

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な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

第1章 序論	7
1.1 本研究の背景	7
1.2 本研究の目的	8
1.3 本研究の成果	8
1.4 本研究の構成	8
第2章 研究背景	10
2.1 dependable システムを実現するには？	10
2.2 dependable システムを実現するには故障検知が欠かせない	10
2.3 Javassist	10
2.4 Addistant	10
第3章 関連研究	11
3.1 関連研究 1	11
3.2 関連研究 2	11
第4章 Cassandra について	12
4.1 Cassandra の概要	12
4.2 Cassandra の軽量化	12
4.2.1 プログラムの改変	12
4.2.2 設定ファイルのパラメータ調整	13
第5章 測定手法	14
5.1 実験シナリオ	14
5.1.1 実験環境	14
5.1.2 計測方法について	15
5.2 通信量の測定について	16
5.3 自動で測定をして、解析まで行うプログラムの作成	17
第6章 実験・評価	22
6.1 予備実験	22
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。	22

6.1.2	予備実験 (2)	23
6.2	本実験	23
6.3	評価	23
6.3.1	総通信量の見積もり	23
6.3.2	1 ノードあたりの通信量	23
6.3.3	システム全体の限界とは？	24
6.3.4	通信量が $O(n^2)$ でスケールしていく理由	24
6.3.5	Cassandra のメンバーシップ管理の実装	24
6.3.6	数式から説明	24
第 7 章	結論	29
7.1	まとめ	29
7.2	今後の課題	29
7.2.1	.gossip protocol を別の切り口から評価する	29
7.2.2	Gossip Protocol 以外のメンバーシップ管理の評価	29
付 録 A	プログラム例	31

図 目 次

5.1	ifconfig の実行結果	18
5.2	Cassandra クラスタを構成したときの様子	19
5.3	tcpdump の実行結果	20
5.4	実行スクリプト	21
6.1	infer-traffic の実行結果	26
6.2	infer-traffic の実行結果	27
6.3	infer-traffic の実行結果	28

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 dependable システムを実現するには？

いろいろあるよね。そのなかでも、

2.2 dependable システムを実現するには故障検知が 欠かせない

あああ。

2.3 Javassist

Javassist [3, 1] は...

Javassist は [2] で配布されている。

2.4 Addistant

第3章 関連研究

3.1 関連研究1

例のサーベイ論文を引用し、大規模環境における故障検知について、解説。

3.2 関連研究2

具体的に gossip-base プロトコルを引用し、解説（これ、例の flow なんちゃらの論文を読むことになりそうだなあ。。）

第4章 Cassandraについて

4.1 Cassandraの概要

- ・メンバーシップ周り？
- ・故障検知（もしかしたらリングの図とかはいるかもね。）
- ・seedの説明
- ・データ保管部分の解説
- ・ある特定のポートで通信を行なっている。
- ・cassandraは、IPアドレスのみでノードを判定している。

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い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 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

計測にわたって、いくつか工夫した点を紹介する。

- ユーザーのリソース制約を外す
(これをもっと具体的に掘り起こしていこうか)
(通常の linux では、一人のユーザーがリソース資源を専用しないために、OS が制限を加えている。) そのため、まるまるできなかった。デフォの設定が〇〇で何台までしか起動できなかった。と。特に、Cassandrad はメモリー使用量域、使用するスレッド数が多いため、ユーザーに与えられてユーザーリソース制約にぶち当たることがある。
具体的にいくつか? ulimit でそれを外した。エラーがなくなった。
今回プロセス数、メモリー数の制約を外した。
- 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した
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、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 マシン同士の見分けが付き、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。ここでノイズとは、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。これらが誤って計測結果に加わることを避けるために、IP エイリアシ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 のネットワークを用いた。さらに Cassandra ノードがどのマシン上で起動しているかを判別しているために、クラスタ番号を n として、10.20. n .0/24 なるサブネットを仮想的に設けた。 $n=19$ のとき、10.20.19.1 ~ 10.20.19.254 までの仮想アドレスを作成した。以下が ifconfig コマンドを実行したときの実行結果と、プライベートアドレスで Cassandra クラスタを構成したときの様子である。

