スケールアウト可能なログ検索エンジンの実現と評価

阿部 博[1,2,3,a)]　篠田 陽一[2,b)]

概要：ネットワークのトラブルシューティングやセキュリティインシデントに対応するため，ネットワーク管理者はトラブルの原因を特定するためにサーバやネットワーク，セキュリティ機器から出力されるログを蓄積し，検索をすることがある．大規模なネットワークでは，出力されるログの量も多く蓄積・検索システムの規模も巨大化する．大量に出力される機器のログを高速に蓄積し，高速に検索する先行研究として Hayabusa を実装した．本研究では，Hayabusa の検索性能をスケールアウト可能なシンプルな分散システムの設計を行い評価した．結果として，スタンドアロン環境で動作する Hayabusa の約 78 倍高速な分散処理システムを実装し，144 億レコードの syslog メッセージを約 6 秒でフルスキャンし全文検索可能なスケールアウトするシステムを実現した．

キーワード：syslog，全文検索，GNU Parallel, 分散処理，SQLite3，ZeroMQ

Research of the scalable syslog search engine and evaluation

Hiroshi Abe[1,2,3,a)]　Yoichi Shinoda[2,b)]

Abstract: In this study, we introduce a simple and high-speed search engine for large-scale system logs, called Hayabusa. Network administrators can use Hayabusa to accumulate and store log information at high speed and to search logs quickly to solve network troubles. We also improve Hayabusa to the simple distributed system. Distributed Hayabusa can scale out its processing speed increasing processing hosts and worker processes. Compare with the standalone Hayabusa environment, the proposed distributed Hayabusa was approximately 78 times faster. Finally, distributed Hayabusa required only 6 seconds to search a keyword from 14.4 billion records.

Keywords: syslog, full text search, GNU Parallel, distributed system, SQLite3, ZeroMQ

1. はじめに

1.1 背景と目的

安定したネットワーク運用やトラブルシュートを行う為に，ネットワーク管理者はネットワーク機器から出力されるログを収集し統計情報として表示を行ったり，トラブルの原因であるログを検索する方法が実運用では多く用いられる．また，セキュリティインシデントに対応するために，トラブルの対応方法と同様にログからどのようなインシデントが発生したかを探ることが多い．大規模なネットワークでは，多くのネットワークやサーバ，セキュリティ機器から日々大量の通信を記録したログが出力され，ネットワーク管理者はそれら大量のログをストレージシステムに蓄積し高速にログを検索するシステムを管理する．

大規模なログ検索システムと蓄積システムを扱う為に，クラスタリングシステムや専用の管理ソフトウェアを用いることになり，本来はログの解析に時間を費やしたいネットワーク管理者が検索・蓄積システムの管理に時間を割かれることも多い．「ログの蓄積」と「ログの検索」がシンプルに動作し，かつ複雑なクラスタリングシステムを用いずに実現されるならば，ネットワーク管理者が検索・蓄積システムの管理に時間を割かれることはなくなり，ネット

1 株式会社 IIJ イノベーションインスティテュート
2 北陸先端科学技術大学院大学
3 情報通信研究機構
a) abe@iij.ad.jp/h-abe@jaist.ac.jp
b) shinoda@jaist.ac.jp

ワークのトラブル対応やセキュリティインシデントの解析に集中することができる．

本論文では，多数のマルチベンダ機器が出力する大量のログを高速に蓄積し，高速に検索可能なシステムの提案を行う．また，システムが扱うログの量が増加した場合にも，システムの検索性能が容易にスケールアウトし，検索速度が飛躍的に向上する分散システムのコンセプトモデルについて提案する．本提案において，データ蓄積の並列化に関してはログ生成プログラムを元に検証を行い，検索の高速化に関しては，多数のマルチベンダ機器が出力する大量のログの実データとして，600 以上のサーバ・ネットワーク・セキュリティ機器から出力された Interop Tokyo 2017 で収集された ShowNet の syslog 実データを元に検索の検証と評価を行う．

1.2 本論文の構成

2 章では関連研究に関する調査と問題点を提示する．さらに本提案の元となる先行研究「Hayabusa」について解説を行う．3 章では提案手法の設計について詳細を示す．4 章では実装方法を示し，5 章では評価について述べる．6 章では得られた結果から考察を行い，7 章でまとめと今後の課題を述べる．

2. 関連研究

MapReduce アルゴリズム [9] や Apache Spark[15] などの Hadoop エコシステム [1] は全文検索やログ解析によく用いられる．巨大は Hadoop クラスタや Spark クラスタはユーザに高速な検索速度を提供し，ストレージ容量や処理速度がスケールアウト可能な作りになっている．しかしながら，運用者が Hadoop クラスタを管理するのはとても難しく，構築でさえ専用ソフトウェアを必要とする．運用者がシンプルにクラスタを運用したいと思っても，規模が大きくなることにより故障率は高まり，故障箇所の特定や安定したクラスタ運用には複雑な知識が要求される．

