博 士 論 文

大規模ネットワーク環境における
高速なログ解析基盤と異常検知に関する研究

阿部 博

主指導教員 篠田 陽一

北陸先端科学技術大学院大学

情報科学研究科

平成 31 年 3 月

目 次

図 目 次

表 目 次

第1章　序論

1.1　大規模なネットワークを安定的に管理するための問題点

ネットワーク管理者は日々ネットワークの健全性を評価し、トラブルが発生した場合には様々な角度から分析を行いその原因を特定し安定したネットワーク運用を実現している。そのため、ネットワーク管理者は、ネットワークを流れるパケットやログメッセージ、サンプリングされたネットワークのフローデータなど、サーバ機器やネットワーク機器、セキュリティ機器などたくさんのネットワークへと繋がるデバイスから生成される多種に渡るデータを収集する必要がある。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パケットやログ、フローデータをストレージへと蓄積し高速にデータを検索するシステムを管理する。

ネットワークやシステムに障害が発生した場合に、ネットワーク管理者は可能な限りトラブルの原因を早急に調査し解決しなければならない。ネットワークに対してサイバー攻撃が行われた場合にももちろん、管理するシステムやユーザが保持するデバイスに深刻なダメージを与える前に対処を行う必要があり、かつシステムにダメージを受けた場合には早急に原因と影響範囲を特定し迅速な対処が必要となる。しかしながら、それらトラブルの解決やサイバー攻撃に対処する場

合には、ネットワーク管理者は収集した大量の種類のログデータやフローデータ、パケットなどを分析する必要がある。さらに、調査を行っているトラブルの原因を解明するために、関連するデータが見つかるまでにデータを繰り返し検索する作業が発生する。

調査のために必要な検索時間は、データの種類とデータ量によって大きく異なる。トラブルの原因となるデバイスの数およびデータ量は、多種多様のネットワーク機器が開発されネットワークに接続されることにより確実に増加している。ネットワーク上のルータやスイッチ、サーバ群、ファイアウォール、Sandbox、ネットワークアクセラレータなど、機材はさまざまな機能を提供しており、それら機器からのログ量は増加することはあっても減ることはない。さらに、これらの機材で構成された監視対象のネットワーク規模が非常に大きくなる可能性がある。データセンターでは多くの場合、施設は数万台のサーバとネットワークスイッチやルータ、セキュリティ機器で構成されている。仮想化技術の普及により、単一の物理デバイス上にコンテナインスタンスや仮想マシンの統合が行われ、これらの仮想デバイスは、物理デバイスと同様のトラフィックとログを生成する。ネットワーク管理者はこれら仮想デバイスを物理デバイスと同様に無視することはできず、仮想化によりネットワークやサーバのサイズは物理的に縮小するが、出力するデータサイズ的にはほとんど縮小することができないどころか、仮想環境を構築すればするほど、扱わなければいけないデータの種類と量は増加する。したがって、ネットワーク運用管理において、ログデータをスケーラブルかつ横断的に高速に検索できる検索エンジンが必要になり、それを使うことでネットワーク管理者のトラブル解決を迅速に行うことができる可能性がある。

ファイアウォールのようなセキュリティ製品では、Fortigate の FortiAnalyzer[10] や PaloaltoNetworks の Panorama[19] の様にファイアウォールベンダが提供する専用の商用製品が提供されることも多いが、これらは特定のファイアウォールに特化されたレポーティング製品となっており、汎用的なログ管理・分析には利用で

きない。汎用ログを管理するための製品は多数のものが商用化されている [37] が、製品価格やライセンス価格、保守費用含む導入コストは低くはなく、さらにログの保存期間や保存データ量の規模が大きくなった場合にかかるコストは大きくなる。

場合によっては、ネットワーク管理者は、高価な商用製品の代わりにオープンソースソフトウェアを利用した Hadoop[3] や Elasticsearch[9] のようなクラスタで構成されたデータストレージと検索エンジンを使用する。商用製品を利用する場合と同様に、ログの保存期間や保存量が増加した場合や処理速度を向上させる場合には、サーバを追加しクラスタの全体サイズを大きくしていく必要がある。しかしながら、これらのクラスタを構築する分散システムは、ネットワーク管理者が容易に運用管理を行うことができず追加の運用コストをもたらす場合が多い。大規模な Hadoop クラスタを停止せずに運用する手法自体が研究対象として存在する [29]。さらに、クラスタを効率的に運用するためには、多くの最適化構成の施行とパラメータチューニングが必要となる。

ネットワーク管理者が日々対応するトラブルシューティング作業のために高速なログ検索システムを利用する必要があるが、ネットワーク管理者自身が複雑なクラスタを管理することになりクラスタの管理時間が増え、ネットワークを安定的に運用するという本質的な作業の時間が削られる問題が発生する。

1.1.1 マルチベンダ機器で構成される大規模ネットワークでのトラブル対応の難しさ

企業や学校で利用される大規模なネットワークでは、マルチベンダのネットワーク機器を用いてシステムが構築されることが多い [52, 40]。ネットワーク管理者は特定ベンダのエンジニアではないために、複数のベンダ機器が出力するログに対する理解や未知のログに対する対応、または膨大なログ量の解析に関して出力されるログのどのキーワードがエラーやアラートであるのかを判断すること可能で

はあるが、ネットワーク管理者の経験に大きく依存する。ネットワーク管理者が事前に把握している特定のキーワードに基づくエラーハンドリングは可能ではあるが、ログの急増や未知のログに対処することは困難であり、そもそも膨大なログに埋もれて本来発見したい異常を発見できない場合もある。

大規模なネットワークは企業や学術機関以外でも構築されることがある。大規模な展示会やカンファレンス、シンポジウムといったイベントでは、来場者や参加者に対してインターネットへのアクセスがサービスとして提供されることがある。これらのネットワークを総称してイベントネットワークと呼ぶ。イベントネットワークはネットワーク機器のショーケースを兼ねることがあり、その場合にはマルチベンダのネットワーク機器を用いてシステムが構築されることが多い [20, 48, 16]。1 週間から 2 週間という短期の準備期間内でネットワークの構築から運用/撤収を行うため、構築されるネットワークが安定稼働するまでに様々なトラブルが発生する。通常のネットワーク運用とイベントネットワーク運用の違いは、大規模なイベントネットワークでは多くのマルチベンダ機器や世界で初めて投入される機材、ソフトウェアが利用され、通常の運用では知り得ないバクやエラー、未知のログメッセージに遭遇する可能性があり、ネットワーク管理者がログの意味自体をその場で理解することが難しい。マルチベンダの機材を用いることにより、機材の互換性に関するトラブルの把握や解決など運用者の経験に基づく問題解決に依存することが多く、短期間でのトラブル対応の自動化が困難である。

1.1.2 データ量の問題

ネットワーク上のルータやスイッチ、サーバ群、ファイアウォール、Sandbox、ネットワークアクセラレータなど多種多様なデバイスからインターネットへと膨大なデータが流れ込むようになり、それらデータを蓄積し、処理するシステムが多数構築されている。しかしながらデバイスから流れ込む情報は膨大な規模となり、それら多種多様なデバイス自体が大量のデータを出力する装置と化している。

ビッグデータという抽象的なキーワードが先行する中において、堅実なデータ処理基盤を構築するには、ネットワーク・サーバ・セキュリティ機器群が出力する全てのデータを真面目に蓄積し、検索するにはデータ量が膨大になる。

膨大なデータを処理するための多くのアプローチは情報をフィルタすることとなる。具体的には、

1. データを出力するエッジデバイスで出力する情報を取捨選択
2. データ処理システムによる情報の取捨選択
3. データ蓄積システムによるスキーマ定義と適用

といったアプローチで、処理可能な情報量までデータの粒度を落とすことにより効率的な対処を実現する。しかしながら、情報の精度を下げることや情報のフィルタリング・サンプリングは、データが非可逆的な状態となり元データを完全に復元することはできない。トラブル発生時には、欠落のないデータからトラブルの元となる情報を得る必要があり、安定したネットワーク運用を目指す場合には可能な限り全てのデータを保存したい。また、ネットワークを安定に運用するために中長期のデータ分析を行うためには、さらに長期間のデータ蓄積が必要となる。単発のトラブル対応と中長期を視野に入れた安定運用の両方を実現するために、データの保存とデータの検索時間の即応性の両方を実現する必要がある。

1.1.3 時系列データ

スイッチ、サーバ群、ファイアウォール、Sandbox、ネットワークアクセラレータなどのネットワークを構成する機器は、大量のデータを時間軸に沿って出力する。そしてそのように時間軸に沿って収集されたデータは時系列データと呼ばれる。時系列データはその名の通り古い順番より収集され新しいものが最新データとなるように蓄積される。ログデータとしては、一般的に「時刻」と「送信元識

別子」、「メッセージ部」となるが、収集するシステムによっては時間以外のデータは異なる場合がある。例えば統合監視などでSNMPを利用して収集される情報も時系列データとして格納可能であるが、収集される情報は「時間」と「各種監視メトリックの数値や値」となる。収集された時系列データがどのように蓄積され、処理されるかは、そのデータを扱い何を行いたいかによって異なる。

ネットワーク管理者がトラブルやインシデント対応を行う場合には、事象が発生した時刻を起点にして時系列に事象を調査することが多い。トラブル発生時刻の前後の時間を調査対象とし原因究明を進める運用が行われるため、時系列データはネットワーク管理者にとってとても重要なデータ形式と言える。

1.1.4 障害と損失

EMCジャパンの報告によると[1], 2015年の段階で、データ損失やシステムダウンなどの障害により生じた過去1年間の損失額は、国内企業1社あたり約2億1,900万円（1米ドル＝100円で算出）[2]、国内全体で約4兆9,600億円に上ることが明らかにされた。大規模ネットワークを運用するような組織では、システム障害による経済的な損失は大きくなり、システムの障害をいかに早く発見し解消することが損失を抑える方法となる。

しかしながら、IPAの報告によると運用のコストがITシステム関連のコストの中で保守も運用管理の一部とした場合に約80％近くになるとの試算が報告されている[3]。運用を効率的に行うことがシステム運用コストを抑えることとなり、さらに障害発生時には短時間で障害を特定し復旧させることもやはり運用コストを抑えることとなり、ネットワーク管理者は効率的なシステム運用や監視の自動化、トラブル解消の即時性が求められることとなる。

[1] https://japan.emc.com/about/news/press/japan/2015/20150123-1.htm
[2] https://www.emc.com/microsites/emc-global-data-protection-index/images-infographics/infographic-japan.jpg
[3] https://www.ipa.go.jp/files/000045091.pdf

ログを分析することによるシステム障害への即時対応、ならびにログ監視の効率化やログに基づく異常検知自動化による障害の予兆検知が可能になることにより、ログ解析の高速化がシステム障害への損失を減らすことができる可能性がある。

1.2 研究の目的と概要

1.2.1 研究の目的

時系列データを対象とした「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。本研究では、多数のマルチベンダ機器が出力する大量の時系列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。さらに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。

また、多数のマルチベンダ機器が混在する大規模なネットワークにおいて、ネットワーク管理者にとって未知のログが多数出現する過酷な環境下であっても時系列で収集されるログの総量から異常を読み取り、効率的に通知可能なアルゴリズム適用の提案を行う。提案手法では株式市場で用いられるテクニカル分析手法を採用し、正規分布に基づく確率理論からログ総量の突発的な上昇や下降を検知する。本手法を用いることでアラートの通知頻度を減少させ、ネットワーク管理者への現実的な異常発生回数の通知が可能となり、トラブルへの初動対応の高速化が、将来的な運用自動化への一手法として成立する可能性を探る。

1.2.2 研究の概要

本研究では、以下の 4 項目について実装と評価を行った。

1. 高速な時系列ログ検索システム Hayabusa の設計と実装
2. Hayabusa の分散システム化
3. 分散 Hayabusa のさらなる効率化
4. 計算が軽量な異常検知手法の提案

高速な時系列ログ検索システム Hayabusa の設計と実装

大量のデータを処理するためには、大量のデータを受信し、かつ大量のデータ処理を並行して行う必要がある。本研究ではデータの蓄積に特化したエンジンと、データの検索に特化したエンジンの 2 つを大きな部品として実装する。これにより大量のデータを受信しつつ、大量のデータを処理するデータ処理機構を実現しこの実装基盤を「Hayabusa」と名付ける。検索対象のデータは、スキーマとして定義するには複雑であったりそもそもパースできないデータのため、Hayabusa はデータに対して全文検索処理を基本的に行う。

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、ターゲットとなるファイルから syslog メッセージを取り出し SQLite3 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。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から、時間のクエリ条件を指定することなく時間指定のログ検索が可能になる。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、全文検索のためのインデクシングにより高速なログ検索を実現する。

Hayabusa の分散システム化

Hayabusa はスタンドアロン環境で動作するが、小規模な分散処理クラスタよりも全文検索性能が高い。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。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ことを目標とする。本研究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/Consumer モデルによる検索の分散化を行う。

分散 Hayabusa のさらなる効率化

分散 Hayabusa の課題としてあがった、データの複製を行わずに処理可能な仕組みとしてネットワークストレージを用いたストレージの集中管理機構を実装する。これにより、それぞれの処理ノードでデータを保持する必要がなく、障害等で発生する各計算ノードのデータの不整合や欠落を気にすることなくネットワークストレージに注力して運用することが可能となる。また、同様に課題としてあがったエラー発生の把握についてリクエスト処理機構を実現することで対処を行う。これによりクライアントは、投入したリクエストに発生したエラーを把握することができると同時に、処理ノードでの挙動をクライアントが直接把握する必要がなくなり利用効率が向上する。さらに、OSS 化を行うことで、広く一般的に利用可能なようにソフトウェアとして公開をする。

計算が軽量な異常検知手法の提案

本研究では、大規模ネットワークの一例として、ShowNet という大規模でマルチベンダ機材が大量に投入されるイベントネットワークでの異常検知を、syslog の総量を集計してボリンジャーバンドアルゴリズムを用いて解析し検知する手法を実現する。これによりネットワーク管理者の経験と勘に頼る属人性を排除した自動的な異常検知を行い、ネットワーク管理者へ通知するシステムの実現が可能となる。また、指数移動平均をベースとした動的なアラートのレベル分けを追加することで、ネットワーク管理者に対して現実的な異常発生回数を通知する機構を実現する。結果として、精度の高いアラート検出とレベル分けを行うことが可能となり、ネットワーク運用者へのトラブル対応の高速化の可能性を探る。

また、ログの意味解析をあえて行わず本来は意味解析が有効である機械学習や統計分析では実現できない、軽量で高速に計算できる単純なアルゴリズムが有効に働くことを検証する。高速な計算による自動的な異常通知の結果として、トラブルへの初期対応を素早く実行することが可能となり、マルチベンダによって構成された複雑で大規模なネットワーク運用に広く貢献できる可能性を示す。

1.2.3 本研究で用いるデータ

本研究では時系列ログデータとして、Interop Tokyo[15] で構築される ShowNet[20] の syslog データを用いた。ShowNet は、毎年千葉幕張メッセで開催される Interop Tokyo 内で構築される最新のネットワーク機器や技術を集めた相互接続検証とデモンストレーションを行う実験ネットワーク環境である。また、出展社へのインターネットアクセスをサービスとして提供する側面もあり、実験とサービス提供の 2 面性を有したイベントネットワークである。

ShowNet を構成する機材は、世界で初めて展開されるような最新の製品が多く、機材数は数百を超える。ShowNet には実験ネットワークという一面もあり、試作

レベルの機材やソフトウェアが展開される。ShowNet を運用するメンバーは、これらの機材やソフトウェアを組み合わせサービスを提供するネットワークを運用構築する。

ShowNet では、毎年異なる機材を用いてシステムやネットワークを構築するため、構築の自動化を行うことが難しい。安定したネットワークを構築するために運用メンバーの専門性に特化した高度なスキルが要求される。さらに数年先を見越した最先端の技術導入チャレンジを行うため、運用ノウハウが存在しない技術を利用し、発生したトラブルを運用メンバーの経験と勘で対応することが多々発生する。そのため、展示会直前までネットワークが不安定になることがある。また出展社や関係者の不注意により、ネットワークにループが発生するなど、運用メンバーが意図しない異常が発生することも多い。

ShowNet で収集される時系列ログデータは、構築期間と安定運用期間が明示的なイベントネットワークの特性を保持しており、不安定な状態と安定状態のネットワークをログから観測することが可能である。

1.3 本論文の構成

本論文では、第 2 章で、本研究に関連する既存研究について述べる。第 3 章で、高速なログ検索システム Hayabusa の設計と実装について述べる。第 4 章では、第 3 章で述べた Hayabusa を分散システムとして再定義し、ログ検索システムの分散システム化についての設計と実験、考察について述べる。第 5 章では、分散 Hayabusa のさらなる効率化について述べる。第 6 章では、大量のログデータを効率的に集計し、計算が軽量な異常検知手法の提案について述べる。第 7 章で、本研究の結論を述べる。