- 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 Node 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ため、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した。具体的には、tcpdump を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以下の二つの条件でフィルターをかけたものである。

- 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。

この二つの条件により、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

しかしながら、この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、我々は仮説をもとに、計測した総通信量から Cassandra Node で発生する総通信量を推測することにした。我々がたてた仮定である。

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

この仮説をもとにすると、 n 台の各マシンで m 台の `cassandra node` を起動したとする。(つまり、合計 $n \times m$ 台の `Cassandra ノード` を立ち上げたとする) さらに、上のように、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 とおき、`Cassandra node` で発生するそう通信量 TT とく。 $(n-1) \times m$ 台の `cassandra node` で発生している通信量が T であるので、仮説を使うと、 $n \times m$ 台の場合は、

$$TT = T * (n * m) / ((n - 1) * m) = T * n / (n - 1) \quad (5.1)$$

となる。具体的に、 $n = 10$ 台の時について、図を使いながら解説する。
(グラフ)

5.3 自動で測定をして、解析まで行うプログラムの作成

実際にソースコードを載せていろいろ言いましょう。

`measure` ラ変のソースを載せてしまおうか。//少しコメント書く！

```
eth0:1  Link encap:Ethernet  HWaddr 84:28:28:64:86:88
        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 84:28:28:64:86:88
        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 84:28:28:64:86:88
        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 84:28:28:64:86:88
        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 84:28:28:64:86:88
        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 84:28:28:64:86:88
        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 84:28:28:64:86:88
        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 84:28:28:64:86:88
        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 84:28:28:64:86:88
        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 84:28:28:64:86:88
        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 5.1: ifconfig の実行結果