Hadoop エコシステムで利用される HDFS[13] や，Elasticsearch[2] は分散ストレージとして動作し高可用性を実現している．これら分散ストレージはデータのコピーを保持し，故障時にデータが完全に失われないように動作するが，信頼性向上のためにストレージの処理性能は低下する．

grep や awk などの UNIX コマンドもログの検索や集計に利用される．しかしながらこれらのツールを用いて高速な検索や集計を行うには，熟練した知識と専用のデータ構造を事前に定義し実行しなければならない．またこれらコマンドはシーケンシャルに実行され，複数ホストで処理を分散させることは基本的には想定していない．

Google が開発した Dremel[11] をベースとしたクラウドサービスである BigQuery[7] は高速に動作するデータベースとして用いられる．BigQuery は 120 億レコードを 5 秒で全スキャンすると言われるほどの高い性能を発揮するがバックエンドで動作するサーバが数千台や数万台と言われる規模で運用される．またスキャンをするたびに課金が発生し，繰り返しデータをスキャンする場合にはコストパフォーマンスはよくない．

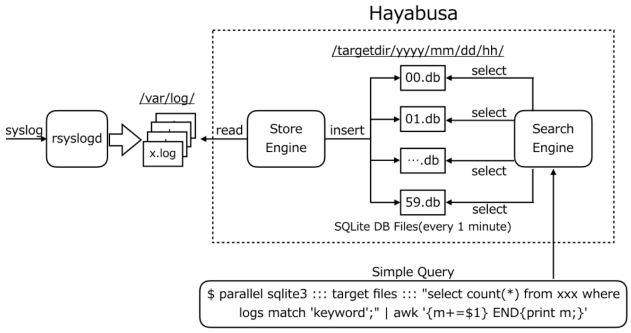

図 1 Hayabusa のアーキテクチャ

2.1 Hayabusa について

Hayabusa[8] は，Interop Tokyo で収集された大量の syslog を高速に検索するためのシステムとして設計された．図 1 に Hayabusa のアーキテクチャを示す．Hayabusa はスタンドアロンサーバで動作し，CPU のマルチコアを有効に使い高速な並列検索処理を実現する．Hayabusa は大きく StoreEngine と SearchEngine の 2 つに分けられる．StoreEngine は cron により 1 分毎に起動され，ターゲットとなるログファイルを開きログメッセージを SQLite3[3] ファイルへと変換する．ログデータは 1 分ごとの SQLite3 ファイルへと分割され，検索時に複数プロセスにより並列処理される．ログが保存されるディレクトリは以下の様に時間を意味する階層として定義される．

```
/targetdir/yyyy/mm/dd/hh/min.db
```

そのためログ検索のための時間情報をデータベース内部に保持することなくディレクトリとのマッチングで行うことができ，時間のクエリ条件を指定することなく時間指定のログが検索可能になる．ログが保存される SQLite3 ファイルは FTS(Full Text Search) と呼ばれる全文検索に特化したテーブルとして作成され高速なログ検索を実現する．

SearchEngine は，並列検索性能を向上させるために分単位に細分化された FTS フォーマットで定義された SQLite3 ファイルへアクセスを行う．各 SQLite3 ファイルへは GNU Parallel[14] を用いて並列に SQL 検索クエリが実行され，結果は UNIX パイプラインを経由して awk コマンドや count コマンドを用いて集計される．

Hayabusa はスタンドアロン環境で動作するが，小規模な Apache Spark のクラスタよりも全文検索性能が高い．先行研究では，Hayabusa は 3 台の Apache Spark クラスタ

```
$ time parallel  --controlmaster -S host1,
    host2,host3 sqlite3 ::: ...
```

図 2　GNU Parallel のリモート実行

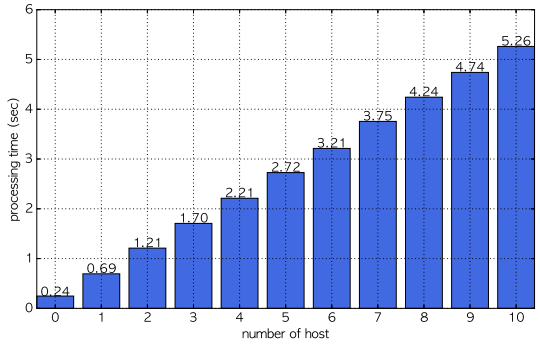

図 3　GNU Parallel の Remote exec 処理時間

より 27 倍早い検索性能を示した．しかしながらスタンドアロン環境にはハードウェア限界が存在し，規模が拡大した他の分散処理クラスタにいつかは性能が抜かれてしまう可能性が高い．そこで本提案では，Hayabusa の限界であるスタンドアロン環境という制約を取り払い，複数ホストで Hayabusa の分散処理環境を構築し，検索性能がスケールアウトするアーキテクチャの実現を目指す．