第2章　既存研究

2.1　ログ収集システム

2.1.1　汎用的なログ収集システム

MapReduce アルゴリズム [33] や Apache Spark[50] などの Hadoop エコシステム [3] は全文検索やログ解析によく用いられる。巨大な Hadoop クラスタや Spark クラスタはユーザに高速な検索性能／サービスを提供し、ストレージ容量や処理速度がスケールアウト可能な設計となっている。しかしながら、ネットワーク管理者が Hadoop クラスタを管理するのは難しく、構築でさえ専用ソフトウェアを必要とする。Cloudera[6] や Hortonworks[13]、MapR[18] のように Hadoop の構築を支援する企業も複数社存在する。これら大規模なビッグデータ解析基盤の性能を引き出すためには、数百台や 1,000 台を超える規模の計算機クラスタを構築する場合が多く、専任の運用管理者を置き細かなパラメータチューニングや障害対応を行うことで、やっと環境を効率的に利用することができる。環境を効率的に利用するためにかかる構築コストと運用コストはやはり莫大となる。10 台程度の規模の計算機群であればクラスタ運用や導入コストを低く抑えての環境構築は可能であるが、その規模では大きなストレージ量や大量の計算リソースを得ることは難しい。ネットワーク管理者がシンプルにクラスタを運用しようと努めても、規模が大きくなることによりハードウェアの故障率は高まり、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と経験が要求される。

Hadoop エコシステムで利用される HDFS[45] や、Elasticsearch[9] 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複雑なプロセスを経由してストレージにアクセスするために処理性能は低下する。

商用製品やサービスとして提供されるSplunk[21]やVMware vRealize Log Insinght[25]などは、ログ蓄積とインデクシング、高速検索に特化したソリューションである。検索性能は、動作させるクラスタの台数と性能に大きく依存する。しかしながらコスト面を考えると扱うログの量が増加した場合にクラスタの拡大と追加ライセンスが必要となり、高い性能と冗長性を実現するには、莫大なコストが発生する。

Googleが開発したDremel[42]をベースとしたクラウドサービスであるBigQuery[27]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デモ[1]がGoogleのエンジニアによって行われたが、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ているため、大量のクエリを発行した場合には莫大なコストがかかる。

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想定されていない。

USP（ユニバーサル・シェル・プログラミング）研究所の提唱するユニケージ開発手法[51]はシンプルなシェルプログラミングを用いて、大規模データを処理することができる。データはテキストファイルへとまとめられリレーショナルデータベースを用いることなくリレーショナルデータベースのような大規模システムを構築することができる。複数ホストで分散処理を行うこともでき、シンプルでスケールアウト可能なツールとして利用することが可能である。しかしながら高

[1] https://www.youtube.com/watch?v=swsS12c1VGE

速処理を実現するために、データを処理に合った形へと前処理し、リレーショナルデータベースと同等なデータの正規化を行う。ログデータを生ログのままの形式で扱うような処理は行えない。

ここまで言及した既存研究に関しては、どれも時系列データに特化していない汎用的なデータストアとして利用することができる。もちろん時間情報をキーとしてデータを格納することは可能であるが、そこに最適化されているわけではない。

2.1.2 時系列データに特化したデータ収集システム

汎用データストアとしてではなく時系列データを扱うことに特化したシステムも存在する。OpenTSDB はオープンソースの時系列データストアであり、大量のメトリックデータを Hadoop エコシステムである HBase[34] に、Key-Value の形で蓄積する。OpenTSDB は HBase の分散機構による書き込みスケールアウト性と高可用性を実現する。HBase は、Google の Bigtable[32] を参考設計されたオープンソースソフトウェアであり、Bigtable[32] と同様に、HBase は LSM-Tree(Log Structured Merge Tree)[43] を採用し、ログに先行書き込みを行い、続いてメモリ上のデータ構造である MemStore へとデータを格納する。MemStore のサイズが設定値を超えると、HDFS 上の HFile へとデータをまとめて書き込む。ディスクへと直接データを書き込む方法と比較して、単位時間あたりのディスク書き込み回数が小さくなるように設計されている。また、OpenTSDB からフォークされたプロジェクトである KairosDB[17] は、HBase ではなく Cassandra[5] をストレージエンジンとして用いる。KairosDB では Cassandra を用いることで、ミリ秒単位の粒度までデータを保存することが可能だが、OpenTSDB で用いる HBase では秒単位のデータ蓄積にしか対応していない。

Facebook によって開発された Gorilla[44] は、高い圧縮率によりデータをオンメモリで扱うことで、高い読み込み性能を実現した時系列データベースである。データの永続化に関し Gorilla は、HBase を用いた Operational Data Store(ODS) と呼

ばれるストレージに長期間のデータを保持する戦略をとる。Gorilla は、Gorilla 自身のメモリへのデータ書き込みと同時にライトスルー形式で ODS の両方にデータ書き込みを行う。高い読み込み性能を達成するために 26 時間分のデータを高い圧縮率でメモリへと保存し、オンメモリ処理に特化した実装として高い読み込み性能を実現する。

InfluxDB[14] は、LSM-Tree に似た Time Structured Merge Tree(TSM) を実装しており、HBase 同様にメモリ上のデータ構造に蓄積したデータをまとめてディスクに書き込む実装を採用している。さらに、データの圧縮関しては、Gorilla で実装される差分符号化手法を採用し、高いデータ圧縮率を実現する。

ここであげた時系列データベースは、時系列に特化したデータ収集機構・蓄積・検索システムとして動作する。それらは監視データのような数値メトリックを蓄積することを得意し、取得したデータは「key」と「value」のような形で保存されるものがほとんどである。さらに、取得した値をいかに圧縮してたくさんのデータをメモリ上へと蓄積し、高速に処理するかに主眼が置かれている実装となっている。本研究で扱う、時系列データではあるがスキーマが定義されないフリーフォーマットのメッセージデータのようなログを格納するには向かない実装となる。

2.2 統計手法を用いた異常検知

異常検出アルゴリズムは昔から多くの研究がなされている。時系列データに周期的な規則性がある場合には、データの波形を予測し外れた場合に異常値とする Holt-Winters 法 [38] のような予測アルゴリズムが利用できる。安定的なネットワークには適するが、大規模ネットワークの安定性に季節変動のような一定の周期は期待できない。またイベントネットワークのような大規模ネットワークでは開催期間が短いためデータの周期性を観測することは困難であり、周期性に頼るアルゴリズムは適さない。

時系列データにおいてイベントが急増したことを検出する手法の一つとして、Jon Kleinberg のバースト検知アルゴリズム [39] がある。Kleinberg のバースト検知アルゴリズムでは、解析対象のテキストに含まれるキーワードに対し、確率モデルで定義されたコスト計算を行う。このアルゴリズムはバースト状態よりも定常状態に遷移する特徴があり、一時的なバーストに反応しにくくなるという特性がある。syslog のような時系列のログを解析し異常を検知するには有効な手法であるが、ログの意味解析を行う必要があり、ログの総量が多い場合には解析処理の計算量が必然的に増加することになり、大量のログを扱う場合には適さない。

また、バーストの変化点に着目するアルゴリズムとして ChangeFinder[46] の手法がある。ChangeFinder は統計的な処理を行い外れ値ではなく変化点を見つけるアルゴリズムで、ログ総量のような時系列なデータの値の急増のように定常状態を設定できないデータに対して有効に働く。しかしながら局所的な値の変動に関しては、スコアが平滑化され、時系列に連続する大きな変動ほど異常状態を見つけられない。一瞬の突発的なログ増加に関してはスコアが低くなり異常ではないと判断される可能がある。つまり、スコアの低高から異常の重要度を判定することが難しくなる。

ARIMA（自己回帰移動平均）モデル [31] は、時系列データに適用されるモデルであり、過去の時系列データから規則性を見つけ出しその規則に基づいて将来の値を求める。ARIMA モデルで予測に用いる変数は予測対象の実績値のみでありデータの取得コストが低い。欠点として移動平均の係数の推移のようにパラメータの調整に時間がかかる点があげられるため、イベントネットワークのようは大規模ネットワークの運用には適さない。

第3章　高速なログ検索システム Hayabusaの設計と実装

3.1　はじめに

本研究では、大量のログメッセージから必要な情報を検索するためのシンプルで高速に動作するシステムである「Hayabusa」を提案する [28]。通常、大量のデータを処理する場合、Hadoop などの大規模分散処理システムが使用されるが、Hayabusa では SQL をベースとしたスタンドアロンな環境で CPU コアスケールをする処理システムを提案する。提案システムは、複雑な分散システムではなく、シンプルな運用管理が行え、ログメッセージの検索に適した単純な並列処理機構を提供する。管理者は、ログメッセージを検索するために単一のコマンドを実行することで並列処理が実現でき、複数の CPU コアに分散された単純な SQL 文が発行され、UNIX パイプラインのメカニズムと集計を得意とする UNIX コマンドを使用することで、結果を容易に得ることができる。

3.2　提案手法

3.2.1　Hayabusa のアーキテクチャ

本節では、提案システムである Hayabusa の設計指針を示す。Hayabusa のシステムとしての目標は、大量のデータをストレージへと格納し、かつ簡素なアプローチを用いて高速な検索を実現し、必要なデータを取得することである。

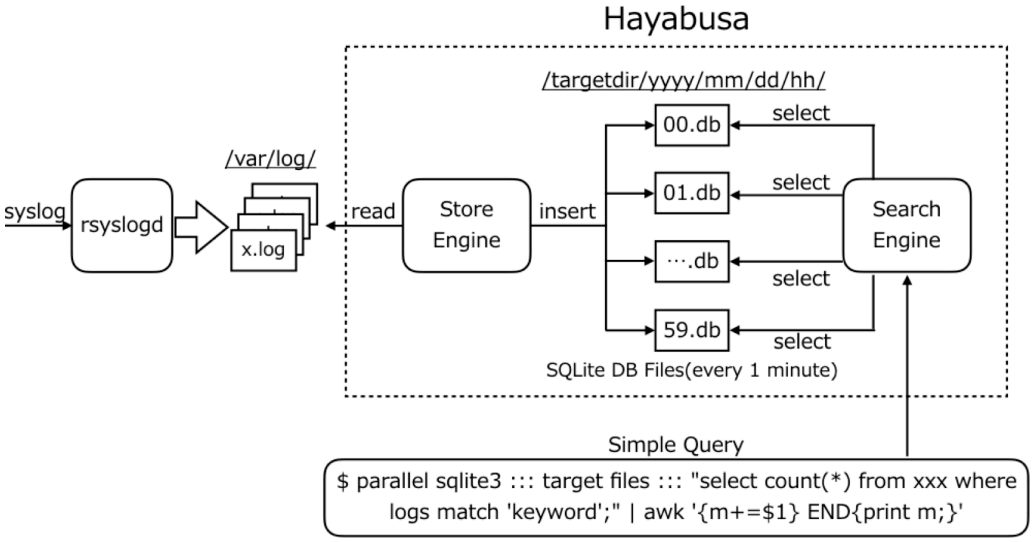

図 3.1: Hayabusa のアーキテクチャ

図 3.1 に、Hayabusa のアーキテクチャを示す。Hayabusa のアーキテクチャはとてもシンプルに設計されている。Hayabusa は、大きく StoreEngine と SearchEngine という 2 つのコアシステムで構成される。StoreEngine は、rsyslogd などの外部プログラムから出力された syslog ファイルを読み込み、対象となる syslog ファイルのデータをデータベースファイルへと変換を行う。変換されたデータベースファイルは、検索に適した構造化ディレクトリに配置される。SearchEngine は、複数のコマンドを並列に実行し、UNIX の Pipeline メカニズムを使用して実行結果を集計する。この検索操作は、複雑な分散システムを使用しないでシンプルな並列処理メカニズムを実現する。

3.2.2 データ蓄積と検索のためのディレクトリ構造設計

SQL を用いて指定期間範囲のデータをユーザが検索する場合には、通常 SQL のパラメータとしてデータの開始時間と終了時間を検索アプリケーションに渡す必要がある。SQL コマンドは、検索パラメータが増えれば増えるほど検索処理実行時間は長くなる場合が一般的である。ログファイルを検索対象とする場合に、ユー

ザまたは検索プログラムは、記録された時刻が引数としての開始時刻と終了時刻の間の範囲にある行を一致させ検索結果を返さなければならない。本来であれば、SQL のデータベーススキーマとして、starttime, endtime などをカラムとして設計する必要があるが、本提案では、時間の概念をログ時間に基づくディレクトリパスとして定義しファイルに格納する手法を用いる。検索プログラムは、SQL へと startime, endtime などの時間のパラメータをを渡す必要がなく、ディレクトリパスを使用して検索対象の時刻を指定するだけで検索範囲を絞り込むことが可能となる。

具体的には、ディレクトリパスは以下に示すように、ログ時間ファイルパスとして表現可能な時間列として表現される。「yyyy」は年を、「mm」は月を、「dd」は日付を、「hh」は時間を示し、最後の「min.db」の「min」が分を表す。

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

利点:

*"*対象のディレクトリ *+ yyyy + mm + dd + hh + min.db"*で検索時間範囲を指定することにより、検索プログラムは検索時間を体系的に指定することができる。

欠点:

本手法では StoreEngine が毎分データベースファイルを生成することとなり、管理するデータベースファイルの数が増加する。さらに、SearchEngine が検索操作を実行するとき、今まさにデータをデータベースファイルへと変換しているデータベースファイル内のデータを検索対象にすることはできない。データベースファイルへとデータ挿入作業を行っているファイルに対しては、トランザクションにより書き込み・読み込みのロックがかけられることになり、データを書き込みながらの検索は不可能となる。データベースファイルは毎分 cron により作成されることから、データベースファイルへの検索操作は、データ挿入が完了した 1 分以上前に記録されたデータに対してのみ行うことができる。つまり厳密なリアルタ

イム処理は行えず、検索は必ず 1 分以上前に記録したファイルに対するデータ検索処理が行えることが条件となる。

3.2.3 Hayabusa の実装

提案実装では、Hayabusa のコア機能としてフルテキスト検索（FTS）機能を実現するために、SQLite[22] を検索コンポーネントとして選択した。SQLite を選択する理由は、複雑な分散環境がなくても、スタンドアロン環境で高性能の全文検索を実現できる部分にある。Hayabusa の実装で用いる SQLite は、FTS(Full Text Search) の機能を主として用いるフルテキスト検索エンジンとして使用する。つまり、本提案では SQLite の通常の利用方法としてのリレーショナルデータベースとしての機能を用いず、FTS 機能を主として用いる方法を選択した。本提案では、格納されたデータを特定のカラムのキーで結合したりする一般的なリレーショナルデータベースの使用法は行わない。

SearchEngine は、GNU Parallel[12] を採用することで、複数の SQLite の SQL コマンドを並列に実行し、実行された結果を UNIX パイプラインのメカニズムを使用してデータの集約を行う。

3.2.4 高速なデータ蓄積

高速にデータを蓄積するために、StoreEngine は syslog が書き込まれたファイルを読み込み、SQLite のトランザクション機能を用いて、SQLite のデータベースファイルに一括 insert 処理で読み込んだデータを挿入する。StoreEngine は毎分動作しデータベースファイルを作成する様に設計されているため、1 分単位の SQlite データベースファイルが該当ディレクトリに作成される。。

実際に動作するデータベースファイル作成のコード (StoreEngine) を図 3.2 に示す。このコードは以下の順序で実行される。

```
1  import os.path
2  import sqlite3
3
4  db_file = 'test.db'
5  log_file = '1m.log'
6
7  if not os.path.exists(db_file):
8      conn = sqlite3.connect(db_file)
9      conn.execute("CREATE VIRTUAL TABLE SYSLOG USING FTS3(
           LOGS)");
10     conn.close()
11
12 conn = sqlite3.connect(db_file)
13
14 with open(log_file) as fh:
15     lines = [[line] for line in fh]
16     conn.executemany('INSERT INTO SYSLOG VALUES( ? )', lines
           )
17     conn.commit()
```

図 3.2: データベース作成コード

1. SQLite ファイルを FTS フォーマットで作成

2. ログを Python のリスト内包表記でオンメモリ処理

3. データの挿入を、SQL トランザクション機能を用い実行

最初に、StoreEngine は SQLite の保村形式である Full-Text Search バージョン 3（FTS3）形式でファイルを生成する。次に、StoreEngine はログファイルを読み込み、`lines` 配列として、データをメモリ上に蓄積する。データベースファイルへと書かれる値は、FTS3 フォーマットを用いて、データベースのトランザクション機能である `executemany` を用いて挿入される。データベースファイルは、データ挿入が完了するまでトランザクション機能によりロックされ、他のプロセスはデータ挿入処理が完了するまで該当データベースファイルからデータを読み出すことはできない。

利点: StoreEngine は、ログデータから読み込んだ値をメモリ上で処理し、データベースファイルへと挿入操作を行うことで高速に動作することができる。昨今

```
$ parallel sqlite3 ::: target files ::: "select count(*)
   from xxx where logs match 'keyword';" | awk '{m+=$1} END{
   print m;}'
```

図 3.3: シンプルな並列実行

のサーバ機器が保持するメモリ量的に、1 分単位のログをメモリ上で処理しても問題は発生しない。

欠点: 他のプロセスは、StoreEngine が実行するデータ挿入トランザクションが終わるまでは、ファイルへアクセスができない。したがって SearchEngine は、StoreEngine のトランザクション処理が終わるまでは、該当ファイルに対する検索が行えない。