```
[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	4314978364132992012674257271498299095	A
10.20.17.6	Up	425 bytes	4839881809623382721097958364920894855	v
10.20.20.1	Up	425 bytes	6827595557239396462236173296726362762	A
10.20.16.1	Up	425 bytes	7607640158418069473059629780680245157	v
10.20.17.4	Up	425 bytes	9389953706838658079423415178254007965	A
10.20.19.10	Up	425 bytes	9803386324366287984953454301612049702	v
10.20.18.3	Up	425 bytes	16824077227008375746864756517649110092	A
10.20.18.4	Up	425 bytes	16970089849287928520056185962916910711	v
10.20.19.4	Up	425 bytes	21453736472470189442045684688158253837	A
10.20.19.1	Up	425 bytes	30045533282186119755517114836312421390	v
10.20.17.8	Up	425 bytes	30561924667547409456492967919689178174	A
10.20.16.5	Up	425 bytes	35357694699336187159173035464032215701	v
10.20.19.9	Up	425 bytes	37269556680357219670698670544413970676	A
10.20.20.6	Up	425 bytes	42201755925953533763677908828016102717	v
10.20.16.9	Up	425 bytes	45010746386291484006566869831564051409	A
10.20.19.6	Up	425 bytes	46097269388138052453692171246390439488	v
10.20.17.1	Up	425 bytes	49103233791318210184954713826031057282	A
10.20.16.7	Up	425 bytes	49857719210968615789268421197456778750	v
10.20.18.10	Up	425 bytes	57193631033722536856660963931105198530	A
10.20.15.5	Up	425 bytes	57404447595623259094507184383606753641	v
10.20.15.2	Up	425 bytes	58123346321107105182797031907466841181	A
10.20.19.7	Up	425 bytes	58397253042327466340732240251736784947	v
10.20.20.3	Up	425 bytes	62652520733588842400562703813068799603	A
10.20.15.4	Up	425 bytes	66437957552106730255203940733580530584	v
10.20.20.8	Up	425 bytes	6668778531109562268269367850478860342	A
10.20.18.6	Up	425 bytes	67892381050658834195299438447613812892	v
10.20.16.4	Up	425 bytes	68995557168719190853308388570908661605	A
10.20.20.9	Up	425 bytes	71154629349397145370036212770545590353	v
10.20.18.8	Up	425 bytes	718621825003907965631928235215223294808	A
10.20.15.10	Up	425 bytes	77124014914662418489035269787332401382	v
10.20.18.9	Up	425 bytes	79979217022126509130544779191625322733	A
10.20.19.3	Up	425 bytes	80577754551354232895665885409255064784	v
10.20.15.8	Up	425 bytes	82195483320356584138198060570508746077	A
10.20.17.3	Up	425 bytes	87752048049037222814689026332340802798	v
10.20.16.3	Up	425 bytes	87789746831117404867567798220214325174	A
10.20.17.2	Up	425 bytes	90660644192024494777894760137774138468	v
10.20.16.6	Up	425 bytes	99407794183748826984736365446055424511	A
10.20.15.9	Up	425 bytes	103635175154239501346508063855898319482	v
10.20.19.5	Up	425 bytes	106044207090187810847241880724827998143	A
10.20.18.5	Up	425 bytes	109431520932365668592317396658483025231	v
10.20.18.2	Up	425 bytes	110536565009837493888526806665501824858	A
10.20.15.6	Up	425 bytes	113986445399625650015178405378662878842	v
10.20.18.7	Up	425 bytes	120431413577384292974893120015049683769	A
10.20.16.10	Up	425 bytes	121195808210942951839441858506370162981	v
10.20.15.1	Up	425 bytes	121806516210465232001823842096605993377	A
10.20.19.8	Up	425 bytes	125229918980056081877133717088313708594	v
10.20.19.2	Up	425 bytes	132916248192243341116893897139506414509	A
10.20.15.7	Up	425 bytes	137038795314537910543913190477304826209	v
10.20.16.2	Up	425 bytes	138496688040377870701804136013550144110	A
10.20.17.9	Up	425 bytes	139430788532221109981210738558903593155	v
10.20.20.4	Up	425 bytes	141050011155581257101026576501772996392	A
10.20.20.10	Up	425 bytes	145958602750794150728355155895296339431	v
10.20.18.1	Up	425 bytes	150753381137191738918435410201602841568	A
10.20.15.3	Up	425 bytes	152339177417673218930367959504259411623	v