2.2　GNU Parallel のリモート実行

Hayabusa で用いる GNU Parallel には外部ホストでコマンドを実行するために「remote execution」という機能が存在する．GNU Parallel を用い外部ホストでリモート実行を行うコマンドを図 2 で示す．「-S」オプションの引数としてリモート実行を行うホスト名を指定する．「-S」オプションの実態は ssh コマンドの実行であり，ssh コマンドを用いてリモートホスト側で指定したコマンドを実行する．「–controlmaster」オプションは 1 本の ssh のコネクションで複数の ssh セッションを束ねる．このオプションを指定しない場合には，ssh コネクションが都度作成されリモート実行処理は遅くなる．

図 3 に GNU Parallel をリモート実行した結果を示す．ここでは，10 万レコードが記録された 10 ファイルの SQLite3 データベースファイルを全ホストにコピーし全文検索を行った．各ホストでは 1 台のホストで実行される処理が分散実行される．今回の実験では 10 台のホストで処理を行うため 1 台あたりのホストの処理量はスタンドアロン環境と比較して 1/10 となる．GNU Parallel をリモート実行した場合に，ホスト数が増加するに従い処理速度が遅くなる結果が得られた．図では host 番号 0 がスタンドアロンでの実行を意味し，「-S」で指定するホスト数を 1 台から 10 台まで増加させた．スタンドアロン環境では 0.24 秒で終わる処理が，ホスト 10 台の場合には 5.26 秒かかり約 22 倍処理が遅くなる．処理速度が遅くなる理由は以下が考えられる．

- ssh のオーバヘッド
- ssh で用いられる暗号化のオーバヘッド
- その他問題 (スケジューラ，fork のオーバヘッドなど)

ssh はネットワークを経由して外部ホストと通信を行う．外部ホストとの通信はスタンドアロン環境と比較して通信オーバヘッドが大きくかかる．さらにそこに ssh の暗号化やその他処理のオーバヘッドが上乗せされる．GNU Parallel のリモート実行では対象ホストに対してリクエストを全て ssh プロセスとして起動し実行するため，リクエストの処理ターゲット数が多い場合にはオーバヘッドがさらに大きくなる．

Hayabusa の検索クエリは対象ファイルの数だけプロセスを生成するため，全ての生成されたプロセスが ssh 経由で実行される．これによりホストが増加するごとに対象ファイル分の ssh プロセスが生成され処理に時間がかかる．結果として，GNU Parallel のリモート実行を用いたとしても Hayabusa のフレームワークでは検索処理は遅くなりホストを増加させても処理はスケールアウトしない．

3.　提案手法

3.1　設計概要

本研究では，スタンドアロンで動作する Hayabusa を分散処理システムとして再定義し，検索処理性能をスケールアウトさせる．Hayabusa で利用する GNU Parallel のリモート実行機能では検索処理はスケールアウトしないため，Hayabusa のスタンドアロン処理性能を生かすべく，処理リクエストをクライアントから高速な RPC（Remote Procedure Call）として処理対象ホストへと送り込む．GNU Parallel を用いた並列検索を処理ホスト内で実行し，結果を RPC を用いてクライアントへと返し集計を行う．これにより高速なスタンドアロン環境での検索と RPC による高速なリクエスト/レスポンスを実現する．分散ホスト環境におけるスケールアウトする検索機能を実現するために，データの並列蓄積と並列検索に分けて設計を行う．

3.2　並列蓄積

検索処理をスケールアウトさせるために，どの処理ホストに処理リクエストが届いても検索可能なように全ての処理ホストに同一のデータを保持させる．これは全ての処理ホストへと syslog データを複製して配送することを意味する．これにより，どの処理ホストへと処理リクエストが渡ろうとも同じ結果が返ることが保証される．また syslog が複製されることにより，データ自身の保全性が高まり処理ホストが故障しデータが消失したとしても他の処理ホストにデータが残り対故障性が向上する．

syslog を複製する処理を 1 プロセスで行う場合には CPU 負荷が高まり syslog データが消失する可能性がある．そこで，syslog の複製処理がボトルネックとならない様に，複製処理をマルチプロセス化して CPU コア数に応じてコアスケールする設計とする．

3.3 分散検索

分散している処理ホストへの検索リクエストの実現には RPC を用いる．本提案では，クライアントからのリクエストを受け取った処理ホストは，スタンドアロンの Hayabusa と同等に複数データベースファイルに検索クエリを発行することを前提する．SQLite3 のコマンドを検索リクエストとすることも可能ではあるが，本提案ではスタンドアロンの Hayabusa の性能を活かしつつ，シンプルに分散処理を実現するために，スタンドアロン環境で処理されるものと同等の GNU Parallel のコマンドをリクエストのパラメータとして受け渡す．分散検索では複数処理ホストへと処理リクエストが発行されることから，クライアントが処理リクエストをキューイングし，処理ホストで実行される Worker がキューイングされた処理リクエストを取得して結果を返す Producer/Consumer モデルを選択する．これにより，各処理ホストで動作する Worker は処理リクエストを取得したのちに，検索処理を実行できる．本提案では，全ての処理ホストにデータが複製されていることから，どの処理ホストへと処理リクエストが渡ったとしても同様の処理結果が返される．