3.2.5 高速なデータ検索

SearchEngine はデータ検索において、シンプルな並列処理機能を高速に実行する。このシンプルな並列処理機能は、GNU Parallel を用いて SQLite コマンドを並列に実行することで実現することができる。さらに、結果を UNIX のパイプライン機能を用いて集約し、awk コマンドで結果の集約を行う。

図 3.3 で示す様に、SearchEngine は並列処理を実現するために、GNU Parallel と SQLite のコマンドを組み合わせて実行する。GNU Parallel の引数として、`target files` で指定されたデータベースファイルと同等のプロセスが生成され並列に実行される。ここでの `target files` は,

```
/targetdir/2018/10/28/11/*
```

のような形で指定することも可能であり、この場合には「11:00 - 11:59」までの 60 個のデータベースファイルが検索対象となり、並列に SQLite の検索クエリが実行される。そして結果の集約機構として、UNIX のパイプラインと awk コマンドを組み合わせが、並列実行された SQLite の出力結果を集計する。

利点: 多くの SQLite の検索プロセスが GNU Parallel によって一斉に実行されるため、もしシステムが多数の CPU コアを備える場合に、検索性能がコアスケールする可能性がある。また SQLite のファイルへとアクセスが高速に行えれば、小さな I/O 待ち時間で結果を集計することが実現可能となる。

欠点: SQLite のデータが保存されているディスクへのアクセススピードが遅い場合には、ディスクへの I/O 負荷が高くなる可能性がある。なぜなら、DB が保存されているディスクへ SQLite コマンドが並列実行され、同時に複数のプロセスからディスクアクセスが実行されるからである。

3.3 評価

この節では、提案システムである Hayabusa の評価を行う。評価を行うにあたり、実環境で収集された大規模データセットを使用してストレージと検索のパフォーマンスを調査するためのベンチマークを実施した。

評価で利用する Linux では一度ファイルから読み込んだデータは、Linux のファイルキャッシュへとキャッシュされてしまうため、この実験で測定を行うたびに、Linux ファイルキャッシュをクリアし、ファイルキャッシュへとデータが蓄積されていない状態にした。キャッシュがクリアされた状態は free コマンドの「buff/cache 項目」で確認できる。具体的には、測定を開始する前にキャッシュをクリアするため以下のコマンドを都度実行した。

```
# echo 3 > /proc/sys/vm/drop_caches
```

3.3.1 実験環境と実験データ

実験ホストのスペックは 表 3.1 である。本実験のベンチマークには、実環境で収集されたデータセットを用いた。このデータセットは、Interop Tokyo ShowNet

表 3.1: Hayabusa の実験環境

CPU	Intel Xeon E5-2670-v3（12 コア、2.3 GHz）× 2
メモリ	DDR4 384GB
ディスク	800GB (NVMe)
OS	CentOS 7.1 (Linux カーネル 3.10)

[20] 2016 で収集を行った syslog データである、ShowNet は 400 を超えるネットワーク、サーバ機器、IoT デバイスで構成された日本の大規模なデモンストレーションネットワークであり、Interop Tokyo ShowNet 2016 の各機器から受信したログメッセージの数は、2 週間にわたって収集された 4350 万行で構成されている。本データセットは、セキュリティのデータを除いた通信機器のログとなっている。

3.3.2 ストレージ性能

Interop Tokyo 2016 の ShowNet では syslog メッセージの受信速度が毎秒 2 万メッセージに達しているおり、この数値を実験の目標値とした。1 分間に格納する syslog メッセージの総数は、1200k（20k x 60 秒）となる。したがって、ShowNet クラスのネットワークよって生成される全ての syslog メッセージを受信するには、Hayabusa は 1 分間に 1200k メッセージを格納できる必要があるため、StoreEngine の目標値として、1 分間に 1200k メッセージのデータを蓄積することとする。syslog メッセージ量が 1 分あたり 1200k メッセージ以上に増加しても処理できる場合には、StoreEngine は ShowNet におけるメッセージ数を満たし、かつバースト的に発生する syslog 量や規模が拡大した環境でのログ処理も可能であると推測できる。そこで、本研究で実施した目標と性能指標の予備実験として、1200k メッセージ以上のベンチマークデータとして 100k、1M、10M というメッセージデータ量を蓄積して、パフォーマンス測定を行った。

図 3.4 は、1k、10k、100k、1M、10M レコードの挿入に費やされた平均時間を示す。挿入するメッセージ数が増加するにつれて、データ挿入にかかる時間がリ

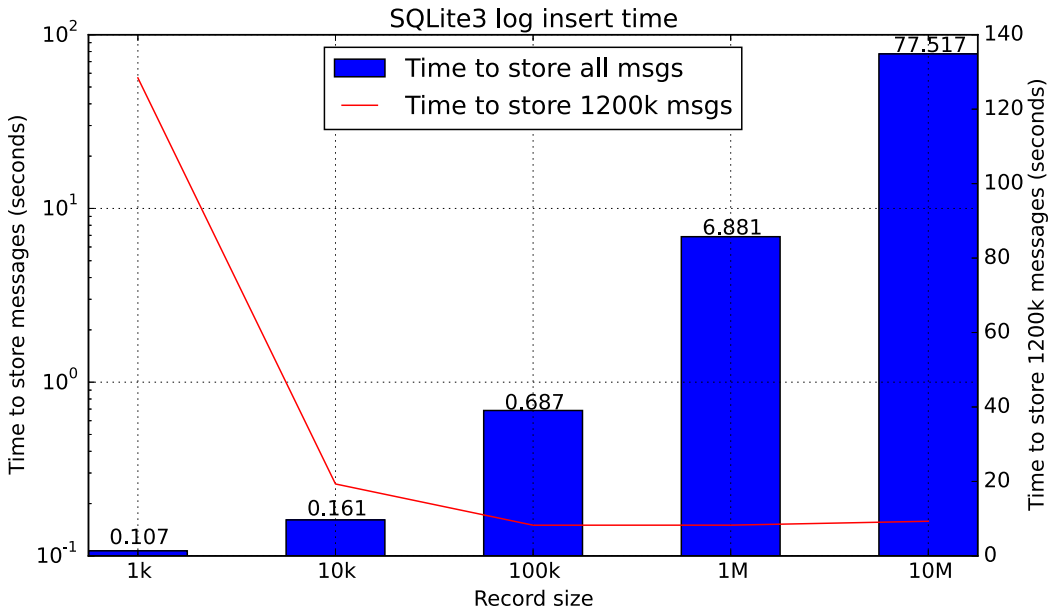

図 3.4: SQLite データ挿入 time

ニアに増加しました。図 3.4 は、1200k メッセージを保存するのに必要な時間も示す。1k の場合に 0.107 秒のデータ挿入時間がかかるため、1200k のデータを挿入した場合にかかる時間は (0.107 x 1200) = 約 128 秒と予想されたが、1M レコードと 10M レコードの場合に、1200k のメッセージを保存するのにかかる平均時間は約 8 秒だった。10M データの挿入ではトータル 77 秒かかっているが、そのうち 1200k のメッセージを挿入するのにかかる時間は約 8 秒ということになる。1200k のメッセージ挿入にかかる時間が、1k の場合の予想値と 10M の場合で大きくずれた理由は、実際のデータ挿入時間よりもトランザクション処理に大きく時間がかかっていると予想される。結果として、StoreEngine は望ましいパフォーマンスを発揮することができ、バーストトラヒックが発生したとしても、十分にデータの保存が可能と予想できる。

```
$ parallel --dry-run sqlite3 ::: /path/1k-[0-9].db ::: "
   select count(*) from syslog where logs match 'noc';"
sqlite3 /path/1k-1.db select\ count\(\*\)\ from\ syslog\
   where\ logs\ match\ \'noc\'\;
sqlite3 /path/1k-2.db select\ count\(\*\)\ from\ syslog\
   where\ logs\ match\ \'noc\'\;
sqlite3 /path/1k-3.db select\ count\(\*\)\ from\ syslog\
   where\ logs\ match\ \'noc\'\;
...
```

図 3.5: GNU Parallel のドライラン実行の結果

3.3.3 検索性能

高速データ検索を行うために、Hayabusa は GNU Parallel を利用してシンプルな並列処理メカニズムを実現した。並列処理機構は、GNU Parallel、SQLite、UNIX パイプライン、および awk の集計プログラムの組み合わせで実装を行った。GNU Parallel は、コマンドライン引数として渡されるファイルの数と同じ数のプロセスを実行し、各ファイルに対して SQLite プロセスを起動する。本提案では、以下の 2 つのベンチマークを採用し性能を測定した。

1. 複数のレコードサイズを持つ単一のデータベースファイルに対する単一の検索プロセスの実行
2. 複数のレコードサイズを持つ複数のデータベースファイルに対する GNU Parallel を用いた複数の検索プロセスの実行

図 3.5 が示すように、`dry-run` オプションを用いることで、GNU Parallel コマンドがどのようなコマンドを実行するかを確認することができる。

並行して実行される SQLite コマンドの結果は、図 3.6 に示すように、awk コマンドによって集計が行われる。一連の SQLite コマンドは、GNU Parallel で実行される並列処理プロセスの一部であるが、集約処理である awk コマンドは並行して実行されない。全ての SQLite コマンドが実行されたのち、検索結果が単一プロセスである awk コマンドによって集約される。

```
$ parallel sqlite3 ::: /path1k-[0-9].db ::: "select count(*)
    from syslog where logs match 'noc';" | awk '{m+=$1} END{
   print m;}'
```

図 3.6: awk を用いた集計サンプルコード

3.3.4 ベンチマーク結果（1 つのファイルで異なるレコードサイズ）

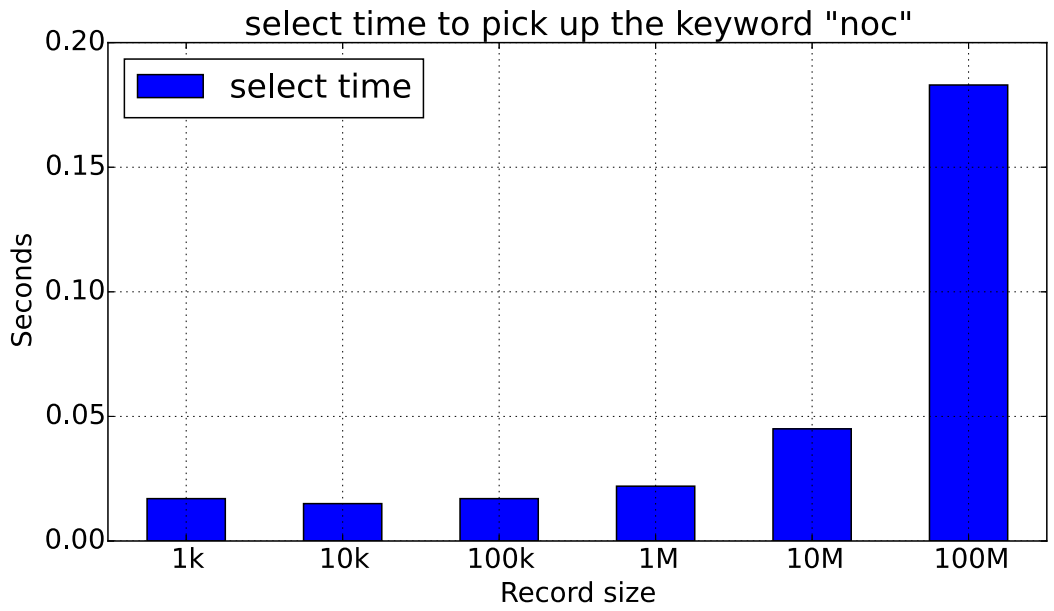

図 3.7: SQLite の検索時間

図 3.7 は、異なるレコードサイズ（1k、10k 100k、1M、10M）の 1 つの SQLite データベースファイルの合計検索時間の結果を示す。各ベンチマークでは、FTS を使用してファイル内に格納された単語である「noc」を検索した。結果として、100M レコードのデータベースファイルでも、FTS 機能を用いたデータベーステーブルでは、検索プロセスが 0.19 秒以下で実行が完了する。したがって、SQLite 上の FTS 機能は 1 つのデータベースファイルに対して高速に検索が行われることが実証される。

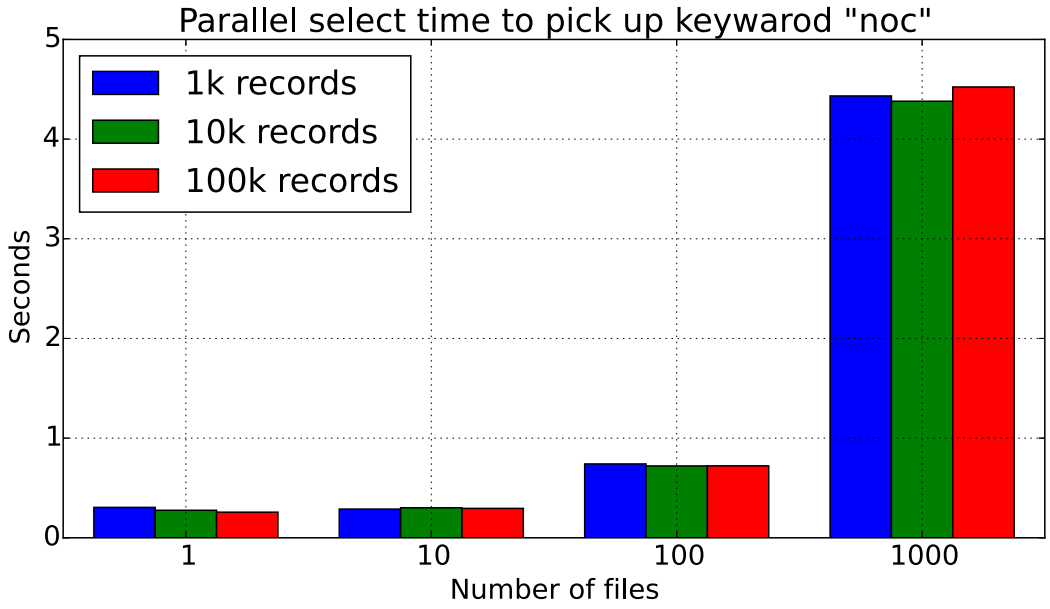

図 3.8: SQLite の並列実行時間

3.3.5 ベンチマーク結果（異なるファイル数）

前節での実験では、単一のデータベースファイルを対象とした検索時間をベンチマークした。次に、さまざまなレコードサイズ（1k、10k、100k）の複数ファイル（1 個、10 個、100 個、1,000 個）に対する検索時間を測定した。この実験では、GNU Parallel を用いて評価実験を行った。

図 3.8 は、検索結果のベンチマークを示す。グラフの x 軸はファイル数を示し、y 軸は時間を秒単位で示している。結果が示す通り、ファイル数にかかわらずレコード数が増加しても SQLite の FTS 機能は検索速度を維持したままとなり、検索スピードはファイル数に依存せず高速なままでだった。

3.3.6 Apache Spark との比較

提案手法を他の全文検索手法と比較するために、Apache Spark を用いて検索実行時間を測定し、Hayabusa との比較実験を行った。スタンドアロンの Spark 環境で、`spark-submit` コマンドを使用して Python コードを実行した。

```
from pyspark.sql import SQLContext
from pyspark import SparkContext

sc = SparkContext(appName = "test")

sqlContext = SQLContext(sc)
lines = sc.textFile("/path/to/100k-*.log")

print(lines.filter(lambda s: 'noc' in s).count())
```

図 3.9: PySpark 検索コード

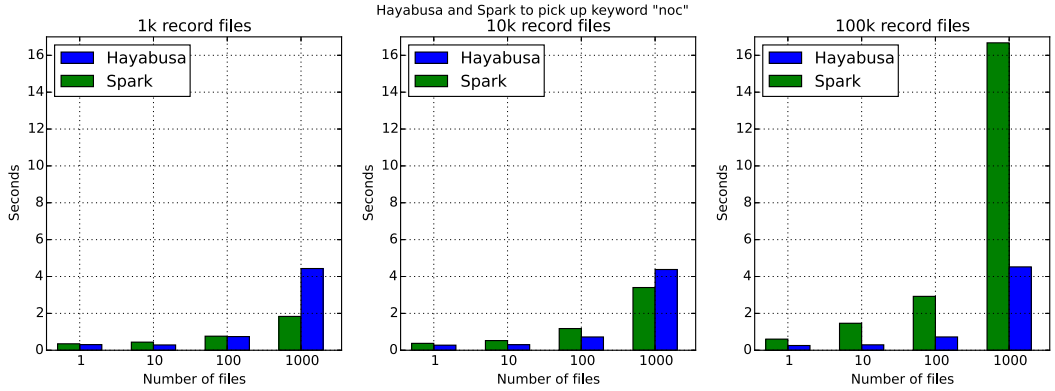

図 3.10: Hayabusa と Spark 検索時間比較

評価に用いた Python コードは、図 3.9 に示す。本比較実験では、Hayabusa が検索に用いたログと同様の syslog データファイルを、Apache Spark を Python を用いて利用可能な pyspark ライブラリを用いて全文検索を行う。Hayabusa と Spark 共に検索する元データは同等のものである。