10.20.16.8	Up	425 bytes	153431539485348575916518861163780764977	A
10.20.17.7	Up	425 bytes	154761433484781753832788085503046616299	v
10.20.17.10	Up	425 bytes	155433845433713829798253308110962076012	A
10.20.15.5	Up	425 bytes	156123608719484498978266840685280669276	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

```
[okudera@0]me15 measure-tool]$ sudo tcpdump -n -vvv -i eth0 src net 10.20.0.0/16 and dst net 10.20.15.0/24
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
19:14:55.174149 IP (tos 0x0, ttl 64, id 7851, offset 0, flags [DF], proto TCP (6), length 52)
    10.20.19.1.afs3-callback > 10.20.15.3.57913: Flags [.], cksum 0xfc64 (correct), seq 207677494, ack 210219408, win 231, options [
nop,nop,TS val 1886547388 ecr 1886536803], length 0
19:14:55.174864 IP (tos 0x0, ttl 64, id 62176, offset 0, flags [DF], proto TCP (6), length 980)
    10.20.19.1.51061 > 10.20.15.3.afs3-callback: Flags [P.], cksum 0xdf2c (correct), seq 217456594:217457522, ack 210104255, win 46,
options [nop,nop,TS val 1886547388 ecr 1886521804], length 928
19:14:55.176817 IP (tos 0x0, ttl 64, id 7852, offset 0, flags [DF], proto TCP (6), length 52)
    10.20.19.1.afs3-callback > 10.20.15.3.57913: Flags [.], cksum 0xfbf1 (correct), seq 0, ack 111, win 231, options [nop,nop,TS val
1886547390 ecr 1886536806], length 0
19:14:55.227984 IP (tos 0x0, ttl 64, id 28554, offset 0, flags [DF], proto TCP (6), length 52)
    10.20.16.1.afs3-callback > 10.20.15.8.45985: Flags [.], cksum 0xb40b (correct), seq 142390819, ack 154694363, win 176, options [
nop,nop,TS val 1886543781 ecr 1886536857], length 0
19:14:55.229323 IP (tos 0x0, ttl 64, id 28940, offset 0, flags [DF], proto TCP (6), length 835)
    10.20.16.1.46985 > 10.20.15.8.afs3-callback: Flags [P.], cksum 0x1503 (correct), seq 157338711:157339494, ack 150298975, win 46,
options [nop,nop,TS val 1886543782 ecr 1886529019], length 783
19:14:55.230815 IP (tos 0x0, ttl 64, id 28555, offset 0, flags [DF], proto TCP (6), length 52)
    10.20.16.1.afs3-callback > 10.20.15.8.45985: Flags [.], cksum 0xb28c (correct), seq 0, ack 366, win 189, options [nop,nop,TS val
1886543783 ecr 1886536860], length 0
19:14:55.268893 IP (tos 0x0, ttl 64, id 23898, offset 0, flags [DF], proto TCP (6), length 901)
    10.20.20.2.59376 > 10.20.15.9.afs3-callback: Flags [P.], cksum 0xe1fb (correct), seq 426930826:426931675, ack 423853900, win 46,
options [nop,nop,TS val 1886549872 ecr 1886469893], length 849
19:14:55.270588 IP (tos 0x0, ttl 64, id 57278, offset 0, flags [DF], proto TCP (6), length 52)
    10.20.20.2.afs3-callback > 10.20.15.9.48604: Flags [.], cksum 0x4f0a (correct), seq 425519268, ack 424631950, win 82, options [n
op,nop,TS val 1886549873 ecr 1886536900], length 0
19:14:55.271795 IP (tos 0x0, ttl 64, id 23899, offset 0, flags [DF], proto TCP (6), length 417)
    10.20.20.2.59376 > 10.20.15.9.afs3-callback: Flags [P.], cksum 0x0c45 (correct), seq 849:1214, ack 1, win 46, options [nop,nop,T
S val 1886549875 ecr 1886536898], length 365
19:14:55.326246 IP (tos 0x0, ttl 64, id 54289, offset 0, flags [DF], proto TCP (6), length 901)
    10.20.18.2.41527 > 10.20.15.6.afs3-callback: Flags [P.], cksum 0x926f (correct), seq 2238297682:2238298531, ack 387110483, win 4
6, options [nop,nop,TS val 1228851878 ecr 1886486952], length 849
19:14:55.327656 IP (tos 0x0, ttl 64, id 44258, offset 0, flags [DF], proto TCP (6), length 52)
    10.20.18.2.afs3-callback > 10.20.15.6.46979: Flags [.], cksum 0x2fcd (correct), seq 2242237003, ack 378341025, win 110, options