また，クライアントからの処理リクエストが特定の処理ホストに集中しない様にロードバランシングを行い，各処理ホストへと均等にリクエストを配布可能とする．

3.3.1 適切な処理データサイズ

スタンドアロン環境で高速に動作する Hayabusa の性能を最大限に引き出すために，スタンドアロン環境で実行される処理をある程度のサイズにグルーピングし RPC のオーバヘッドを抑え，結果をまとめてクライアントへと返す方法が高速に動作すると推測できる．1 データベースファイル毎に対する検索クエリーを 1 リクエストとして処理することは可能であるが，対象ファイルの数だけ RPC が実行され，ネットワークの処理オーバーヘッドが加算され処理にかかる合計時間が長くなる．そこで，ある程度のサイズに処理をまとめて各処理ホストへと RPC をリクエストする方法をとる．

ここでは処理するデータベースファイル群をグループ化して処理時間を測定する．対象となるデータのグループは 1，2，3，6，12，24 時間とし，測定した結果を表 **1** とする．また，測定した計算機環境を表 **2** に示す．ShowNet の syslog 実データから抜き出した 10 万レコードが保存されたデータベースファイルを処理対象ファイル数複製して計測を行った．時間を測定したコマンドは図 4 であり，各処理時間は 10 回試行した平均値となる．

表 **1** 適切なデータグループサイズの推定

時間	レコード数	対象ファイル数	処理時間
1 時間	600 万	60	0.576 秒
2 時間	1200 万	120	0.788 秒
3 時間	1800 万	180	0.977 秒
6 時間	3600 万	360	1.501 秒
12 時間	7200 万	720	2.632 秒
24 時間	1 億 4400 万	1440	4.831 秒

表 **2** 実験環境

CPU	Intel Xeon CPU E5-2650 (2.0GHz/8 core) x 2
メモリ	128GB
ディスク	SSD 200GB(SATA/MLC) x 4
NIC	Intel X520v2 DualPorts 10G
OS	Ubuntu 16.04.3 LTS (Xenial Xerus)

```
$ time parallel sqlite3 ::: /targetdir :::
    "select count(*) from syslog where
   logs match 'noc';"
```

図 **4** 適切なグループサイズの試行

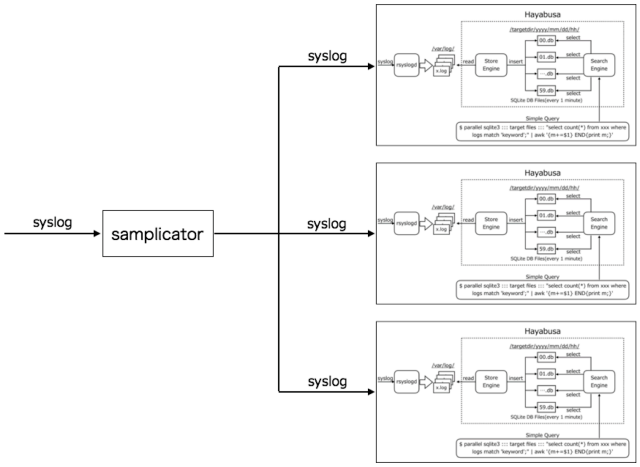

図 **5** UDP Samplicator による syslog の複製

4. 実装

4.1 並列蓄積

4.1.1 データの複製

syslog を全ての処理ノードへと複製するために本研究ではオープンソースソフトウェアである，UDP Samplicator [5] を利用した．UDP Samplicator は，受信した UDP パケットを送信元アドレスを変更せずに，指定した対象ホストへと転送する．これにより転送先のホストは，あたかも自身が直に送信元からデータを受信したかの様に UDP パケットを受信することができる．

UDP Samplicator を利用することで，複数の処理ホストへと syslog パケットを転送する．図 5 に示す様に，全ての処理ホストは複製された同じ syslog パケットを受信する．

```
/* add SO_REUSEPORT */
int optval = 1;
if (setsockopt (ctx->fsockfd, SOL_SOCKET,
    SO_REUSEPORT, &optval, sizeof(optval))
    == -1)
  {fprintf ("setsockopt(SO_REUSEPORT)
      error\n";}
bind (ctx->fsockfd, (struct sockaddr*)&
    local_address, sizeof local_address);
```