図 3.10 に示すように、Spark と Hayabusa はそれぞれファイル数（1 個、10 個、100 個、1,000 個）を対象として検索処理を実行した。さらに、保存されたレコード数（Spark の場合はレコード数ではなくログの行数）を変更しながら、実験を行った。レコード（またはログ）のサイズが 1k と 10k の場合、単語を検索するのに必要な時間は、Spark と Hayabusa の両方でほぼ同時であった。

ファイル数が 100 個の場合、Hayabusa は Spark より高速に動作した。ただし、1k および 10k レコードでかつ 1,000 ファイルの場合、Spark は Hayabusa より高

表 3.2: Spark との比較環境

CPU	Intel Xeon E3-1231-v3（4 コア、3.4 GHz、8 MB キャッシュ）
メモリ	DDR-3 32GB
ディスク	400GB　Intel SSD910（PCIe 2.0）
OS	CentOS 7.1 (Linux カーネル 3.10)

速に動作した。それにもかかわらず、大きなファイル（100k レコード）の場合、Hayabusa は Spark より約 4 倍高速に動作した。

3.3.7　分散処理環境の比較

本評価では、複数ホストの Spark 環境と、本提案に基づくスタンドアロンの Hayabusa を比較した。評価では、同じ構成の 3 台のサーバを使用した。実験ホストのスペックは 表 3.2 である。

ファイル数は、1 個、10 個、100 個、および 1,000 個ファイルを配置するパターンを用いた。 データベースレコードのサイズとログの行数は共に 100k とした。Apache Spark の環境は、CDH（Cloudera のオープンソースソフトウェア配布）[7] を使用して構築を行った。

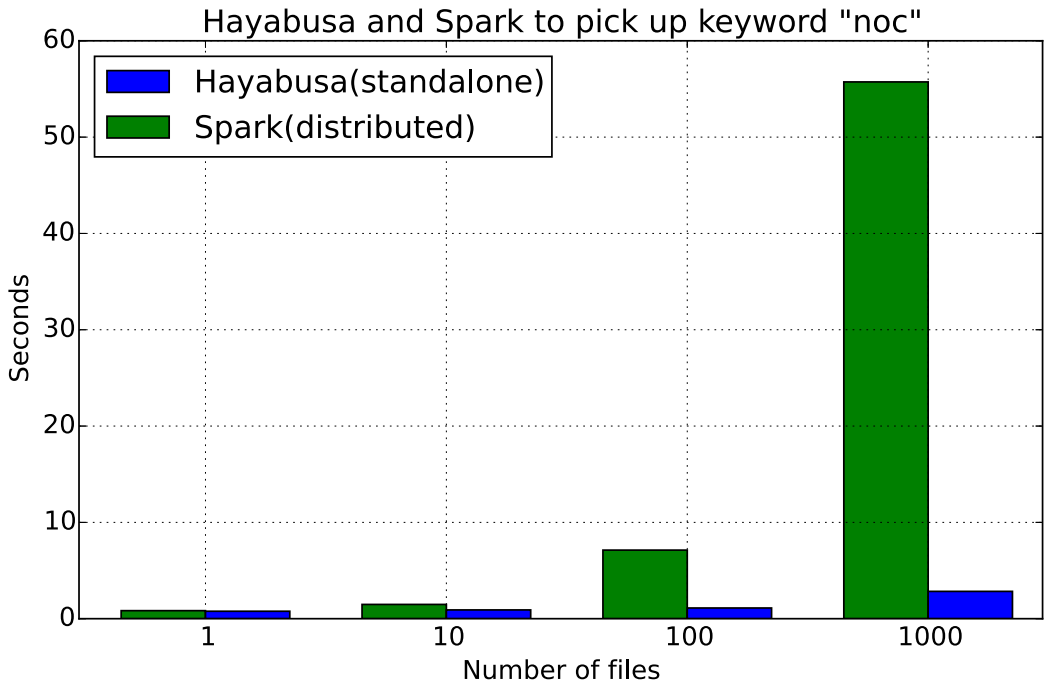

図 3.11: 分散 Spark 環境とスタンドアロン環境の Hayabusa の比較

Spark は、Hadoop の MapReduce[50] よりも 10 倍以上高速に動作すると言われる。しかし、図 3.11 に示すように、Spark はスタンドアロン環境の Hayabusa よりも処理に時間がかかった。1,000 個のファイルを対象とした場合、スタンドアロン環境の Hayabusa は、分散環境の Spark 環境より約 27 倍高速に動作した。

3.4 考察

3.4.1 時系列ログ検索に適したデータ構造

本提案では、時系列ログデータの保存手法として時間範囲をもとにしたファイル分割手法を設計した。ディレクトリ構造とファイル名を時間検索の条件として扱うことで、従来のアプローチで要求される SQL のクエリパラメータとしての時間条件を省略することで、検索速度を向上させることができる。しかしながら、分単位でデータベースファイルを作成することで、ファイル数が増加するという欠

点が存在する。そのような欠点があるにもかかわらず、適切なファイルパス上でGNU Parallel を使用して並列処理を行うことができるため、検索操作の効率が向上する。

3.4.2 SQLite の検索速度

SQLite の FTS 機能は、B-Tree のインデックス機能を使用して単語の高速検索を実現する。FTS テーブル自体は大きなインデックスであり、標準的な SQLite の LIKE 操作で実行される文字列照合処理よりも高速な処理を提供する。ログメッセージのフォーマットが既知でかつ構造化されている場合、特定のメッセージに適したデータベーススキーマを設計できるため、検索パフォーマンスが向上する可能性がある。しかし syslog のメッセージ部分のフォーマット定義はフリーフォーマットであり、さまざまな機器からのログメッセージのパターンを特定の形に構造化できない場合には、全文検索が有効である。

SQLite による全文検索は、他のプログラムでの全文検索と比較して、非常に高速で動作する。これは、SQLite の全文検索処理実装方法によるものである。SQLite はメモリ内でデータベースファイルを処理するために、mmap システムコールを用いる。他のプログラムでは、ファイルコンテンツの読み込みに read システムコールを使用するため、全文検索処理では SQLite よりも遅くなる可能性がある。

3.4.3 GNU Parallel の並列性

Apache Spark は、YARN(Yet Another Resource Negotiator) と呼ばれる分散環境のスケジューラを使用して効率的にプロセスを実行する。しかしながら、GNU Parallel はスタンドアロン環境で複数のプロセスを同時に効率的に実行できる。本研究では、Hayabusa はスタンドアロン環境での並列 SQLite 実行を採用しているが、この環境は分散するサーバ上に構築された Spark 環境より速いことが実証さ

れた。Hayabusa の並列実行はログ分析のための強力な環境であるといえる。なぜなら、Hayabusa のアーキテクチャと実行処理部分は非常にシンプルに設計されている。

Apache Spark は通常、HDFS ファイルシステムと統合され、抽象的なストレージアクセスレイヤを提供する。抽象レイヤを使用することで、大量の分散ファイルシステムを 1 つのファイルシステムのように見せることが可能だが、メタデータサーバを経由して HDFS にアクセスするには処理に時間がかかり、Hayabusa が用いる SQLite の mmap システムコールのように、Spark は分散ファイルディレクトリを直接メモリにマップすることはできない。

3.4.4 システム管理コスト

一般的なログ処理に用いられる Hadoop エコシステムは、大規模な分散処理システムを管理する必要があるため安定的に運用することはとても難しい。分散処理システムには、システムの一部が動作不能であっても高可用性を実現するために、多くの複雑な管理プログラムが存在し、システムの健全性を維持する。さらに、データ分析のためのプログラムまたはアプリケーションは、複数の複雑な計算ノード上で実行される。これらの複雑な基盤上で動作するプログラムやアプリケーションに問題が発生した場合、システム管理者は問題の場所を特定することが非常に困難となる。対照的に、Hayabusa はスタンドアロンのサーバで動作することで、問題が発生した場合にも問題箇所の特定にかかる原因究明コストを最小限に抑える。

3.4.5 シンプルなコマンド実行形態

Hayabusa の実行コマンドは、図 3.6 に示すように、単一行のコマンド群である。評価実験では、Spark フレームワークが提供する `spark-submit` コマンドを使用し

た。図 3.9 に示す Spark 全文検索コードと比較して、Hayabsa の検索コマンドはとてもシンプルであるといえる。

3.5 まとめと今後の課題

本研究で Hayabusa は、データ蓄積と検索に関して、とても優れたパフォーマンスを発揮した。これは、Hayabusa のデータ構造とデータ保存の設計が、時系列ログデータに適した時間軸のファイル分割手法に基づいているためである。

StoreEngine は、Interop Tokyo 2016 の実際に生成された syslog のメッセージ生成速度（1 分あたり 1200k メッセージ）に対して、十分に高速なデータベースファイルの読み込みとデータの保存処理に、7 秒未満の処理時間で完了する性能を示した。SearchEngine は、登録されているレコードの数とデータベースファイルの数の影響を受けずに、高速全文検索を実現した。結果として Hayabusa は、分散 Spark 環境よりも約 27 倍もの速さで動作するキーワード検索を実現した。本節で提案する Hayabusa は、GNU Parallel を使ったシンプルで強力な並列処理で実装を行った。Hayabusa はシステムやネットワークの問題をすばやく解決できる効率的なシステムだと考えられる。

Apache Spark はテキスト検索機能を提供するが、メモリ内で繰り返し検索するためのキャッシュのメカニズムを備えている。さらに、Spark には機械学習のための複数のライブラリを利用することができる。しかし、ネットワーク管理者にとってのトラブルシューティングの最も簡潔で重要な方法は、高速なログの保存と検索である。したがって、Hayabusa は Spark の様な複雑な運用や利用オペレーション無しに、ネットワーク管理者にとって重要な情報検索機能をシンプルに提供する。

Hayabusa は単一のサーバ上でしか動作しないため、ストレージの記憶容量が制限される。第 5 章では、ネットワークストレージを用いて Hayabusa のストレージの容量制限に関する制約を緩める手法を解説する。

第4章　ログ検索システムの分散システム化

4.1　提案手法

本研究では、第3章で設計、実装を行った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ことを目標とする[55, 53]。

4.1.1　アーキテクチャ

分散Hayabusaを定義する上でストレージとスケジューラに関して定義を行う。アーキテクチャを定義するにあたり、システムの複雑化を避けるためにスタンドアロン版Hayabusaのシンプルさを継承しつつ、処理性能を低下させない設計を目指す。Hadoopの様な分散処理システムは、システムが健全・堅牢に動作するために常にシステム内部に複雑なデータのやりとりが発生する可能性があるが、分散Hayabusaでは、エラー処理や処理失敗時のリトライ処理を考慮せずにシステムのシンプルさの維持と処理速度を重視する方針で設計を行う。図4.1に分散Hayabusaのアーキテクチャを示す。

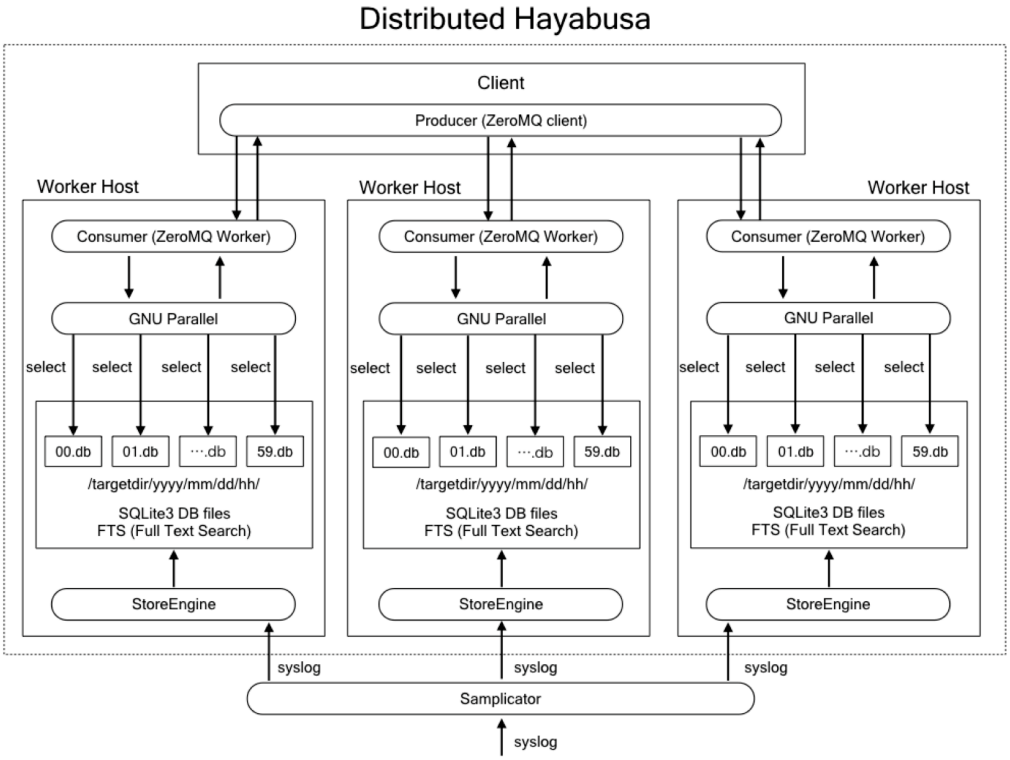

図 4.1: 分散 Hayabusa のアーキテクチャ

4.1.2 ストレージ

分散 Hayabusa のストレージは、スタンドアロン版と同様に時系列ログデータをディレクトリ階層に時間単位でマッピングし、時間のクエリ条件を指定しなくても、時間範囲の検索ができる形とする。

さらに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。

4.1.3 スケジューラ

分散 Hayabusa は、検索処理のスケジューリングに RPC（Remote Procedure Call）を用いたロードバランシングと Worker ホストによるプロセス実行機構を用いる。分散したホストへの検索処理投入に関して、Producer/Consumer モデルを用いた RPC と処理を均等に振り分けるロードバランシングにより、各ホストへと順番に検索処理が投入される。各ホストで受け取った検索処理がスタンドアロン版の Hayabusa と同等に並列検索として実行され、結果を Worker 経由でクライアントへと返す。

4.2 実装

4.2.1 データの複製による並列蓄積

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 [24] を利用した。UDP Samplicator は、受信した UDP パケットの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。図 4.1 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。

4.2.2 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。そのため、大量の syslog を受信し負荷が上昇した際にプロセスのコア使用率が 100%となりパケット転送処理が追いつかなくなる場合があり、データが破棄される可能性がある。そこで本提案では socket のオプションに「SO_REUSEPORT」を利用して UDP Samplicator へパッチを当て、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。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、複数 CPU コアを利用し複数をプロセス起動することで解消した。

4.2.3 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[36] 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」など多様な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。

4.2.4 ZeroMQ の Push/Pull

図 4.2 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。

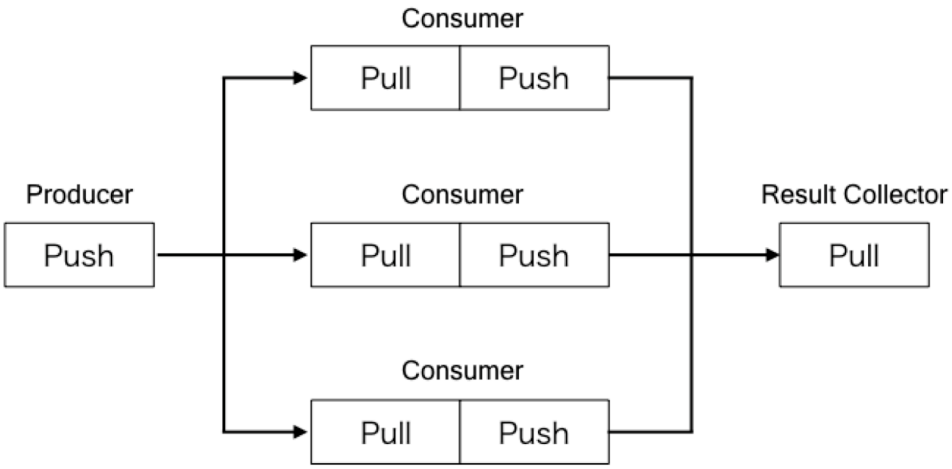

図 4.2: Push/Pull パターン

1) Producer がリクエストをキューイング (Push)

2) Consumer が Producer からリクエストを取得 (Pull)

3) Consumer が結果を Result Collector へと送る (Push)

4) Result Collector で取得した結果 (Pull) を集計する

本提案でのクライアントは、Producer と Result Collector 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 4.3 にクライアントのソースコードを示す。

```
1  import sys
2  import zmq
3
4  context = zmq.Context()
5  sender = context.socket(zmq.PUSH)
6  sender.bind("tcp://*:5557")
7  receiver = context.socket(zmq.PULL)
8  receiver.bind("tcp://*:5558")
9
10 cmd = 'parallel target-data "SQLite3 Query Strings"'
11
12 REQUEST_SIZE=100
13 for i in range(REQUEST_SIZE):
14     sender.send(cmd.encode('utf-8'))
15
16 cnt = 0
17 while True:
18     message = receiver.recv()
19     print(message.decode('utf-8'))
20     cnt = cnt + 1
21     if cnt == REQUEST_SIZE:
22         sys.exit()
```

