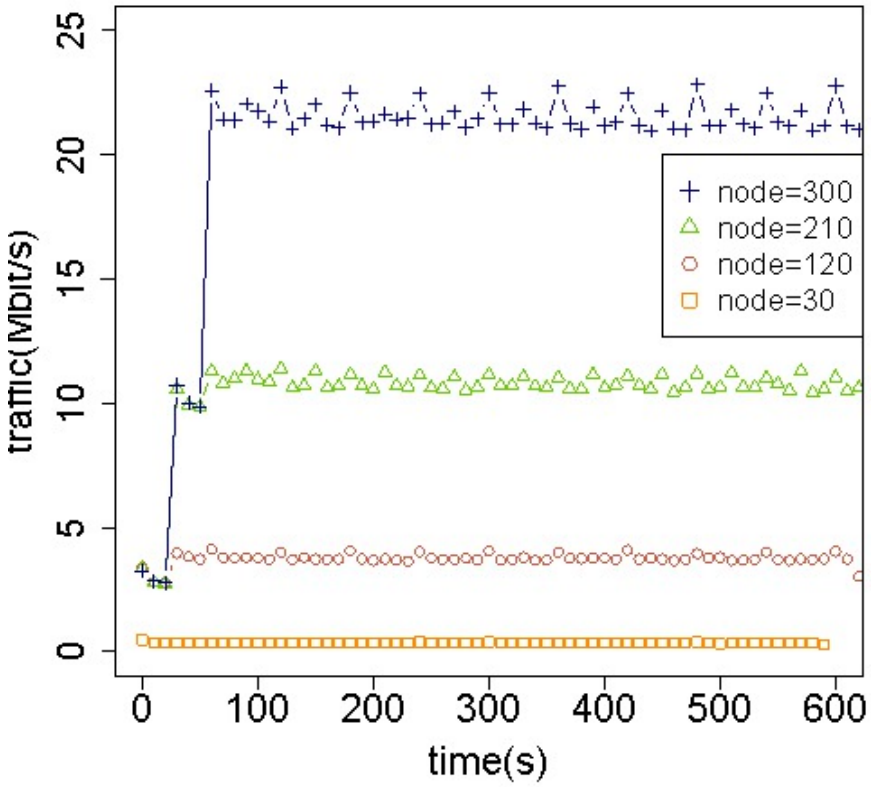

図 7.4: infer-traffic の実行結果

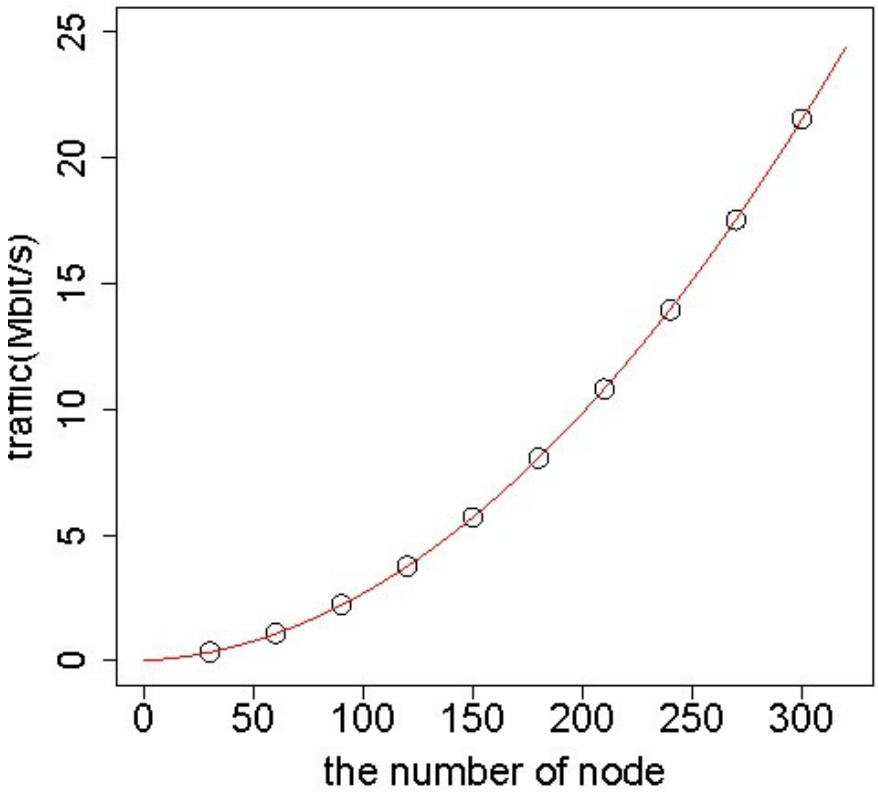

図 7.5: infer-traffic の実行結果

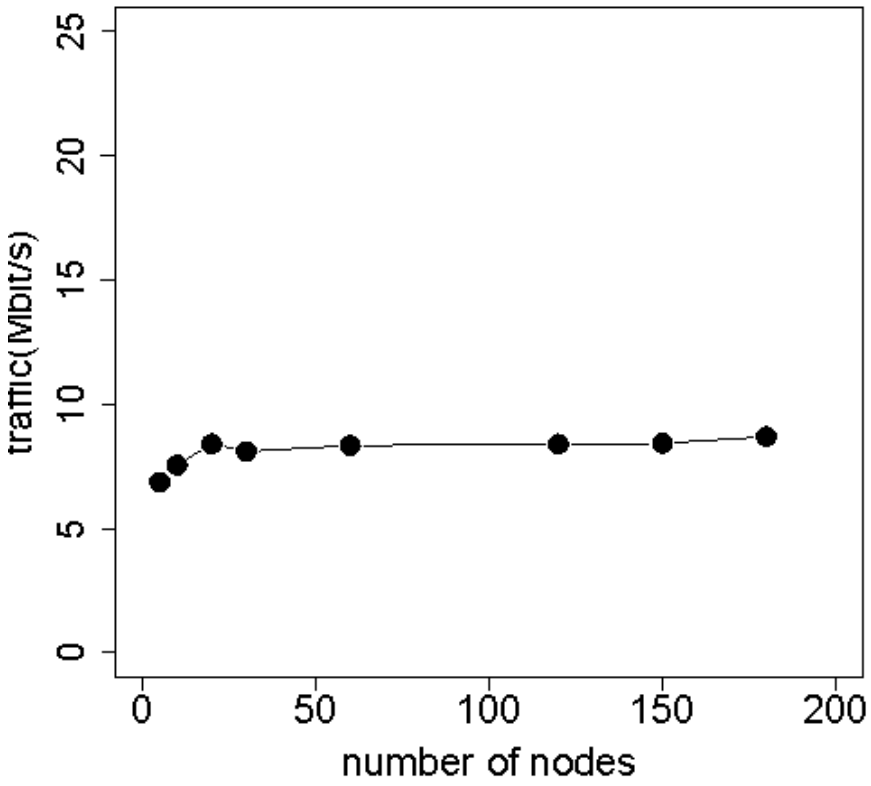

図 7.6: ノード台数に応じたデータ読み書きの通信量の変化

7.3　評価

7.3.1　gossip 通信量の見積もり

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、gossip の通信量

$$[\text{通信量}\ (bit)] = 224.6 \times n^2 + 4314.8 \times n \tag{7.4}$$

である。よって、通信量は $O(n^2)$ で増加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。このように、この関数を使って gossip による通信量が見積もることができる。

gossip プロトコルによる故障検知に必要な通信量が $O(n^2)$ で増加していくということは、スケールしにくいことを表している。つまりノード台数を増加してスケーラビリティを確保するときに gossip における通信量がボトルネックになりうるのである。

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。ここでは受信する通信量のみを考えるが、受信する通信量と送信する通信量は同じである。総通信量を Cassandra ノード台数 n で割った値が、1 ノードあたりの通信量となる。つまり、

$$[\,1\ \text{ノードあたりの通信量}] = (224.6 \times n^2 + 4314.8 \times n)/n = 224.6 \times n + 4314.8 \tag{7.5}$$

と O(n) で増加することがわかる。

7.3.3　複数のデータセンターをまたぐクラスタにおける gossip プロトコルの問題点

通信量を見積もるケースとして 2 つのデータセンターをまたいでクラスタを構成することを考える。(図 7.7)

このようにデータセンターを複数またいでクラスタを構成することは特別なことではない。データの複製を別のデータセンターに保持しておくことで、データセンターごと故障が起きてもデータの損失が避けられる。このように最近のクラウドストレージではデータセンターの故障にも耐性があるシステムを実現しているのである。