図 6　SO_REUSEPORT を使ったマルチプロセス化

4.2　マルチプロセス化による負荷軽減

UDP Samplicator は 1 プロセスで UDP の転送処理を行う．しかしながら大量の syslog を受信した場合には，プロセスの負荷が上昇し CPU のコア利用率が 100%になってしまう場合があり，パケット転送処理が追いつかなくなりデータが破棄される可能性がある．そこで本提案では UDP Samplicator へとパッチを当て，マルチプロセスとして動作する様にソースコードに修正を行った．

図 6 に追加したコードを示す．socket のオプションに「SO_REUSEPORT」を用いることで，複数プロセスで同じ待ち受けポートが利用可能となる．本提案の場合には，syslog 受信ポート「UDP 514」が複数プロセスで共有されることとなり，UDP 514 ポートへと届いたパケットは，複数の UDP Samplicator プロセスに自動的にロードバランスされる．これにより大量に syslog を受信した時に，1 CPU コアではボトルネックになりがちな syslog パケットの複製と転送処理を CPU コアスケールすることができる．

4.3　分散検索

検索処理リクエストはキューイングされ，Producer/-Consumer モデルで処理される．Consumer にあたる処理ホストはキューイングされた処理リクエストを取得するが，この時に Producer は各ホストに均一に処理リクエストが行き渡る様にロードバランスを行う．

Producer/Consumer モデルは多くのソフトウェアで実装可能であるが，本研究では処理が高速に実行可能で，ライブラリとしてクライアントと Worker プロセスを実装可能な ZeroMQ[10] を用いた．ZeroMQ は高速に動作する分散メッセージキューとして利用され，「Request/Response」「Publish/Subscribe」「Push/Pull」などたくさんのメッセージングパターンを容易に実装することができる．本提案では，「Push/Pull」パターンを用いて Producer/Consumer モデルを実装する．

4.3.1　ZeroMQ の Push/Pull

図 7 で示すように ZeroMQ の Push/Pull パターンは以下の順序で処理が行われる．

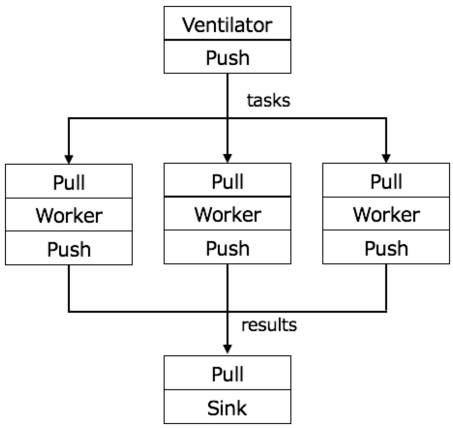

図 7　Push/Pull パターン

```
import sys
import zmq

context = zmq.Context()
sender = context.socket(zmq.PUSH)
sender.bind("tcp://*:5557")
receiver = context.socket(zmq.PULL)
receiver.bind("tcp://*:5558")

cmd = 'parallel target-data "select
    sentence"'

REQUEST_SIZE=100
for i in range(REQUEST_SIZE):
    sender.send(cmd.encode('utf-8'))

while True:
    message = receiver.recv()
    print(message.decode('utf-8'))
```

図 8　クライアントソースコード

1) Ventilator がリクエストをキューイング (Push)
2) Worker が Ventilator からリクエストを取得 (Pull)
3) Worker が結果を Sink へと送り (Push)，Sink で取得した結果 (Pull) を集計

本提案でのクライアントは，Ventilator と Sink の 2 つの役割を持つ実装とする．これによりリクエストの発行からキューイング，結果の取得と集計を 1 プロセスで行うことができる．図 8 にクライアントのソースコードを示す．

図 9 で示す様にクライアントは，処理ホストへ投入する処理リクエストをキューイングし，各ホストで動作する Worker が処理を Pull し実行した後に結果をクライアントへと送信し，クライアントが結果の集約を行う．クライアントは TCP の 5557 番ポートを用い Worker からの接続を待ち受け，処理リクエストをキューに Push する．処理結

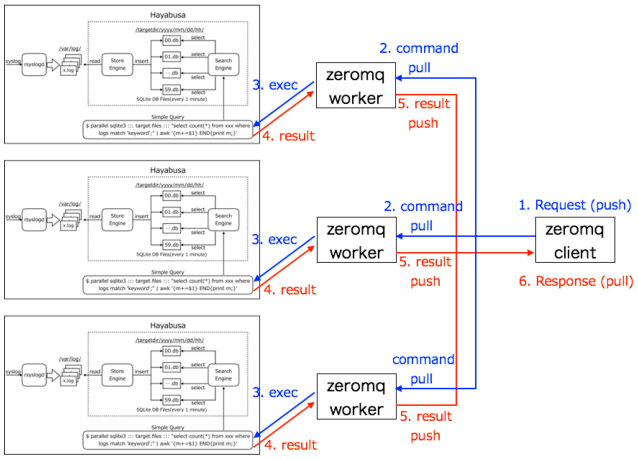

図 9　Hayabusa で用いる ZeroMQ の Push/Pull 実装

```
1  import zmq
2  import subprocess
3
4  context = zmq.Context()
5  receiver = context.socket(zmq.PULL)
6  receiver.connect("tcp://client:5557")
7  sender = context.socket(zmq.PUSH)
8  sender.connect("tcp://client:5558")
9
10 while True:
11     recv = receiver.recv()
12     cmd = recv.decode('utf-8')
13     res = subprocess.check_output(cmd)
14     sender.send(res)
```