図 4.3: クライアントソースコード

図 4.4 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。

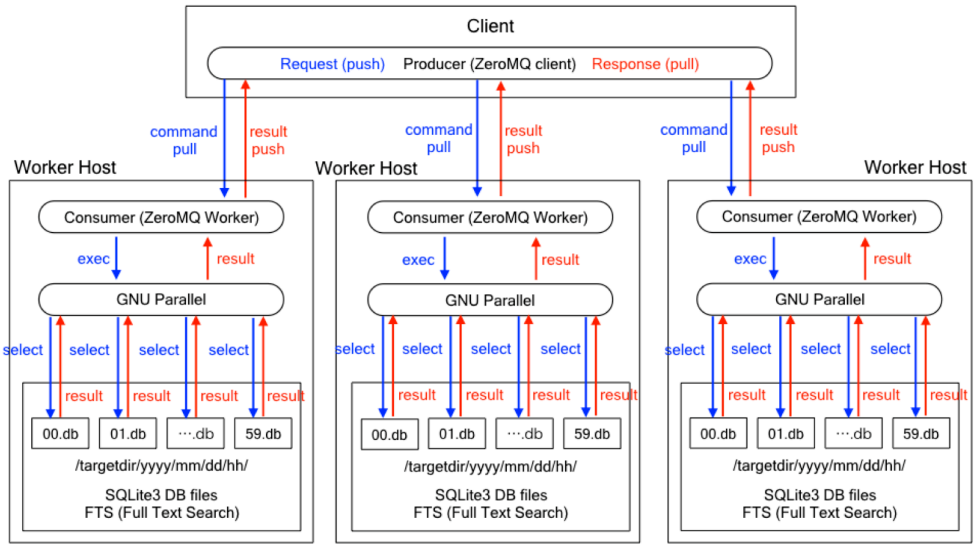

図 4.4: 分散 Hayabusa で用いる ZeroMQ の Push/Pull パターン

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。処理結果は、TCP の 5558 番ポートで受け付け集計を行う。

```
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 4.5: Worker ソースコード

次に、Worker のソースコードを図 4.5 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

表 4.1: 分散 Hayabusa の実験環境

EC2 インスタンス	c4.4xlarge
vCPU	Intel Xeon CPU E5-2660 (2.9GHz/16 core)
メモリ	30GB
ディスク (EBS)	SSD 8GB (OS) + SSD 50GB (Data)
OS	Ubuntu 16.04.4 LTS (Xenial Xerus)

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。

4.3 評価

本研究では、スケールアウト試験を行うために Amazon Web Service [1] にて提供される EC2 上の仮想サーバ群を用いた。スケールアウト試験では、仮想サーバを 1 台から 10 台へと増やし検索速度の評価実験を行う。処理ホストの他に、分散クエリのリクエストを行うクライアントホストを 1 台用意する。実験ホストのスペックは 表 4.1 である。

4.3.1 分散検索

本検証では実データを用いて検証を行うために、Interop Tokyo ShowNet 2017 の syslog データを用いた。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。将来的にはさらなる syslog 受信量の増加が見込まれるため、10 万件の syslog を用いて分散環境でのスケールアウト検索の検証を行った。

4.3.2 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 4.6 に示す。

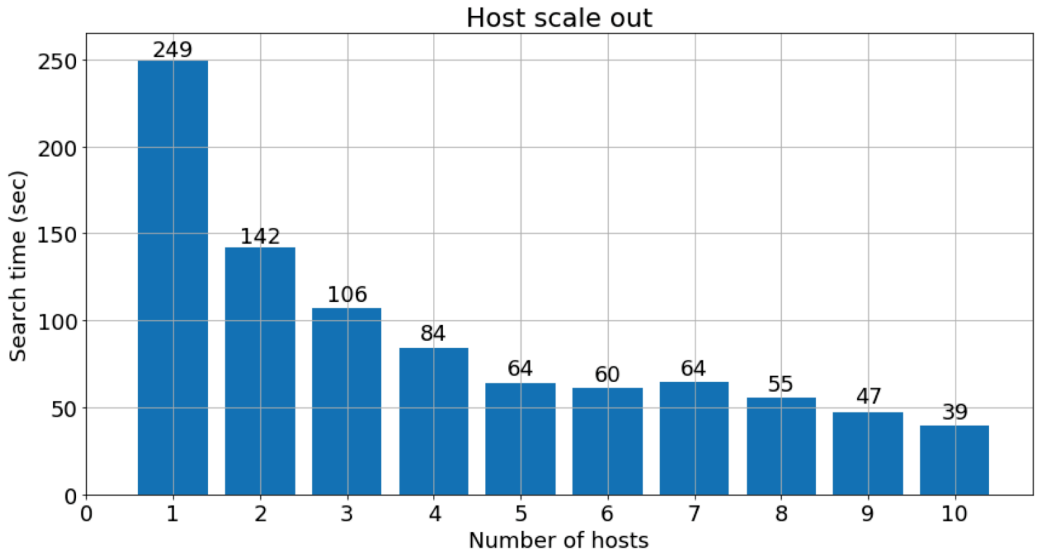

図 4.6: 検索ホストのスケールアウト性能

ホスト 1 台時の検索処理時間は約 249 秒だが、ホストの台数を増やすに従い処理時間は減少し、10 台のホストの処理時間は約 39 秒になる。ホストを増加させた場合に検索処理時間が反比例形で減少しなかったがスケールアウトした結果となった。ここでの各処理時間は 10 回試行した平均値である。

4.3.3 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し、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仮想マシンの vCPU コア数であり、vCPU コアの数だ

け Worker プロセスを増加させた場合にどの程度性能が伸びるか試験を行った。処理結果を図 4.7 に示す。

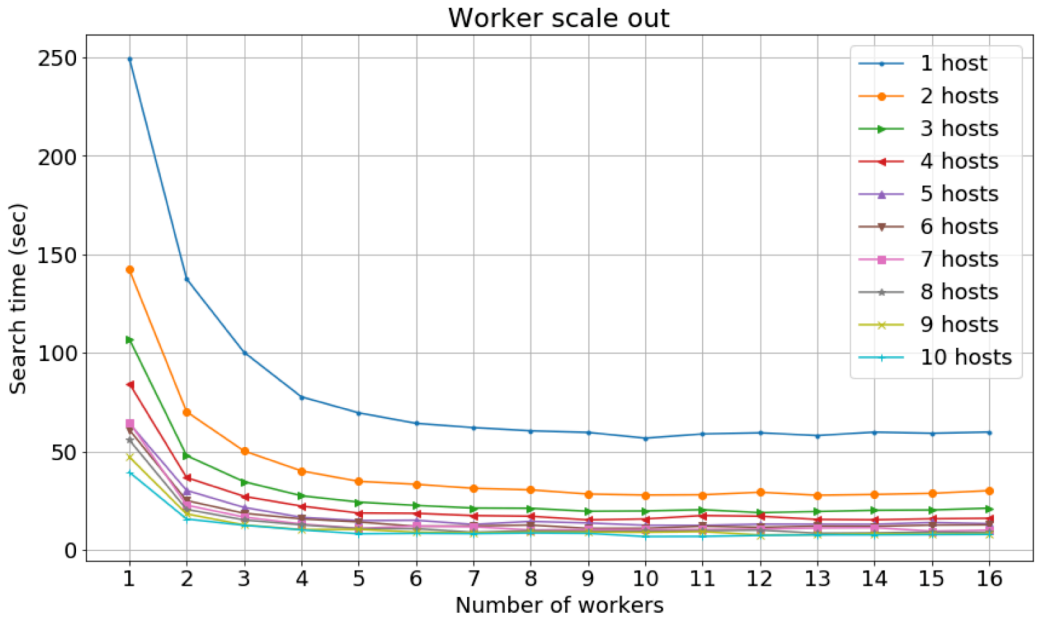

図 4.7: Worker プロセスのスケールアウト性能

各処理ホストを 1 台から 10 台に増加させ、さらに Worker プロセス数を 1 から 16 まで増加させた図となる。ホスト数が 1 台から 10 台まで変化するなか、Worker 数が 10 あたりで最高値となる。1 ホスト 1 Worker 時に約 249 秒かかっていた処理が、10 ホスト 10 Worker 時に約 6.8 秒まで処理時間が短縮し、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。こちらの実験も、各処理時間は 10 回試行した平均値である。

4.3.4 AWS EMR との比較

次に、AWS 上でサービス提供される、Amazon EMR (Elastic MapReduce) サービスとの比較を行った。EMR は指定した台数で Apache Hadoop/Hive/Spark などの Hadoop エコシステムが構築できるサービスである。かつ、Amazon S3 サービス

にデータを置くことで、HDFS 環境を準備することなく EMR から直に S3 のデータを参照することが可能となる。

本実験では、分散 Hayabusa の評価と同等の性能の EC2 インスタンス (c4.4xlarge) を用いた。EMR では、構成として 1 マスターノードが必須となり、それ以外の環境として実際にデータの処理を行うコアノードが必要となる。ホストを増加させた時のスケールアウト性能を確認するために、コアノードの数を 2 台から 10 台まで増加させ評価を行った。EMR のリリース番号は emr-5.12.0 となり、アプリケーションとして、Apache Spark がメインのパッケージ（Spark: Spark 2.2.1 on Hadoop 2.8.3 YARN with Ganglia 2.7.2 and Zeppelin 0.7.3）を選択した。

S3 に、1 ファイル 10 万行の syslog ファイル 1 日分に当たる 1,440 ファイル用意し、クライアントから 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して、対象の syslog 行サイズが 144 億行という分散 Hayabusa と同等の情報量へアクセスする状況で実験をおこなった。

クライアントは、図 4.8 で示す PySpark のコードをマスターノードで実行し、各コアノードで検索処理を行う。5 行目で、S3 から対象ログデータを読み込み、6 行目で Spark の機能である RDD (Resilient Distributed Dataset)[49] へとデータをロードする。RDD は複数コアノード間でシェアされる分散共有メモリであり、複数ノード間で高速にデータを検索できる。