[nop,nop,TS val 1228851880 ecr 1886536957], length 0
19:14:55.328596 IP (tos 0x0, ttl 64, id 54290, offset 0, flags [DF], proto TCP (6), length 621)
    10.20.18.2.41527 > 10.20.15.6.afs3-callback: Flags [P.], cksum 0xac3e (correct), seq 849:1418, ack 1, win 46, options [nop,nop,T
S val 1228851881 ecr 1886536956], length 569
19:14:55.362159 IP (tos 0x0, ttl 64, id 61853, offset 0, flags [DF], proto TCP (6), length 901)
    10.20.18.6.56085 > 10.20.15.5.afs3-callback: Flags [P.], cksum 0x6fba (correct), seq 2004206773:2004207622, ack 137777753, win 4
6, options [nop,nop,TS val 1228851914 ecr 1886520994], length 849
19:14:55.363575 IP (tos 0x0, ttl 64, id 65172, offset 0, flags [DF], proto TCP (6), length 52)
    10.20.18.6.afs3-callback > 10.20.15.5.45707: Flags [.], cksum 0xaac0 (correct), seq 1991491251, ack 129941857, win 192, options
[nop,nop,TS val 1228851916 ecr 1886536993], length 0
19:14:55.364331 IP (tos 0x0, ttl 64, id 3739, offset 0, flags [DF], proto TCP (6), length 52)
    10.20.16.1.afs3-callback > 10.20.15.1.38164: Flags [.], cksum 0x3e12 (correct), seq 280819321, ack 282900075, win 98, options [n
op,nop,TS val 1886543917 ecr 1886536994], length 0
19:14:55.364518 IP (tos 0x0, ttl 64, id 61854, offset 0, flags [DF], proto TCP (6), length 502)
    10.20.18.6.56085 > 10.20.15.5.afs3-callback: Flags [P.], cksum 0xf1f3 (correct), seq 849:1299, ack 1, win 46, options [nop,nop,T
S val 1228851917 ecr 1886536992], length 450
19:14:55.365515 IP (tos 0x0, ttl 64, id 2814, offset 0, flags [DF], proto TCP (6), length 766)
    10.20.16.1.55618 > 10.20.15.1.afs3-callback: Flags [P.], cksum 0x1e2e (correct), seq 268433541:268434255, ack 278359428, win 46,
options [nop,nop,TS val 1886543918 ecr 1886451988], length 714
19:14:55.366660 IP (tos 0x0, ttl 64, id 64827, offset 0, flags [DF], proto TCP (6), length 52)
    10.20.17.4.afs3-callback > 10.20.15.5.39393: Flags [.], cksum 0xffff3 (correct), seq 307623322, ack 304276617, win 112, options [
nop,nop,TS val 1886545993 ecr 1886536996], length 0
19:14:55.367128 IP (tos 0x0, ttl 64, id 3740, offset 0, flags [DF], proto TCP (6), length 52)
    10.20.16.1.afs3-callback > 10.20.15.1.38164: Flags [.], cksum 0x3cd7 (correct), seq 0, ack 298, win 111, options [nop,nop,TS val
1886543920 ecr 1886536996], length 0
19:14:55.367492 IP (tos 0x0, ttl 64, id 60389, offset 0, flags [DF], proto TCP (6), length 52)
    10.20.20.2.afs3-callback > 10.20.15.4.52766: Flags [.], cksum 0x7481 (correct), seq 211348251, ack 204160848, win 139, options [
nop,nop,TS val 1886549970 ecr 1886536997], length 0
19:14:55.367760 IP (tos 0x0, ttl 64, id 27166, offset 0, flags [DF], proto TCP (6), length 810)
    10.20.17.4.50775 > 10.20.15.5.afs3-callback: Flags [P.], cksum 0x8f53 (correct), seq 311782645:311783403, ack 310963269, win 46,
options [nop,nop,TS val 1886545994 ecr 1886531748], length 758
19:14:55.368538 IP (tos 0x0, ttl 64, id 47052, offset 0, flags [DF], proto TCP (6), length 644)
    10.20.20.2.57800 > 10.20.15.4.afs3-callback: Flags [P.], cksum 0x6808 (correct), seq 205702705:205703297, ack 212309465, win 46,
options [nop,nop,TS val 1886549971 ecr 1886526901], length 592
19:14:55.369080 IP (tos 0x0, ttl 64, id 64828, offset 0, flags [DF], proto TCP (6), length 52)
    10.20.17.4.afs3-callback > 10.20.15.5.39393: Flags [.], cksum 0xfd31 (correct), seq 0, ack 689, win 125, options [nop,nop,TS val
1886545996 ecr 1886536998], length 0
19:14:55.369909 IP (tos 0x0, ttl 64, id 60390, offset 0, flags [DF], proto TCP (6), length 52)
```