図 10　Worker ソースコード

果は，TCP の 5558 番ポートで受け付け集計を行う．

次に，Worker のソースコードを図 10 に示す．Worker はクライアントへの接続をブロックし，クライアントが動作したタイミングでクライアントがオープンした TCP 5557 番へ処理リクエストを Pull するために接続する．その後 Pull した処理リクエストを取得し，リクエストに含まれるコマンドを実行した後に結果をクライアントの TCP 5558 番ポートへと Push する．

5. 評価

本研究では，スケールアウト試験を行うために北陸 StarBED 技術センター [6] にて提供されるサーバ群を用いた．StarBED[12] はベアメタルサーバを備える国内最大のテストベッドであり，最先端のサーバ機器を利用することができる．スケールアウトの試験では，処理ホストを 1 台から 10 台へと増やし蓄積と検索の実験を行う．処理ホストの他に，syslog を生成する役割や分散クエリのリクエスト行うクライアントホストを 1 台用意し，さらに UDP

```
$ loggen --dgram --size 140 --rate 100000
    --interval 60 host1 514 -n 0 -P
```

図 11　loggen コマンド例

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
56142 root      20   0    4368    676    580 R  49.5  0.0   1:10.01 samplicate
56136 root      20   0    4368    604    508 R  49.2  0.0   1:36.30 samplicate
56139 root      20   0    4368    544    452 R  48.2  0.0   1:19.47 samplicate
```

図 12　UDP Samplicator のマルチコア対応

Samplicator 用サーバとしてもう 1 台ホストを準備する．実験ホストのスペックは　表 2 と同等である．

5.1 並列蓄積

蓄積の実験を行うために構成を以下の様に定義した．

- loggen を実行する syslog 送信元ホスト (1 台)
- UDP Samplicator による syslog 複製ホスト (1 台)
- rsyslogd により syslog を受信し，StoreEngine により SQLite3 形式で出力するホスト (10 台)

syslog の生成方法として，本実験では syslog-ng[4] の付属プログラムである loggen コマンドを利用した．loggen コマンドは，指定したレートで送信先ホストへと syslog データを出力することができる．図 11 に示す様に，loggen コマンドは指定されたサイズの syslog データを指定レートで指定した時間送信することができる．例では「サイズ 140 の syslog を UDP で 1 分間，秒間約 10 万メッセージのペースで送信する」という意味になる．

5.1.1 UDP Samplicator の複数プロセス化

「SO_REUSEPORT」を使った UDP Samplicator での負荷分散は，図 12 で確認できる．これは，loggen コマンドを 3 プロセスで実行し並列してログを出力した場合に，3 つの UDP Samplicator プロセスの CPU 負荷が均一にバランスしていることを示す．さらに UDP Samplicator の負荷が高まった場合には，UDP Samplicator のプロセスを増やすことにより，負荷を下げることができ CPU コアの数だけ複製・転送処理をスケールアウトすることができる．

5.1.2 各ホストでの並列蓄積

各処理ホストは rsyslogd で syslog を受信しファイルへと書き込み，StoreEngine によって 1 分毎に SQLite3 の FTS 形式でログをデータベースファイルへと書き込む．このログ蓄積動作はスタンドアロンの Hayabusa と同等であり，各処理ホストは NTP(Network Time Protocol) により時間同期されているので蓄積するデータにずれは発生しない．

5.2 分散検索

2017 年の ShowNet の syslog 受信サイズは，実データを解析したところ会期期間 (6 月 7 日から 9 日の 3 日間) に入ってから 1 分あたり平均約 5 万件の受信量であった．3.3.1 章で行った適切なデータグループサイズの検証では，

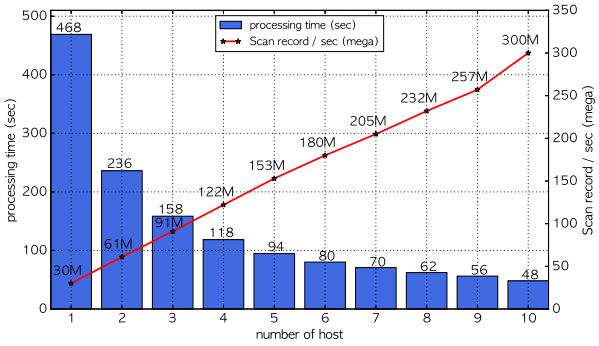

図 13　検索ホストのスケールアウト性能

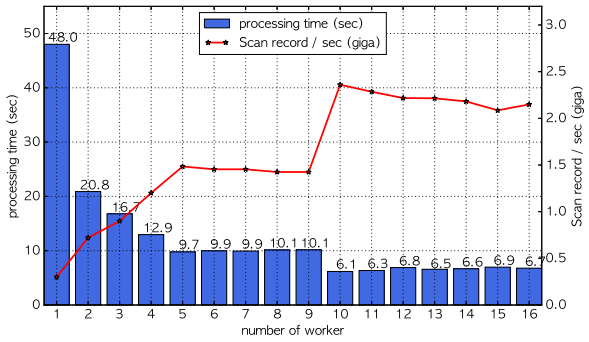

図 14　Worker プロセスのスケールアウト性能