```
1  import time
2  from pyspark.sql import SQLContext
3
4  sqlContext = SQLContext(sc)
5  lines = sc.textFile("s3://abe-work/ssd2/benchmark-log/files
       /100k/100k-*.log")
6  lines.cache()
7
8  for i in range(5):
9      start = time.time()
10     [lines.filter(lambda s: 'noc' in s).count() for i in
           range(100)]
11     elapsed_time = time.time() - start
12     print elapsed_time
```

図 4.8: PySpark ソースコード

処理結果を図 4.9 に示す。

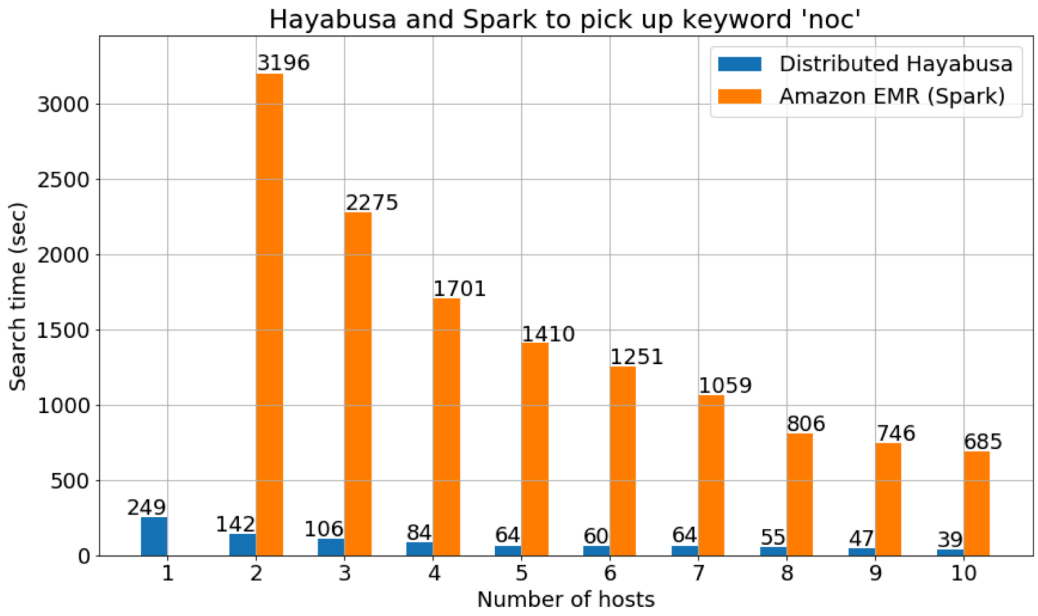

図 4.9: Hayabusa と Spark の性能比較

EMR ではホスト 1 台での環境は構築できないため、ホスト 2 台からとなる。こちらの各処理時間は 5 回試行した平均値である。EMR 上の Spark でもホストを増加させた場合に、全文検索の処理性能がスケールアウトしていく。本実験では、同

じ syslog データに対する全文検索結果として、EMR Spark が 10 台構成の場合と比較して分散 Hayabusa の方が約 17 倍高速に動作する事を確認した。

4.4 考察

4.4.1 検索性能／処理のスケールアウト

4.3.2 項と 4.3.3 項で示した結果から、1 台の処理ホストとの検索時間と比較した時に、ホスト数のスケールアウト実験で約 6.3 倍処理時間が高速化した。またホスト数と 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 249 秒かかっていた検索時間が約 6.8 秒まで短縮し、約 36 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 秒台でフルスキャンできたということは、数千台以上のサーバを用いている 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。

4.4.2 ログデータ蓄積の並列化

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。機器故障以前に、各処理ホストへとデータが正しく配送されずに、処理ホスト間で保持するデータの一貫性が保証されない問題が生じる場合があり、一貫性がないデータへとアクセスを行った場合に異なる値が返される可能性がある。また障害によりログの欠損が発生した場合には、同様に一貫性のないデータが結果として得られる可能性がある。処理ホスト間のデータ一貫性に関しては、ファイルのハッシュ値を用いたチェック機構を実装することで監視を行うことができ、監視によって一貫性の崩れが見つかった場合に、該当ホストを処理対象ホスト群から外すといった運用の実現が考えられる。障害によるログ欠損への対処は上記チェック機構を利用し、正しいログデータを該当ホストに複製する運用を行うことによりデータに一貫性のある状態へと復元可能となる。

今回採用した syslog の複製と言う手法は、本質的にスタンドアロンで処理可能なストレージデータ量しか処理していないことになる。問題を根本的に解決するには、ストレージの分散化が不可欠である。例えば月毎に時間軸単位にホスト増加させ処理を分散させる方法や、Amazon EMR のようにクラウドストレージにデータを保存し、各処理ホストから分散アクセスを行いデータを読み取る手法などが考えられるが、検索性能とストレージ性能の予備実験が必要となり、今後の課題とする。

4.4.3 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セスの数も少ない。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

ラブルが発生した場合には、トラブル原因を把握するコストが高まる。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。

4.4.4 他の分散処理アーキテクチャの違い

本研究では、Amazon EMRとの比較実験を行い、分散Hayabusaの方がEMRより約17倍高速に全文検索を行うという結果を得た。これほど処理に差が出るにはいくつかの点があげられる。EMRの場合にはボトルネックとなりうる箇所が数カ所あげられる。

4.4.5 クラスタのリソース管理

EMRで使われるSparkはHadoopのリソース管理機構であるYARN（Yet Another Resource Negotiator）[47]上で動作する。YARNは、Hadoopエコシステムにおけるリソース管理機構を司っており、クラスタ内のリソース割り当てや実行されるジョブの監視とトラッキングを行い、さらにクラスタが保持する共通のデータセットに対するアクセスを管理する。これによりクラスタ全体の健全性と、マルチテナント化した場合に、複数ジョブの制御を行うことが可能となる。

分散Hayabusaには現状リソース管理機構はなく、クライアントから届いたリクエストは速やかにWorkerで実行するモデルになっている。これは分散Hayabusaが高速に動作する仕組みの一つではあるが、ジョブの管理やトラッキングを行っていない以上、トラブルが発生した場合にエラーハンドリングやリトライ処理ができないことを意味する。本提案でのシステム構成では、トラブルが発生したことを利用者が把握することはできず、処理結果からトラブルを推測することしかできない。

4.4.6　スケジューラの構造

Spark は、タスクをスケジューリングする前にタスク実行の有向非巡回グラフ (DAG: Directed Acyclic Graph) を作成する。また Spark は、RDD を用いて分散共有メモリ内にデータを永続化し、DAG 間でデータを共有する。このように DAG 間でデータを共有（ディスクに中間結果を書き込まない）して最適化することで、高速にジョブを完了することができる。

Hayabusa が実現するスケジューリング機構は、ZeroMQ クライアントが行うロードバランシングと GNU Parallel の実行スケジューリングに依存する。シンプルでオーバヘッドが少ないので高速に動作するが、Spark のように最適な実行計画を計算し、実行計画に沿って分散共有メモリを使いながら処理を実現するようなものではない。

4.4.7　ストレージに対するアクセス性能

本実験で EMR は S3 からデータを読み取ったが、本来であれば Hadoop エコシステムは HDFS 上にデータが分散されて設置される。HDFS を使った場合にはデータがある一定以上のサイズになった場合にはブロック単位で各ホストに分散して配置されることになる。その場合には、データへのアクセスは HDFS のメタデータ機構を経由することになり、ストレージアクセスが低速になる。

Hayabusa はデータを 1 分単位の SQLite3 データベースとして保持し、個々のファイルは全文検索に特化した FTS フォーマットでインデクシングされた高速な検索機構を有する。さらに、時間レンジで絞り込みをかけたい場合に、Hayabusa の手法では時間を SQL のクエリ条件に指定する必要がなく、本来クエリ条件としては遅くなる可能性の高い時間レンジ指定を排除することで高速化が可能となる。また、メタデータ機構を経由しないため、高速なデータアクセスが可能となる。

4.4.8　マルチテナント化に向けた課題

本提案での分散 Hayabusa は、高速化を目指すためにリソース管理機構や明確なスケジューリング機構などいくつかの機能を実装していない。そのためエラーや例外が発生した場合に、リトライできないようなアーキテクチャとなっている。また、インフラのマルチテナント化を考えると分散 Hayabusa 環境を占有できない場合に、現状のクライアントから直にクラスタへとリクエストが渡る設計では、正常な利用はできない可能性がある。マルチテナント化を考えるためには、リクエスト投入部分でのリクエストの識別とキューイング、優先順位づけが必要となるが今後の課題とする。

4.4.9　評価手法定義の課題

本研究では、分散 Hayabusa と Amazon EMR との全文検索にかかる処理時間を比較対象とした。分散 Hayabusa はストレージと密結合に近い形で処理を行うが、EMR の場合は S3 を透過的に扱う部分で、ストレージアクセスに対して遅延が生じてしまう。また、Elasticsearch との比較を行う場合には全文検索部分に対する処理時間は計測可能だが、リクエストを投入する REST の処理にかかる時間とクライアントの距離によって処理時間が変化する問題やシャードの数によるレスポンス時間の変化など考慮するポイントが存在する。予備実験では、Elasticsearch の全文検索はクラスタの性能に依存するが、100 万件の syslog データに対して数ミリ秒で応答を返すものが多かった。しかしながら、REST API の応答にはその 100 倍ほどの時間がかかり、結果トータルの応答時間は 0.1 秒以上になる場合が多く、リクエスト回数を増やした場合に Elasticsearch のエンジンは高速に動作するが、REST API の処理時間を含むトータル処理時間は Hayabusa に劣る結果になる場合がある。エンジンの処理時間、クライアントとの応答時間、ストレージアクセス、それらを踏まえた応答合計時間を評価軸として、公正な評価指標を定め

る必要があるが今後の課題とする。

また、本実験では蓄積されたデータに対して検索を行ったが、蓄積と検索を同時に実施した場合の性能劣化に関して、各比較対象のシステムでどのような結果が出るかも評価手法定義の一つの課題となる。

4.5 まとめと今後の課題

4.5.1 まとめ

本研究では Hayabusa の分散処理の設計と実装を行った。計測した結果、1 台の処理ホストでは約 249 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 秒台でフルスキャンし、全文検索可能な分散 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使い勝手の良いツールとなり、対応時間を著しく短縮できる可能性がある。また、システムを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。

4.5.2 今後の課題

本研究では 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十分に行えておらず、正しく評価可能な比較指標を定めた上でさらなる比較実験を行い 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。

第5章　分散Hayabusaのさらなる効率化

5.1　分散Hayabusaの問題点

第 4 章で紹介した分散 Hayabusa はとても高速な検索結果を返す一方、各計算ノードへとデータを複製する仕様となっている。複製を行うことにより、計算ノードが保持するストレージは結果として 1 ノードであろうが 10 ノードであろうが均一にディスク容量が消費される。10 ノード分のストレージ容量をまとめて利用するような使い方はできない。分散ストレージを利用することで、本課題を解決することは可能であるが、分散ストレージ自体の問題（メタデータを経由することによるストレージアクセスの遅延やシステムの複雑性）が存在する。また分散ストレージ自体、トラブルが発生した場合に致命的な性能低下やストレージとしてのサービスが提供できなくなる危惧が存在する。分散ストレージ自身が大きな研究開発課題となるため、本研究ではその本質へは迫らないこととするがストレージ利用容量の拡大に関しては考慮する必要がある。

また、クライアントから計算ノードへと直にリクエストを送る実装により、現状処理にエラーが発生した場合にエラーを回復する機構は存在しない。それ以前にエラーが発生した場合の挙動は、レスポンス結果が都度違うなどユーザ自身が注力して利用や運用を行わないといけない実装となっている。本来であれば、どの計算ノードでどの様なリクエストが失敗したかを監視したのち、ユーザへと然るべきエラー通知を行うなどの仕組みが必要となる。

上記 2 点の問題を解決するために、新しく分散 Hayabusa の設計に以下を追加

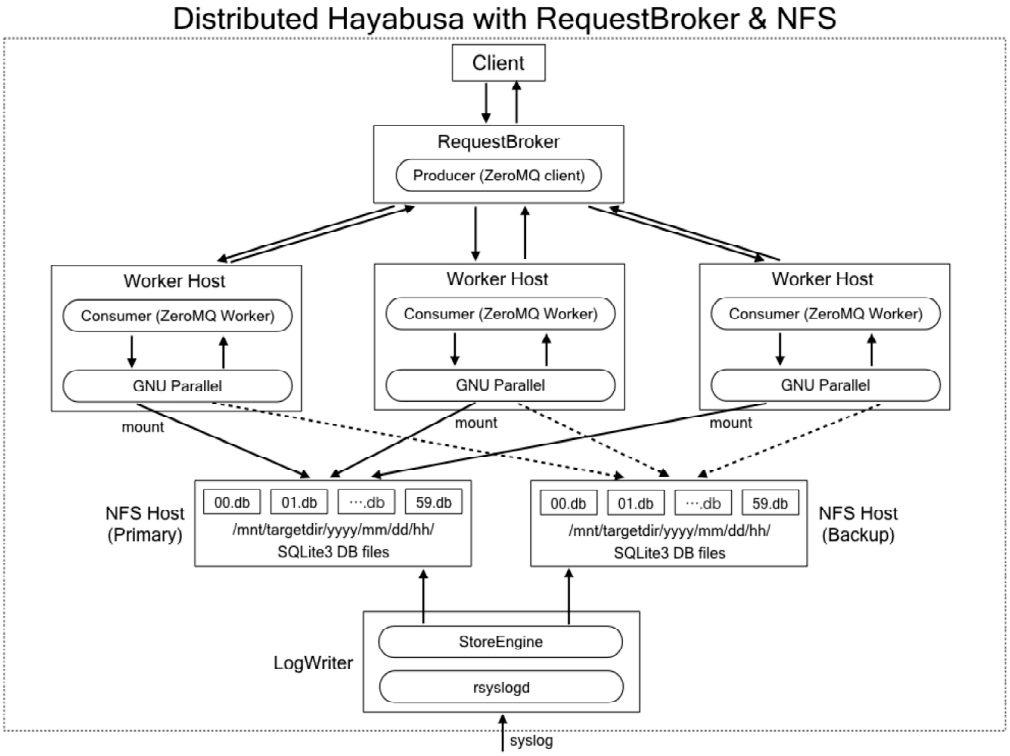

図 5.1: RequestBroker と NFS を用いた新アーキテクチャ

した。

- ネットワークストレージの利用
- リクエスト処理機構の実現

上記を実現する上で、分散 Hayabusa のアーキテクチャに変更が発生した。図 5.1 に、新アーキテクチャを定義する。

5.1.1 分散 Hayabusa の新アーキテクチャ

新しい分散 Hayabusa アーキテクチャでの主なシステムコンポーネントは以下となる。

- クライアント

- RequestBroker
- Worker
- NFS ホスト
- LogWriter

以下それぞれについて解説を行う。

クライアント

旧実装では、クライアントと Worker ノードは直に処理リクエストをやりとりする様に Producer/Consumer モデルを実現していた。新実装では、クライアントは処理リクエストを RequestBroker へと投入する手法へと変化した。クライアントは、REST API を用いて RequestBroker へと処理をリクエストし、結果を REST と ZeroMQ プロトコルを経由して受け取る。クライアントはリクエストした処理が、どの程度まで進んだかを逐次知ることが可能となり、リクエストが失敗した場合にも RequestBroker から通知を受ける。さらに失敗した理由に関しても、処理時間超過エラーの様に Worker ノードでどの様なエラーで処理が失敗したか情報を得ることができる。

RequestBroker

RequestBroker はクライアントから処理リクエストを受信し、各 Worker ノードへと Producer/Consumer モデルを用いて分配する。旧実装でクライアントが行っていた処理を RequestBroker へと移行した形であるが、更にリクエスト毎の処理状態を管理する。RequestBroker は Worker ノードへとどの処理を依頼したかを把握しているため、Worker のレスポンスをチェックすることで、全体の処理がどこまで進んだかやどの Worker の処理が異常終了や時間超過で終わらなかったかを管

理することができる。処理結果に関してはWorkerの全ての処理が終わったのち、結果をまとめてクライアントへとレスポンスを返す。それ以外にRequestBrokerはクライアントに対してWorkerの処理が完了するたびに通知を行い、クライアントは処理の進捗状況を知ることができる。Workerへの処理リクエストの割り振りに関しては、旧実装と同様にZeroMQのロードバランシングにより自動的に均一に分配されることになる。

Worker

Workerの動作は基本的に旧実装と同様にZeroMQを起点とし、GNU Parallelで並列にコマンド実行が行われる。旧実装との大きな違いは、Workerノード自身のストレージに処理対象のデータを保持しない部分となる。新実装では、ストレージにネットワークストレージであるNFSを用いることで、ストレージを外部へと設置した。これにより各Workerノードは外部のNFSサーバをマウントし、処理対象のデータを参照する必要がある。

NFSホスト

新実装でNFSホストは、二つのコンポーネントからアクセスされることになる。1つは複数のWorkerホストであり、2つ目は後述するLogWriterである。NFSホストには比較的大容量のストレージを用意することでこれまでWorkerノード単体の内部ディスク容量しか扱えなかった容量以上のデータを保持することができる。WorkerホストはNFSをReadOnlyでマウントし、参照のみ許可をする。NFSホストをPrimary, Backup構成で冗長化することで、データの冗長性を維持し耐障害性を実現する。LogWriterにはログデータを書き込むために書き込み権限を付与し、ネットワーク越しにログの書き込みを行う。

表 5.1: 新アーキテクチャ実験環境

EC2 インスタンス	m5.xlarge	m5.2xlarge
vCPU	x 4	x 8
メモリ	16GB	32GB
ディスク (EBS)	SSD 50GB (OS)	
OS	Ubuntu 16.04.4 LTS (Xenial Xerus)	

LogWriter

LogWriter の実態は、syslog を受信する rsyslogd と SQLite のデータベースファイルを書き込む StoreEngine の二つになる。共に旧実装と動作は変わりないが、データベースファイルの書き込み先が NFS となる。LogWriter は NFS ホストをリモートでマウントし、該当ディレクトリへとログデータを書き込む。NFS サーバは冗長構成で構築されるため、Primary, Backup の 2 台の NFS サーバをマウントし、データを同時に書き込むことになる。ログの書き込み先が NFS になる以外に旧実装と大きな違いはないが、機能的に 1 つの別コンポーネントとして設計を行った。

5.2 実験

本アーキテクチャの実験は、Amazon Web Service 上で行った。

5.2.1 実験環境

実験環境を表 5.1 に示す。各ホストの台数内訳は表 5.2 となる。NFS に関しては、上記のホストにさらに 4TB の EBS ストレージを追加してマウントして実験を行った。Worker のみコアスケール性能を確認するために、vCPU の数が異なる m5.xlarge および、m5.2xlarge インスタンスで性能試験を行った。他のホストに関しては m5.xlarge インスタンスで構成した。

表 5.2: 各ホスト数の内訳

ホストの役割	台数
client	1 台
RequestBroker	1 台
Worker	1 〜 10 台
NFS	1 台
LogWriter	1 台

実験は AWS 内のネットワークを占有して行うために、VPC(Virtual Private Network) を構築し、各インスタンス間は VPC 内のプライベートネットワークで接続した。NFS の転送性能を向上させるために、本実験ではネットワークの MTU を 9001 としジャンボフレームを利用可能としている。

5.2.2 実験の条件

測定コマンド

各種条件下で CLI Client を実行し、time コマンドを実行し実行時間を測定することで性能試験を行う。CLI Client コマンドにオプションを付け結果のカウント値、およびカウントの合算合計を求める。

ログデータベースファイル書き込み試験

NFS の書き込み負荷計測を行うために、ログファイル生成ツールを使用して様々な書き込み速度でログファイルを生成しながら LogWriter を実行し、NFS 領域にログデータベースファイルを書き込む試験を行う。

ログデータベースファイル読み込み試験

次にシステムの検索性能を測定する。1k レコード/sec (1,000 レコード/sec) の書き込み速度を想定して 1 分間分のログファイルを生成し、そのログファイルか

らデータベースファイルを生成する。そのデータベースファイルを多数複製することで 10 日分のデータベースファイル群を生成する。CLI Client から 10 日分のデータベースファイルを検索する特定のリクエストを複数回実行することで、NFS 領域からデータベースファイル読み込みむ本システムの性能を測定する。システムのスケールアウト性能を測定するために、Worker ノードは 1〜10 台まで増加させ、各 Worker ノード内の Worker プロセスは 1〜4 プロセスへと変化させて上記の測定を行う。

ログデータベースファイル書き込みと読み込み試験（同時実行）

1k、10k、100k レコード/sec の書き込み速度を想定した 1 分間分のログファイルを配置して LogWriter を動作させ、NFS 領域にデータベースファイルを毎分書き込む試験を行いつつ、並行して検索を行い読み込み試験を行う。また、NFS 領域にデータベースファイルを常時書き込む試験を行いつつ、同様に検索を実行して読み込み試験を行う。

5.3 結果

5.3.1 データベースファイル書き込み試験結果

書き込み試験では、以下の手順により試験を実行する。

1. apache-loggen コマンドによる擬似的な syslog の生成
2. apache-loggen のローテーション機能で syslog.1 ファイルを生成
3. store_engine コマンドで NFS 領域にデータベースファイルを書き込み
4. データベースファイルの作成時間を確認

表 5.3: NFS への書き込み性能結果

書き込み速度 (record/sec)	書き込み結果　ツール	
1k	1 秒未満	apache-loggen
10k	約 6 秒	apache-loggen
100k	約 50 秒	loggen
150k	60 秒以上	loggen

apache-loggen[4] は、OSS として公開される Web サーバである Apache のログを擬似的に生成するコマンドである。秒間にある程度の量のログを生成することは可能であるが、100k[record/sec] は生成不可能なため、100k[record/sec] 以上のログ生成に関しては、syslog-ng[23] のパッケージに付属するログ生成ツールである loggen コマンドを利用した。loggen を利用する場合には、UDP でログを生成して rsyslog で syslog を受信してファイルへと書き込み、loglotate デーモンのローテーション機能で syslog.1 を生成した。

書き込み結果を表 5.3 へと記載する。syslog から SQLite3 へのファイル変換は、cron により 1 分毎に実行されるため、100k[record/sec] が NFS を用いた運用を行う上での限界値と考えられる。ただし、バーストトラヒックが発生した場合には当然 syslog の量も増えるため、一概に限界値とは考えずにキャパシティを考慮した設計が必要になる。

5.3.2　データベースファイル読み込み試験

データベースの読み込み試験は以下の条件で行った。

- インスタンスタイプ：m5.xlarge、m5.2xlarge
- Worker ホスト数：1〜10 台
- Worker プロセス数：1〜4 プロセス
- 測定コマンドを 15 回実行し実行時間を計測

測定を行う実行コマンドは以下とする。

