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 研究背景とし分散システム (クラウド) におけるメンバーシップ管理アルゴリズムとその問題点を指摘する。

第 3 章では、Cassandra ノードで発生する通信負荷の測定手法を説明す

る。
第 4 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 5 章では、関連研究を紹介する。
第 6 章では、結論を述べる。

第2章　研究背景

2.1　大規模分散システムとは

2.2　集中型 (?)

2.2.1　集中型の特徴

2.2.2　集中型の代表的なプロトコル

プロトコル A

プロトコル B

プロトコル C

2.2.3　実装されているソフトウェア

ソフトウェア A

ソフトウェア B

2.3　非集中型

2.3.1　非集中型の特徴

2.3.2　非集中型の代表的なプロトコル

プロトコル A

プロトコル B

プロトコル C

2.3.3　実装されているソフトウェア

Amazon Dynamo

Cassandra

2.4　通信量の観点から評価が知られていない

...(私が調べることになった目的について述べる。)

2.5　Javassist

`Javassist [3, 1]` は...
`Javassist` は `[2]` で配布されている。

2.6　Addistant

第3章　測定手法

3.1　Cassandraの概要

・メンバーシップ周り？
・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。

3.2　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。

3.2.1　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。

3.2.2　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 l 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 130 -> 100 に落とした。

3.3　実験シナリオ

実験では、マスターとなるマシンを l 台とワーカーとなるマシンを１０台を用意した。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。

3.3.1　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
- 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

3.3.2　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。

- 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 マシン同士の見分けがつき、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。
- 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、sudo tcpdump -i dst net 10.20.0.0/16 .....
というコマンドを実行した。

- ユーザーのリソース制約を外す
  特に、Cassanddrad 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。今回プロセス数、メモリー数の制約を外した。

- 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、この測定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。

3.4　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。だが、計測した総通信量から Cassandra Node で発生する総通信量を計測できることを示す。
ここで、一つ仮定を立てる。
・任意の Cassandra Node 同士の通信量は平均すると同じ！(言葉をおきかえる!)
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 × 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) * 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 * m 台の場合は、
TT = T * (n * m) / ((n-1) * m) = T * n / (n-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。
(図を使おう)

第4章　実験・評価

4.1　予備実験

本実験に入る前に、予備実験を行った。

4.1.1　予備実験 (1)

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。

図??が,TypeA の場合の取得した通信量の合計の時間変化、図?? が TypeB の場合である。
X 軸は時間軸を示していて、Y 軸は通信量 (単位は M bit) を示している。この値から総通信量を推定する。
Type A の場合は、時間 t の時の通信量を A(t) とおくと、
推定される通信量 $(t) = A(t) * 10/9$
Type B の場合は、時間 t の時の通信量を B(t) とおくと、
推定される通信量 $(t) = B(t) * 2/1$
上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

4.1.2　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
くらいか？グラフを一枚書いて終了！

4.2　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、[通信量 (bit)] = $224.6 \times n^2$ + 4314.8 × n である。

4.3　評価

4.3.1　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。

4.3.2　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 ノードあたりの通信量] $= (224.6 \times n^2 + 4314.8 \times n)/n = 224.6 \times n + 4314.8$
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。(グラフ)

4.3.3　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

4.3.4　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。

4.3.5　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。

4.3.6　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。

- (ノード台数) = n
- (総通信量/s) = (1 台あたりの通信量/s) * (ノード台数)

さらに、(一台あたりの通信量/s) を分解すると、

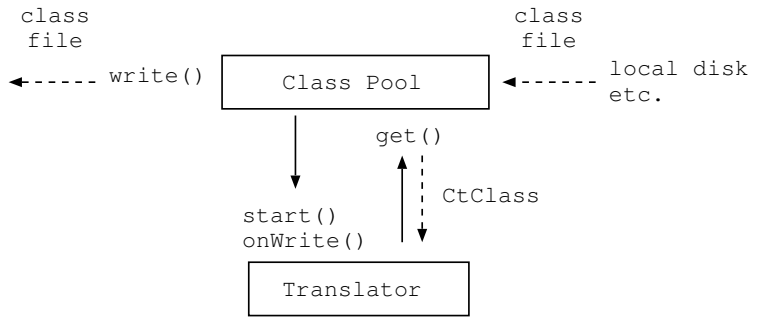

図 4.1: Javassist の処理の流れ

- （1 台あたりの通信量）=（1 秒あたりの gossip 通信する回数）*（一回あたりの gossip 通信にかかる通信量）

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 秒あたりの gossip 通信する回数）= 1,

一方、

- （一回あたりの gossip 通信にかかる通信量）= Order(n)

よって、

- （1 台あたりの通信量）= O(n)
- （総通信量）= Order(n^2)

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 4.1 は...

第5章　関連研究

分散システムのメンバーシップ管理について通信量の観点から調べたという論文をサーチし載せる

5.0.7　論文 A

表 5.1 は...

表 5.1: Addistant 2 の実行時間 (秒)

	時間
X Window System	15.0
Addistant 1	3.0
Addistant 2	2.0

第6章　結論

6.1　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。

6.2　今後の課題

6.2.1　.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。

6.2.2　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