1 データベースファイルに 10 万件の syslog データを保持していた．10 万件は ShowNet の syslog 受信サイズの約 2 倍だが，来年度はさらなる syslog 受信量の増加が見込まれることと，検索性能を測る上で 10 万件のデータはキリが良いことから 3.3.1 章で検証で利用したデータを用い，分散環境でのスケールアウト検索の検証を行なった．

5.2.1　処理ホストのスケールアウト

スケールアウト性能を調べるため，処理ホストが増加した場合に処理時間が短縮するか試験を行った．処理ホストは 1 台から 10 台の範囲で増加し，クライアントは 1 日分のデータに対して繰り返し 100 回リクエストを実行する．100 回分のリクエスト対象のレコードサイズは，144 億レコードとなる．処理結果を図 13 に示す．

ホスト 1 台時の検索処理時間は約 468 秒だが，ホストの台数を増やすに従いホストの数で割った時間に近づき，10 台のホストでかかる処理時間は約 48 秒になる．また，秒間に処理可能なレコードのスキャン数はホスト 1 台の場合には 3000 万件であるが，ホスト数が 10 台の場合には 10 倍である 3 億件まで増加する．これはホストを増加させた場合に検索処理が意図した通りにスケールアウトしていると言える．各処理時間は 10 回試行した平均値となる．

5.2.2　Worker プロセスのスケールアウト

次に 10 台のホストで処理を実行する場合に Worker の数を 1 から 16 の間で増加させた．16 という数字の根拠は CPU の物理コア数であり，物理コアの数だけスケールアウト可能かを確認した．処理結果を図 14 に示す．

各処理ホストの Worker 数が 5 プロセスまでは処理性能が向上するが，5 から 9 プロセスで一度処理時間が横ばいとなる．10 プロセスから再度処理性能が向上するが，11 プロセス以上は 10 プロセス時より検索性能が劣化する．1 Worker 時の処理時間は約 48 秒だったものが，10 Worker 時には約 6 秒となり約 8 倍処理速度が高速化し，Worker の数を増やすことによる処理のスケールアウトも確認できる．1 秒間にスキャン可能なレコード数も，Worker 数が 10 の場合に最大約 23 億件となる．こちらも各処理時間は 10 回試行した平均値となる．

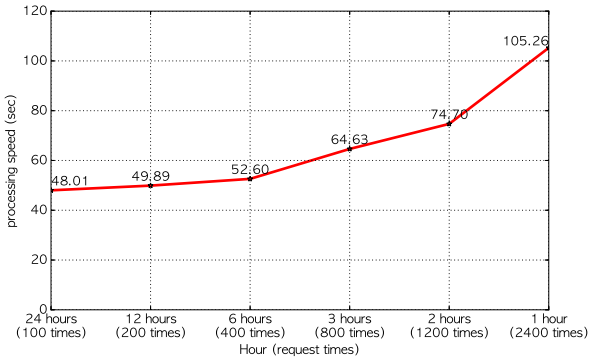

図 15　データのグループサイズとリクエスト回数

5.3　グループサイズとリクエスト回数評価

3.3.1 章で各データのグループサイズを計測したが，1 日分のデータ（1 億 4400 万件）に対し処理リクエストを 10 台の処理ホスト環境に投入し測定した時間を図 15 に示す．全て同じ量のデータに対するアクセスだが，リクエスト回数が増加すると処理に時間がかかる．これは RPC の通信オーバヘッドによるもので，リクエストを可能な限りまとめて投入するべきであり，1 日分のデータは 1 台のホストでも約 4.8 秒で処理可能なことから，細切れにリクエストせず 1 日単位のリクエストで十分高速に動作する．

6.　考察

6.1　検索のスケールアウト

5.2.1 章と 5.2.2 章で示した結果から，1 台の処理ホストとの検索時間と比較した時に，ホスト数の増加実験で約 10 倍処理時間が高速化した．また，Worker の増加実験では最大約 8 倍の性能増加が見込まれた．ホスト数と Worker 数の両方のスケールアウトを組み合わせた結果，1 台の処理ホストで約 468 秒かかっていた検索時間が約 6 秒まで短縮し，約 78 倍高速化した．対象となるレコード数は 144 億レコードであり，144 億レコードを 6 秒でフルスキャンできたということは，Google の BigQuery に匹敵するデータのフルスキャン速度が実現できたことを意味する．10 台の処理ホストでこれだけの高速なスキャンを実現できると

いうことは，コスト的に考えてもリーズナブルで高性能な分散処理システムが実現できたと言える．

6.2 蓄積の並列化

本研究では検索性能を向上させるために分散クエリに対応するため，各処理ホストに同一の syslog データを複製する手法を用いた．これは，本質的には重複するデータを大量に複製する行為であり，データの量が増加すればするほどネットワーク帯域と保持するデータに無駄が発生することを意味する．Hadoop の HDFS のようにレプリケーション数を設定し，複数ホストでデータを分散させ保持することはもちろん可能であるが，その場合にはデータの管理をメタデータ管理機構で行い，データアクセスはメタデータ管理機構経由となり，処理性能を低下させる恐れがある．

本研究では，データを複製することによる帯域問題や容量問題が存在するが，機器の故障時には他の分散ファイルシステムのようにデータの再配置を行う必要もなく，シンプルに機器を管理対象から外すことで対応できる．

6.3 シンプルな設計による運用の簡略化

本提案では分散検索を実現するために，データの複製機構の実現と Producer/Cosumer モデルによる検索の分散化を行なった．設計と実装は共にシンプルであり，管理しなければならないプログラムの数も少ない．Hadoop のような分散システムは，多くの複雑なソフトウェアコンポーネントから実現されており，システムトラブルが発生した時には，トラブル原因を把握するだけで重労働となる．極めて少ないコンポーネントで作られる Hayabusa の分散処理機構は，問題が発生した場合にも原因の把握を高速に行うことができ，システム運用管理の負荷を軽減させる．