```
$ time -p /opt/hayabusa/bin/cli_client.py --start-time '2018-6-1'
--end-time '2018-6-10' --match 'Googlebot' -c -s
```

コマンドの各パラメータの意味は、「–start-time」と「–end-time」で検索の範囲を指定し（この場合は 10 日分のデータが対象）、syslog に含まれるキーワード「Googlebot」を検索し、「-c(ount)」オプションで各対象ファイル毎のカウント値を集計し、全てのカウント値を「-s(um)」引数によりカウント値を足し込み合計値を得る。

表 5.4 の結果より、m5.xlarge から m5.2xlarge に VM のスペックを上げることにより、全般的にパフォーマンスが向上することがわかる。ホスト数が多い場合、プロセス数を増やすと分散されるホストに偏りが生じ、ホスト分散による並列処理のパフォーマンスが十分に出ないことが問題点として見つかった。本来であれば、Worker プロセス数が増えることにより性能が伸びることが期待されていたが、結果として Worker が増えると性能が伸び悩むという現象が観測された。

表 5.4: 検索性能測定 (秒)

ホスト数	Worker プロセス数							
	1		2		3		4	
	m5.x	m5.2x	m5.x	m5.2x	m5.x	m5.2x	m5.x	m5.2x
1	40.06	32.87	34.85	21.36	33.37	19.96	32.29	18.66
2	20.38	16.64	20.35	11.66	18.61	10.59	19.97	11.29
3	16.24	13.24	13.83	8.35	13.85	8.79	13.30	7.45
4	12.54	10.02	13.76	7.73	10.22	5.98	13.22	7.50
5	8.59	6.81	7.39	5.27	10.05	6.19	13.40	7.49
6	8.65	6.84	7.20	5.28	10.04	6.26	13.28	7.53
7	8.71	7.05	7.41	5.46	10.10	6.38	13.31	7.49
8	8.57	7.47	7.20	5.39	10.03	6.45	13.17	7.49
9	8.60	7.67	7.23	5.50	10.02	6.23	13.23	7.50
10	5.32	5.28	7.32	5.28	10.01	6.17	13.16	7.51

表 5.5: 性能測定 [record/sec]

パターン	書込み無し	書込み有り		
		1k[r/s]	10k[r/s]	100k[r/s]
1 worker 1 プロセス	32.84	32.89	32.88	36.07
10 worker 4 プロセス	7.53	7.52	7.49	7.91

5.3.3 データベースファイル書き込みと読み込み試験（同時実行）

本試験では、NFS へのデータベースファイル書き込みと検索を同時に行い、以下を条件とする。

- ログファイルの想定書き込み速度：1k, 10k, 100k[recoeds/sec]
- StoreEngine 動作を想定し、cron で毎分データベースファイルを NFS の領域内に生成

検索の Worker ノードとして、AWS のインスタンスタイプとして m5.2xlarge を利用した。試験の環境として、

- Worker ノード 1 台、Worker プロセス 1 つ
- Worker ノード 10 台、Worker プロセス 4 つ

の 2 パターンを試した。

検索を実行するコマンドは、前説で利用したものと同様に以下とし、NFS へのデータベースファイルの書き込みを行いながら 30 回コマンドの実行を行う。