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、
#Cassandra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 [ "$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=${worker_node_ids[0]}"
">$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[0]} > $final_result_data_directory/$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[0]}
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)/analyzeTraffic $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測定する。
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、

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

Type B の場合は、時間 t の時の通信量を $B(t)$ とおくと、

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

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、

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

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 で発生す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、 $[\text{通信量}] = 229\text{Mbps}$ 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、

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

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。

TYPE A: 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。

$$(\text{ノード台数}) = n \quad (6.5)$$

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

さらに、(一台あたりの通信量/s)を分解すると、

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

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、

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

一方、

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

よって、

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

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

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！)

Figure 1: 実マシン数別の通信量の時間変化

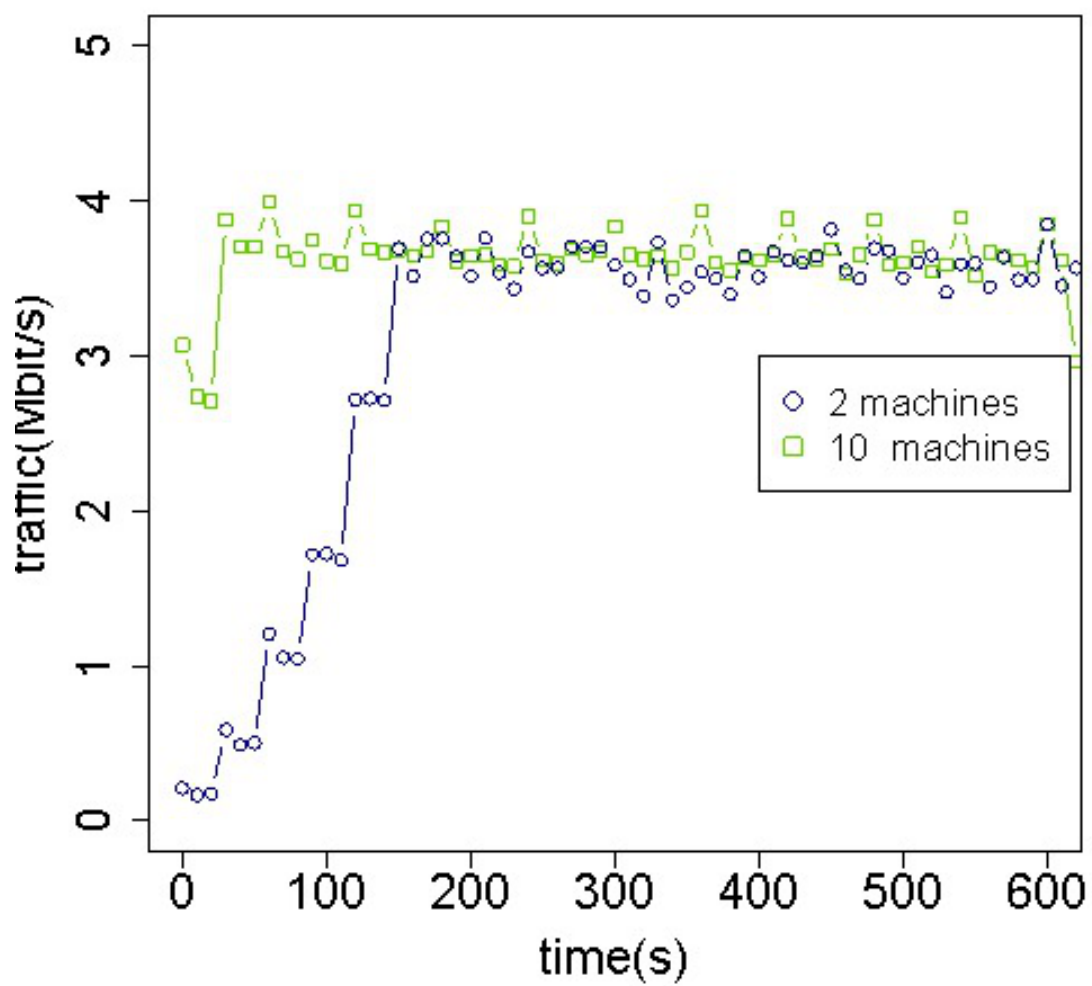


図 6.1: infer-traffic の実行結果

Figure 1: ノード台数別通信量の時間変化

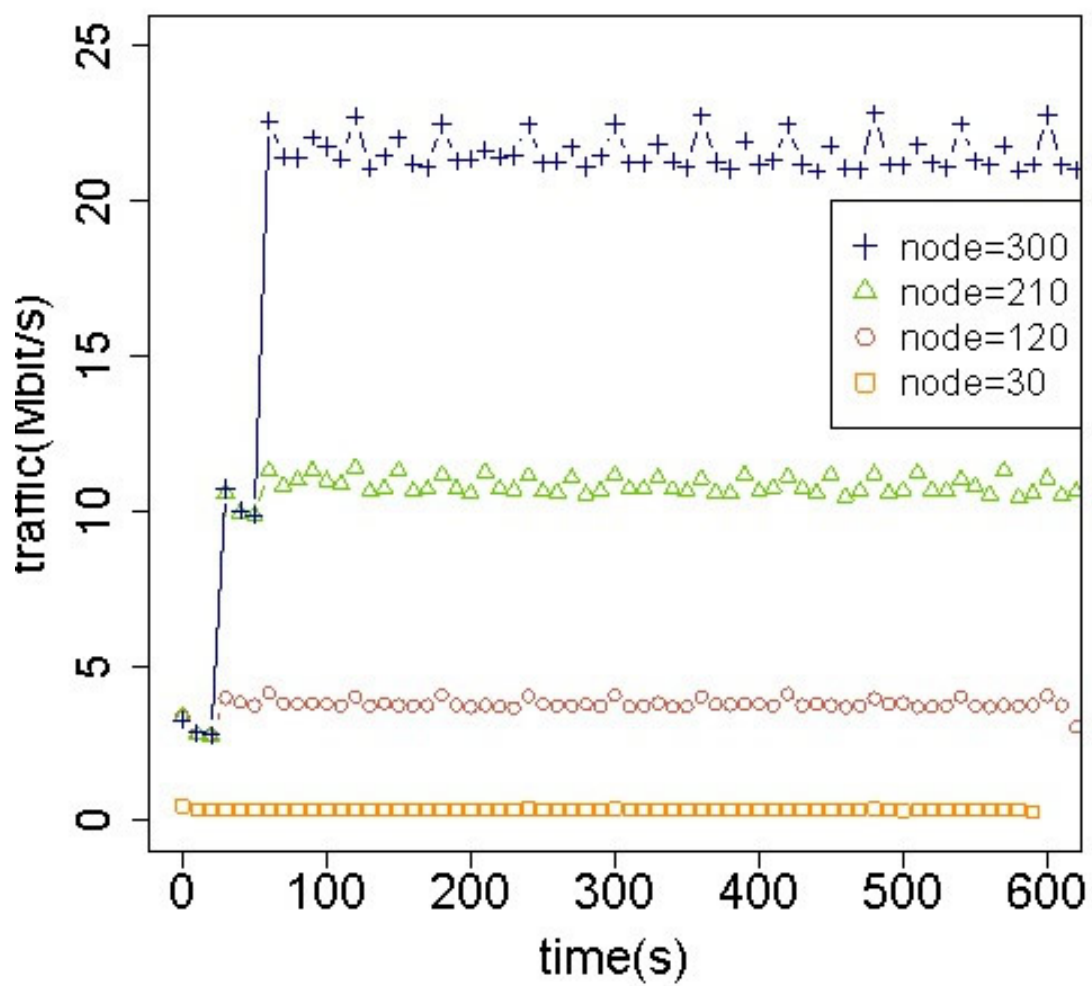


図 6.2: infer-traffic の実行結果

Figure2: ノード台数と通信量の変化

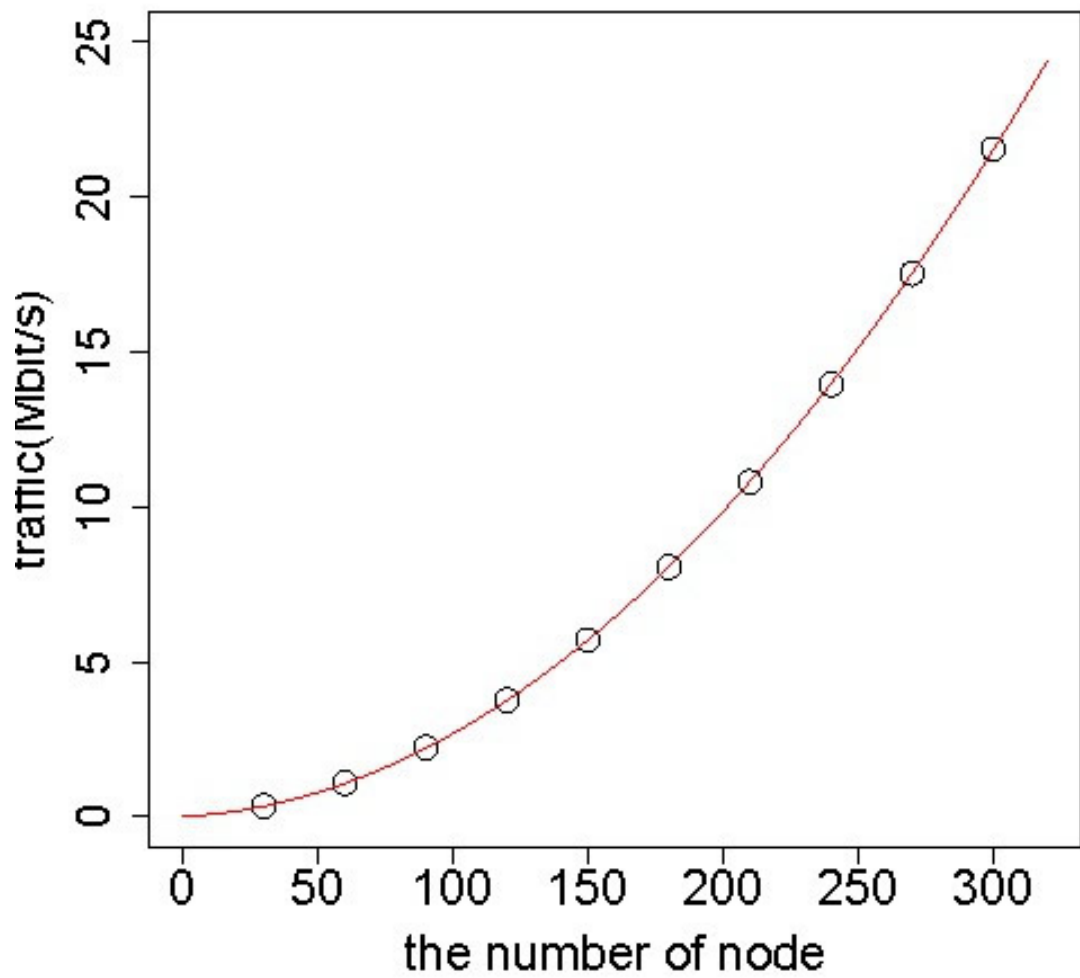


図 6.3: infer-traffic の実行結果

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

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