7. まとめと今後の課題

7.1 まとめ

本論文では Hayabusa の分散処理の設計と実装を行なった．計測した結果，1 台の処理ホストでは約 468 秒かかった検索処理が最大約 6 秒まで短縮した．144 億レコードの syslog データを 6 秒でフルスキャンし，全文検索可能な Hayabusa はログ検索エンジンとして高い性能を発揮する．これはマルチベンダー機器を管理するネットワーク管理者が大量のログを用いて，トラブルシューティグやインシデントレスポンスを行う上でとても使い勝手の良いツールとなり，対応時間を著しく短縮する可能性がある．また，システムをとてもシンプルに設計しているため，システム管理コストが著しく低くなり，ネットワーク管理者は本来注力したい業務へと時間を割くことができる．

7.2 今後の課題

本論文では Hayabusa の分散処理の実現と測定を行い高いパフォーマンスを実現した．しかしながら，他の処理系との比較がまだ行えていないため比較実験を行った上で Hayabusa の優位性を示す必要がある．

また Hayabusa は検索基盤ソフトウェアとして動作するため，その上で動作する具体的なアプリケーションソフトウェアを組み合わせることでさらなる有益なシステムとなりうる．syslog を高速に検索できることから，先行研究であるインベントネットワークにおける syslog を用いた異常検知 [16] と組み合わせることで，高速に動作する異常検知アプリケーションを作成することが可能となる．

謝辞　本研究の一部は，国立研究開発法人科学技術振興機構（JST）の研究成果展開事業「戦略的創造研究推進事業（CREST）」の支援によって行われた．

参考文献

[1] Apache Hadoop. http://hadoop.apache.org/.
[2] Elasticsearch. https://www.elastic.co/products/elasticsearch.
[3] SQLite. https://www.sqlite.org/.
[4] syslog-ng. https://syslog-ng.org/.
[5] UDP Samplicator. https://github.com/sleinen/samplicator.
[6] 北陸 StarBED 技術センター. http://starbed.nict.go.jp/.
[7] An inside look at google bigquery. 2013.
[8] H. Abe, K. Shima, Y. Sekiya, D. Miyamoto, T. Ishihara, and K. Okada. Hayabusa: Simple and fast full-text search engine for massive system log data. In *Proceedings of the 12th International Conference on Future Internet Technologies*, CFI'17, pages 2:1–2:7, New York, NY, USA, 2017. ACM.
[9] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
[10] P. Hintjens. 0mq - the guide, 2011.
[11] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases*, pages 330–339, 2010.
[12] T. Miyachi, K.-i. Chinen, and Y. Shinoda. Starbed and springos: Large-scale general purpose network testbed and supporting software. In *Proceedings of the 1st International Conference on Performance Evaluation Methodolgies and Tools*, valuetools '06, New York, NY, USA, 2006. ACM.
[13] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.
[14] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.
[15] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.
[16] 阿部博 and 敷田幹文. イベントネットワークにおける syslog を用いた異常検知手法の提案と実データを用いた評価. In インターネットと運用技術シンポジウム *2016* 論文集, volume 2016, pages 57–64, dec 2016.