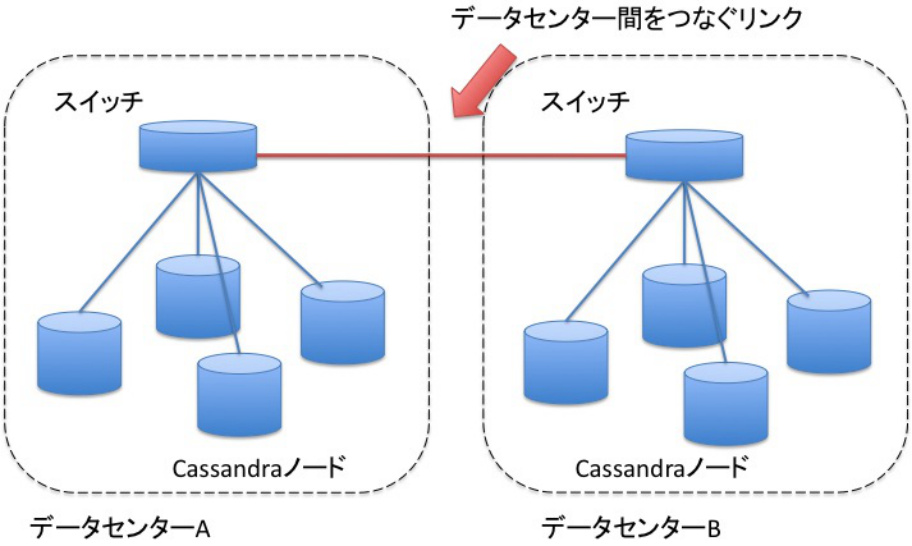

図 7.7: データセンター間をまたぐクラスタの構成

データセンター間を結ぶリンクで発生する通信量を見積もる。簡単のために Cassandra ノード台数を n とした時、各データセンター A,B にそれぞれ $n/2$ ノードの Cassandra が起動しているケースを考える。この時データセンター間での通信量 TAB は、システム全体で発生する通信量を T とおくと、$T/4$ である。一方、式 7.4 より T は $\mathrm{O}(n^2)$ で増加するから TAB も $\mathrm{O}(n^2)$ で増加する。

このように、クラスタを構成する各ノードの通信量は $\mathrm{O}(n)$ で増加していく一方で、データセンター間を結ぶリンの通信量は $\mathrm{O}(n^2)$ で増加していく。つまり、ノード台数が増加したときにリンク部分の通信が圧迫される可能性がある。現状の gossip プロトコルには、このようなデータセンター間のリンク部分を考慮したアルゴリズムではないことが確認できる。

データセンターをまたいでクラスタを構成することが主流の現在において、リンク部分の通信量を考慮して gossip プロコルを応用することが望まれる。例えば、あるノードの故障情報をデータセンターをまたいで伝達させたいとき、現状の gossip プロトコルでは同じ情報を冗長に伝達することが多い。データセンターをまたぐ通信を取りまとめることができれば、データセンター間のリンクで発生する通信を削減できるだろう。

7.3.3.1　gossip の通信量が O(n^2) で増加していく理由

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、

$$T = T' * n \tag{7.6}$$

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、

$$T' \quad = \quad s * tg \tag{7.7}$$

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、

$$s < 2 = O(1) \tag{7.8}$$

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、

$$tg = O(n) \tag{7.9}$$

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、

$$T' \quad = \quad O(n) \tag{7.10}$$

$$T \quad = \quad O(n^2) \tag{7.11}$$

となることが証明できた。

7.3.3.2　データ読み書きの通信量は一定

グラフ図 4 から、QPS を一定にしたときに、構成する Cassandrano ノード台数 n に応じて、増加が緩やかになり一定の値に近づいていくことが確認できた。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 ノードがあるデータの担当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ことになる。よって、n が大きくなるとその確率は 1 に収束してのでほぼ必ずリクエストに対して通信が発生する。ノード台数 n がふえると、通信量はある一定値に近づいていく。

第8章　結論

8.1　まとめ

本研究では非集中なクラウドストレージでの主要なメンバーシップ管理である gossip プロトコルについてスケーラビリティを Cassandra を使用して実験・評価した。計測の結果 gossip プロトコルによるシステム全体の通信量はノード台数を n としたとき O(n^2) で増加することを確認した。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。この結果は、クラスタ設計時等で gossip で発生する通信量の見積もり際に重要である。２つのデータセンター間をまたぐリンクにおいては、O(n^2) で通信量が増加しスケーラビリティの制約になりかねない。このようなケースを考慮した gossip プロトコルの応用した提案が望まれる。

8.2　今後の課題

今後の課題について 3 つ記す。

１つ目は、gossip プロトコルを採用しているクラウドストレージにおけるスケーラビリティ評価を、別の切り口から検討していくことである。参照 [8] でもあるように、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。本研究では通信量の観点からスケーラビリティの評価を行ったが、これらの指標からもスケーラビリティの評価が行える。通信量と比べてどちらがさきにスケールするためのボトルネックとなりうるか議論ができる。

２つ目は、gossip プロトコル以外の故障検知アルゴリズムの通信量を評価し、比較することである。本研究では、gossip プロトコルのスケーラビリティ評価を行った。しかしながら、gossip 以外にも故障検知メカニズムは提案されている。例えば、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。これらのスケーラビリティ評価をし、比較することで gossip や他のアルゴリズムの特性を評価できる。

3 つ目は、データセンター間のリンクでの通信量を考慮した gossip プロトコルの提案である。データセンター間のリンクでは、$O(n^2)$ で通信量が増加しスケーラビリティの制約になりかねない。そこで、このようなリンクでの通信量を抑えるアルゴリズムが必要である。

参考文献

[1] : Cassandra Wiki.

[2] : GXP:Grid and Cluster Shell.

[3] Benchmarking Cloud Serving Systems with YCSB (2010).

[4] Dynamo:Amazon'sHighly Available Key-value Store (2007).

[5] CREW: A Gossip-based Flash-Dissemination System (2006).

[6] Cassandra -A Decentralized Structured Storage System (2009).

[7] CLON: Overlay Networks and Gossip Protocols for Cloud Environments, (2008).

[8] Failure Detection in Large Scale Systems: a Survey (2008).

[9] JetStream: probabilistic contour extraction with particles (2001).

[10] Efficient Reconciliation and Flow Control for Anti-Entropy (2008).