```
$ time -p /opt/hayabusa/bin/cli_client.py --start-time '2018-6-1'
--end-time '2018-6-10' --match 'Googlebot' -c -s
```

結果は表 5.5 となる。

書込み速度が 1k[record/sec]、および 10k[record/sec] の書き込みを行いながら検索をする場合には、書き込みを行っていない状態と比較しても検索への影響はほ

ぼ無い。100k[record/sec] の書き込みを行う場合に、書き込み無しと比較して、若干の検索遅延が見受けられた。1 Worker ホストで 1 Worker プロセスの場合では、100k の場合に 4 秒弱の遅延が発生しているが、10 Worker ノード 4 Worker プロセスの場合では約 0.4 秒の差となり、誤差の範囲と言える。パターンの差異は純粋にスケールアウトした結果、検索性能が向上したものと考えられる。

本実験から、実環境を想定した書き込み速度 (秒間 1k, 100k, 100k の syslog を受信した場合) では、検索性能に顕著な性能低下は見受けられず、十分高速に検索機能が動作するという結果が得られた。

5.4 新アーキテクチャに関する考察

5.4.1 Worker プロセスのロードバランシング問題

実験結果より、Worker プロセスのロードバランシングにおいてホストの分散に偏りが生じることが観測されたため、RequestBroker と Worker での Procedure/-Consumer の実装に変更を行った。具体的には、RequestBroker には各 Woker ノード内の各 Worker プロセスが此処に ZeroMQ の PULL 接続を行っていた部分に問題が見受けられたので、各 Worker ノード内のメインプロセスのみが ReuqestBroker への接続を行い一旦 Request を取得し、さらに Worker ノード内でローカル接続を用いたロードバランシングを行う設計へと変更した。図 5.2 で示すように、これにより RequestBroker から各 Worker ノードへとまずリクエストが均等に分散し、Worker ノード内でも、各 Worker プロセス群へリクエストが均一に分散するという多段の接続構成を形成することでロードバランシングの改善を行った。

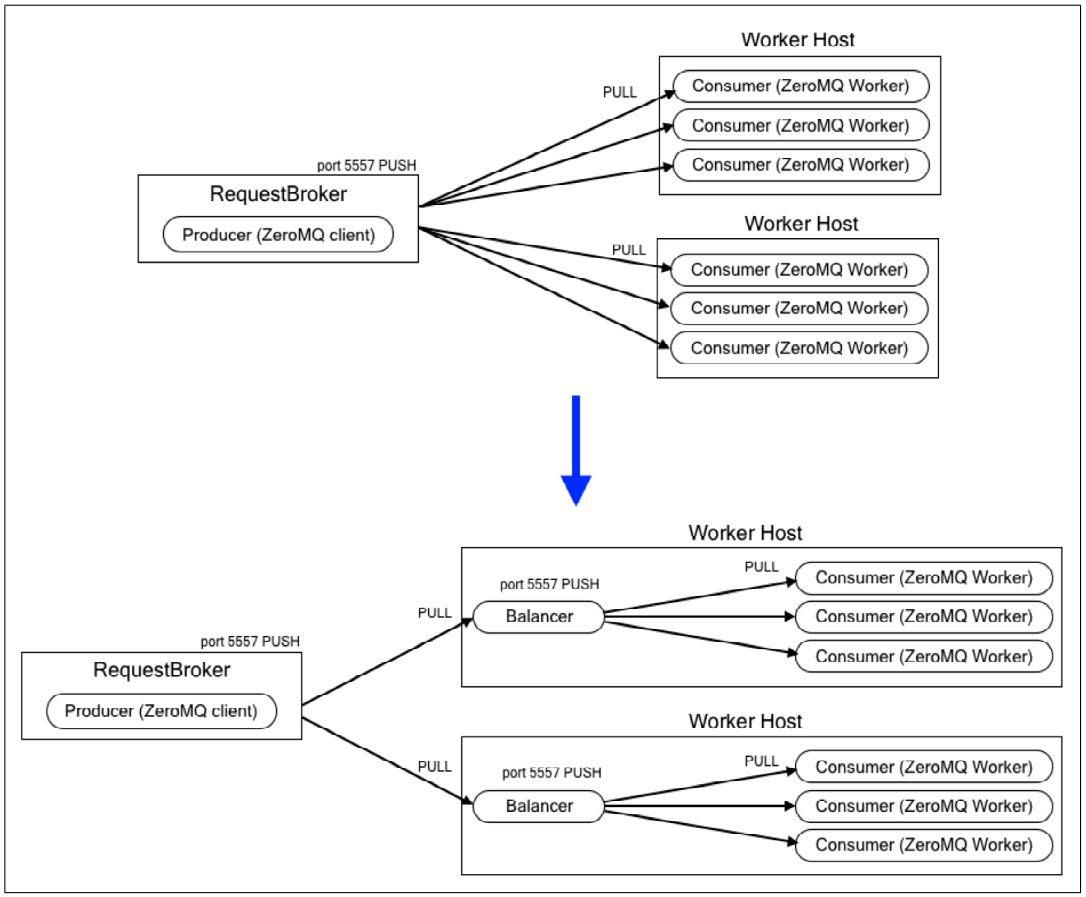

図 5.2: リクエストロードバランスの修正

改善結果を、AWS の m5.2xlarge インスタンスを用い測定した。10 Worker ノード 4 Worker プロセスで測定した結果は、表 5.6 となる。

表 5.6: 検索性能測定 (秒)

ホスト数	Worker プロセス数							
	1		2		3		4	
	改善前	改善後	改善前	改善後	改善前	改善後	改善前	改善後
10	5.28	5.35	5.28	5.34	6.17	5.39	7.51	5.30

ロードバランス改良後は、Worker ノード 10 台で Worker プロセスを増加させてもパフォーマンス低下が発生せずに、リクエストの分散がホスト単位で偏る問題は発生しなくなった。

5.4.2 ネットワークストレージ

NFS は比較的古くから使われ続けている実装であり、バージョンも上がり改良が継続されていて現在は version 4 がリリースされている。NFS を用いることで、単体ノードへと組み込むストレージと比較して容量が大きめのストレージを用意することが可能となり、旧実装で行った各処理ホストが保持するストレージ量しか処理ができない問題が解決できる。結果として、複製を行っていた分のデータを複製せず済み、複製により発生していたネットワーク帯域の圧迫は無視することができる。NFS は長年運用されてきた知見が多数存在し、パフォーマンス向上に関するチューニングを行いやすい。例えば、ネットワーク機器が設定可能であればジャンボフレームを利用し、MTU サイズを 9000 へと設定し、転送するデータ量を高速化することなどが可能である。実験を行った AWS の EC2 環境では MTU が 9001 に設定されており、デフォルトでジャンボフレームが流れる仕様となっていた。

もちろん欠点も存在する。ストレージへのアクセスがネットワーク経由になることにより、ディスクアクセスのオーバヘッドが発生する。ネットワーク障害が発生した場合には、データへのアクセス自身が行えなくなる可能性がある。また、旧実装では、1 台の Worker ノードにトラブルが発生し利用できなくなったとしても、他のノードへ複製データがあるためそちらのデータから処理が復元可能となる。NFS は Primary/Backup 構成で運用されるために、NFS サーバに障害発生時に Worker ホストのマウント先切り替えの時間が発生してしまう。

実験では、ディスクのアクセス速度が、旧実装の様に Worker ノード自身が抱えるディスクへとアクセスした場合と比較して 1/10 程度の速度となった。新実装では、ディスクへのアクセス速度が遅延してでも NFS を利用したストレージ量の増加とデータの集中管理が行えると想定して設計を行った。1/10 のディスクアクセス速度と引き換えに、管理の容易性と大規模なディスク容量を得られるというトレードオフとして成り立つ戦略である。また、全ホストへとデータを複製せずに

Primary/Backup の 2 ホストにのみデータを複製することで、データ複製のためのトラヒックも減り、かつデータの冗長性も担保できる。

5.4.3 RequestBroker

RequestBroker を実現したことにより、どの Worker ノードに処理が割り振られたか追うことが可能となり、クライアントは詳細な処理情報やエラー情報を取得することができる。これによりクライアントは、RequestBroker のみ処理対象として意識すればよく、クライアント実装は容易になる。クライアントにとってバックエンドはブラックボックスでも構わないが、リクエストに関して必要な情報（処理状況やエラー情報）を適切に、ReuqestBroker から取得可能となる。

本来であれば、RequestBroker にはエラーが発生した場合のリトライ処理まで含めるべきであるが、本実装では Worker ノードで発生したエラーの把握までとなった。エラーのリトライ処理に関しては、今後の課題とする。

5.4.4 クライアントの受信データ量

クライアントは「-c(count)」や「-s(um)」オプションを利用しない限り、検索にマッチした全ての syslog を戻り値として受信する。現実装では、RequestBroker が結果をまとめ、クライアントへと返す役割を担う。Haybusa が扱うデータは巨大になることが多いため、マッチした結果が巨大であればあるほど、此処での戻り値も大きなものになってしまう。

現状の参考値として測定した戻り値の数とサイズ、それを標準出力へと表示するのにかかった時間とテキストファイルへとリダイレクトした場合の参考値は表 5.7、表 5.8、表 5.9 となる。

表 5.7: クライアント検索結果その 1

対象ログ	1k[r/s] x 10 日分
ログ総数	1,440 万行
キーワード	'24.129.72.64 Electronics'
ヒット数	約 20 万行
サイズ	約 48MB
標準出力への表示時間	5m 7.77s
リダイレクト時間	5m 9.81s

表 5.8: クライアント検索結果その 2

対象ログ	1k[r/s] x 1 日分
ログ総数	1,440 万行
キーワード	'Googlebot'
ヒット数	約 489 万行
サイズ	約 863MB
標準出力への表示時間	10m 31.05s
リダイレクト時間	3m 18.89s

表 5.9: クライアント検索結果その 3

対象ログ	1k[r/s] x 1 日分
ログ総数	1,440 万行
キーワード	'Firefox'
ヒット数	約 2300 万行
サイズ	N/A
標準出力への表示時間	N/A
リダイレクト時間	N/A

検索ワードに「Firefox」を使った場合の結果は、表 5.9 となる。N/A の部分は、RequestBroker のプロセスが OOM Killer により kill されたため測定不能であった。RequestBroker は結果をオンメモリ処理で集計していたため、物理メモリの限界値がきたと推測される。

本結果より、クライアントへと戻り値を返す場合には RequestBroker で Worker の処理結果をまとめるのではなく、都度 Worker からの結果をクライアントへと部

分的に送信する仕組みが必要であると認識した。その場合には結果表示の順序問題が発生するが、クライアント側でのリオーダーや WebUI での表示に工夫を入れることでユーザビリティを損なわずに実現可能である。

5.5 OSS 化

本実装は PoC 実装済みで OSS(Open Source Software) として、GitHub のリポジトリ [8] で公開済み（図 5.3）である。

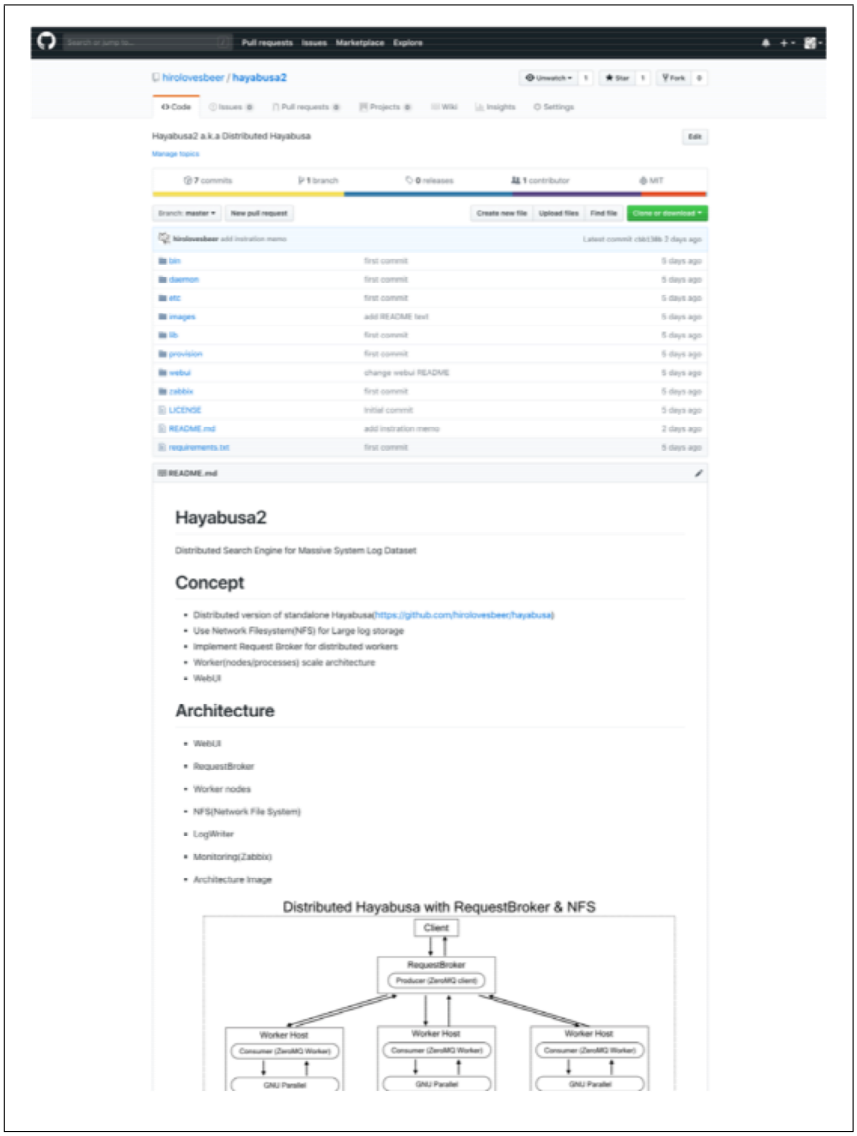

図 5.3: Github で公開済みのリポジトリ

OSS 実装では、Ansible[2] を用いたプロビジョニング、Gentelella[11] を用いた WebUI の提供、さらには Zabbix[26] を用いたシステム全体の監視と、システムログの収集機構を備えたパッケージとして分散 Hayabusa を利用することができる。

Ansible を用いることで、クラウド環境やオンプレミス環境上に自動的に分散 Hayabusa の環境を手軽に構築することが可能となった。また、WebUI を提供することで、複雑なオペレーションなしにユーザが即座に分散 Hayabusa を体験できる様な環境となっている。

WebUI は、図 5.4 に示す様にカレンダーから時間を指定し、図 5.5 の様に範囲を指定するし、検索キーワードを入力することで検索実行が可能となる。

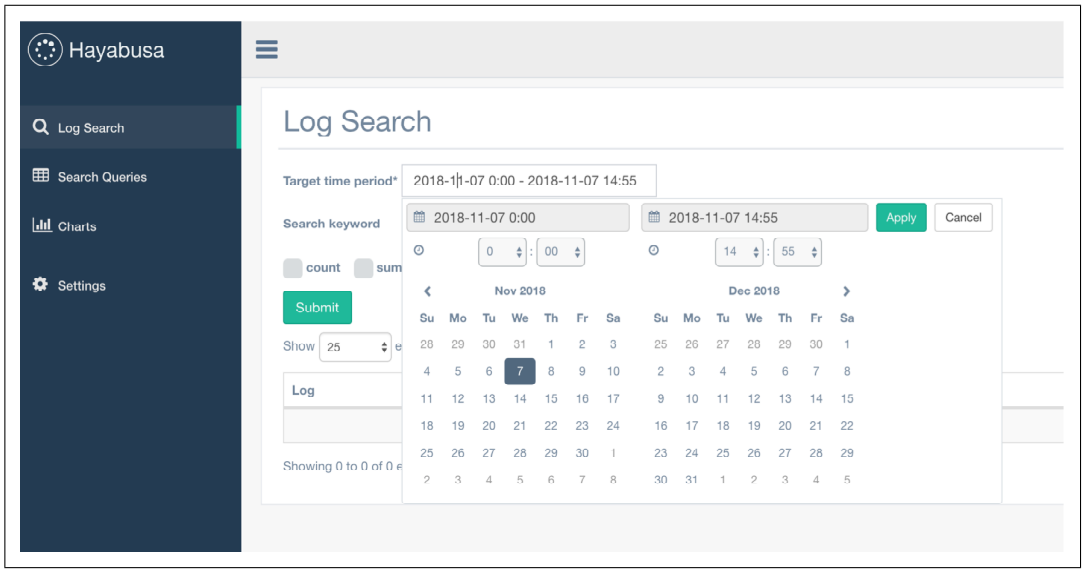

図 5.4: カレンダーから検索範囲を指定

検索リクエストにはそれぞれリクエスト ID が振られ、リクエストの進捗状況が「completed/running/waiting」の様なステータスで表示される。

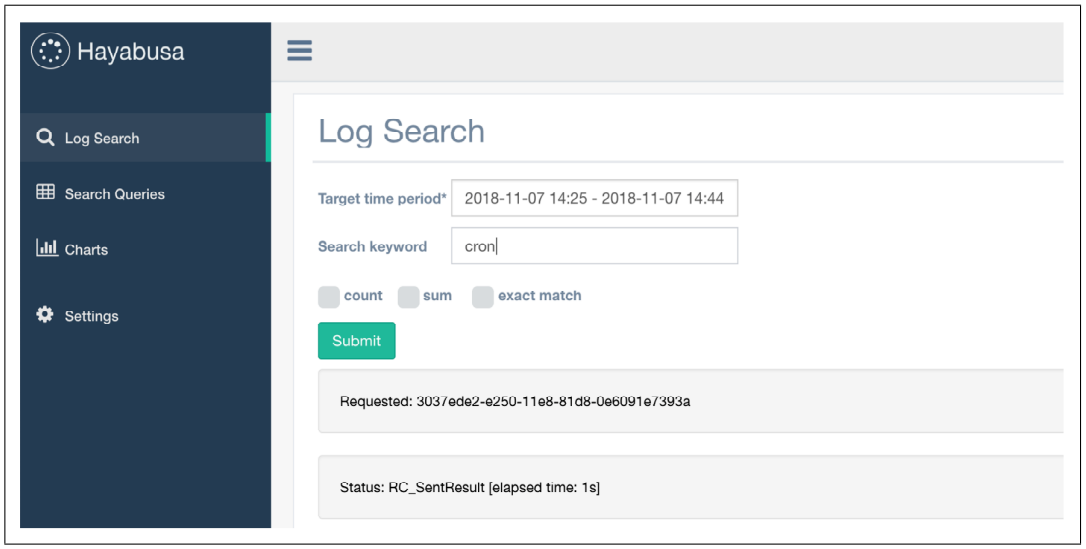

図 5.5: 検索文字列を指定

検索結果は図 5.6 の様に表示される。結果を 1 ページ内に表示する行数は変更可能であり、現在の表示中の行や、複数行にまたがる場合のページ移動等も可能となる。

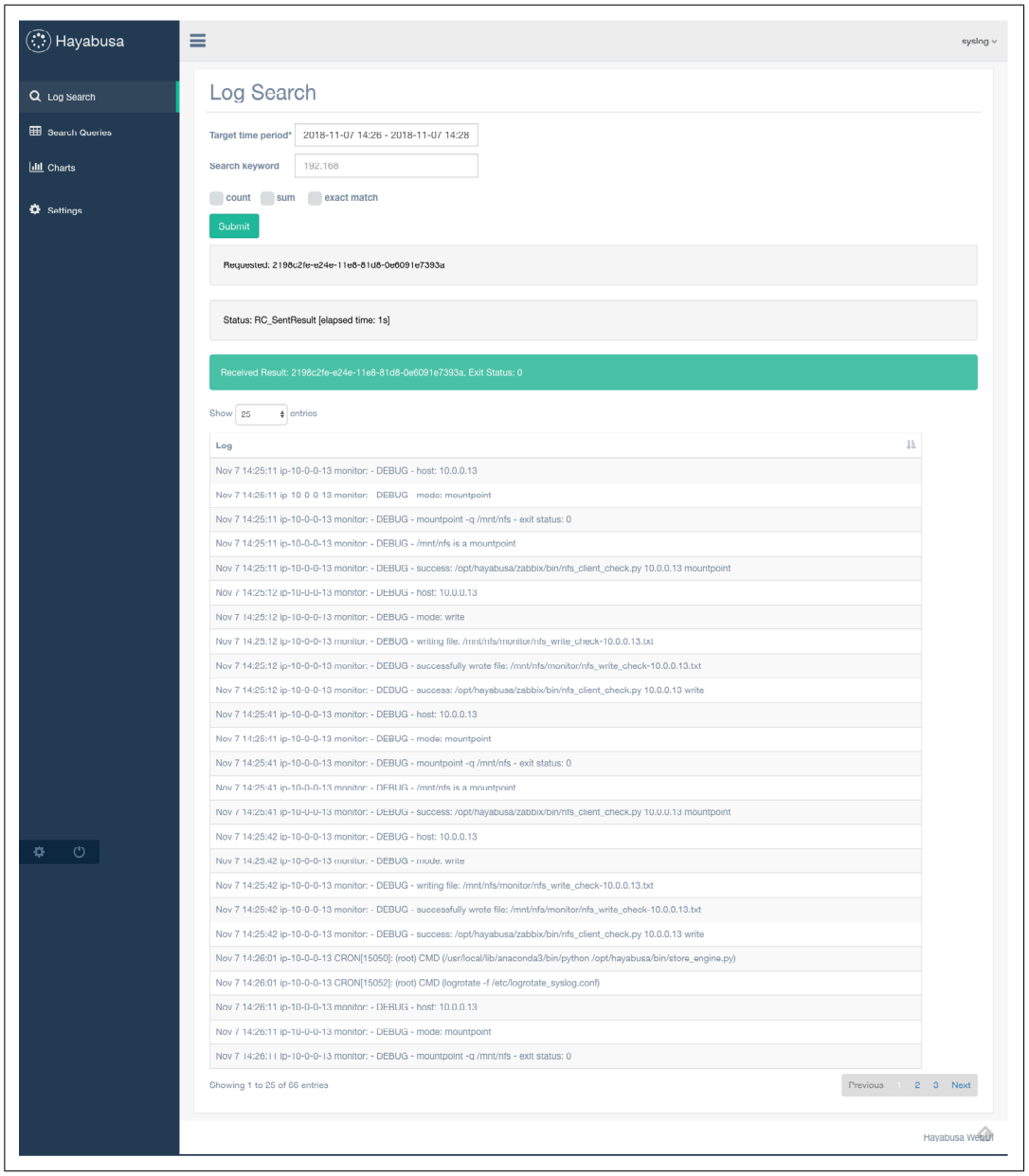
Hayabusa
syslng
Log Search
Search Queries
Charts
Settings
Log Search
Target time period*
2018-11-07 14:26 - 2018-11-07 14:28
Search keyword
192.168
count
sum
exact match
Submit
Requested: 2198c2fe-e24e-11e8-81d8-0e6091e7393a
Status: RC_SentResult [elapsed time: 1s]
Received Result: 2198c2fe-e24e-11e8-81d8-0e6091e7393a, Exit Status: 0
Show 25 entries
Log
Nov 7 14:25:11 ip-10-0-0-13 monitor: - DEBUG - host: 10.0.0.13
Nov 7 14:25:11 ip-10-0-0-13 monitor: - DEBUG - mode: mountpoint
Nov 7 14:25:11 ip-10-0-0-13 monitor: - DEBUG - mountpoint -q /mnt/nfs - exit status: 0
Nov 7 14:25:11 ip-10-0-0-13 monitor: - DEBUG - /mnt/nfs is a mountpoint
Nov 7 14:25:11 ip-10-0-0-13 monitor: - DEBUG - success: /opt/hayabusa/zabbix/bin/nfs_client_check.py 10.0.0.13 mountpoint
Nov 7 14:25:12 ip-10-0-0-13 monitor: - DEBUG - host: 10.0.0.13
Nov 7 14:25:12 ip-10-0-0-13 monitor: - DEBUG - mode: write
Nov 7 14:25:12 ip-10-0-0-13 monitor: - DEBUG - writing file: /mnt/nfs/monitor/nfs_write_check-10.0.0.13.txt
Nov 7 14:25:12 ip-10-0-0-13 monitor: - DEBUG - successfully wrote file: /mnt/nfs/monitor/nfs_write_check-10.0.0.13.txt
Nov 7 14:25:12 ip-10-0-0-13 monitor: - DEBUG - success: /opt/hayabusa/zabbix/bin/nfs_client_check.py 10.0.0.13 write
Nov 7 14:25:41 ip-10-0-0-13 monitor: - DEBUG - host: 10.0.0.13
Nov 7 14:25:41 ip-10-0-0-13 monitor: - DEBUG - mode: mountpoint
Nov 7 14:25:41 ip-10-0-0-13 monitor: - DEBUG - mountpoint -q /mnt/nfs - exit status: 0
Nov 7 14:25:41 ip-10-0-0-13 monitor: - DEBUG - /mnt/nfs is a mountpoint
Nov 7 14:25:41 ip-10-0-0-13 monitor: - DEBUG - success: /opt/hayabusa/zabbix/bin/nfs_client_check.py 10.0.0.13 mountpoint
Nov 7 14:25:42 ip-10-0-0-13 monitor: - DEBUG - host: 10.0.0.13
Nov 7 14:25:42 ip-10-0-0-13 monitor: - DEBUG - mode: write
Nov 7 14:25:42 ip-10-0-0-13 monitor: - DEBUG - writing file: /mnt/nfs/monitor/nfs_write_check-10.0.0.13.txt
Nov 7 14:25:42 ip-10-0-0-13 monitor: - DEBUG - successfully wrote file: /mnt/nfs/monitor/nfs_write_check-10.0.0.13.txt
Nov 7 14:25:42 ip-10-0-0-13 monitor: - DEBUG - success: /opt/hayabusa/zabbix/bin/nfs_client_check.py 10.0.0.13 write
Nov 7 14:26:01 ip-10-0-0-13 CRON[15050]: (root) CMD (/usr/local/lib/anaconda3/bin/python /opt/hayabusa/bin/store_engine.py)
Nov 7 14:26:01 ip-10-0-0-13 CRON[15052]: (root) CMD (logrotate -f /etc/logrotate_syslog.conf)
Nov 7 14:26:11 ip-10-0-0-13 monitor: - DEBUG - host: 10.0.0.13
Nov 7 14:26:11 ip-10-0-0-13 monitor: - DEBUG - mode: mountpoint
Nov 7 14:26:11 ip-10-0-0-13 monitor: - DEBUG - mountpoint -q /mnt/nfs - exit status: 0
Showing 1 to 25 of 66 entries
Previous 1 2 3 Next
Hayabusa Webui

図 5.6: 検索結果表示

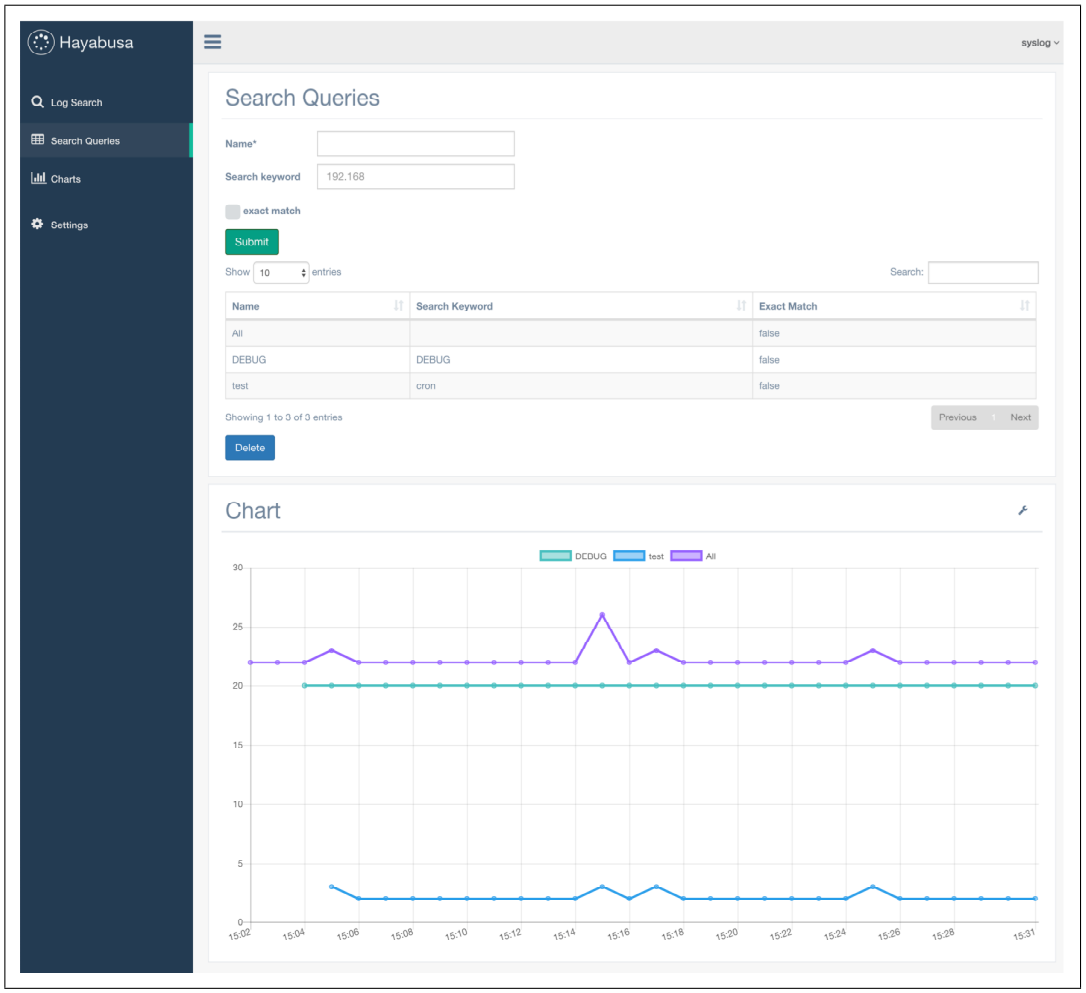

図 5.7: サーチクエリの動的結果表示

また、特定ログを 1 分間隔に検索・集計をして動的に更新表示可能な、Search Query 機能も図 5.7 の様な画面で実装している。これにより特定のログの異常検出をダッシュボードで監視可能となる。

5.6 まとめ

本節では、分散 Hayabusa で実現できていなかった

- ネットワークストレージの利用
- リクエスト処理機構の実現

の実装ならびに評価を行った。

ネットワークストレージを用いることで、分散 Hayabusa で並列にデータを複製し、保持していた部分を集中的にストレージを管理する NFS を利用することで、複製を無くしたストレージの集約と大容量化を実現した。NFS 利用に伴うネットワーク越しの処理データへのアクセスに対する処理遅延は、システム設計のトレードオフと考える。

また、RequestBroker を実現することで、クライアントが直に Worker を呼び出す機構から、クライアントからリクエストを受け Worker の処理状態やエラーを保持し、結果をまとめてクライアントへと返す機構が実現した。これによりユーザは投入したリクエストの状況の把握やエラー通知など、投入した処理に対する詳細な情報を得ることができる。そして WebUI や専用クライアントを用いることで、ユーザビリティも向上し利用しやすい環境となった。

本実装の分散 Hayabusa は OSS として公開されている。これにより多数のユーザに利用してもらうことが可能となり、かつ不具合の修正や機能向上のための改修などはコミュニティベースで行うことができ、必然的にソフトウェアとして成熟していくことが期待できる。

クライアントに大きな戻り値が返された場合など、新実装で見つかった問題点に関しては、今後の課題として OSS ベースに改良を進めることとする。

第6章　計算が軽量な異常検知手法の提案

6.1　はじめに

6.1.1　背景と目的

本研究では、多数のマルチベンダ機器が混在する大規模なイベントネットワークにおいて、運用者にとって未知のログが多数出現する過酷な環境下であってもログの総量から異常を読み取り、運用者へと効率的に通知可能なアルゴリズム適用の提案を行う [53]。提案手法では株式市場で用いられるテクニカル分析手法を採用し、正規分布に基づく確率理論からログ総量の突発的な上昇や下降を検知する。本手法を用いることでアラートの通知頻度を減少させ、運用者への現実的な異常発生回数の通知が可能となり、トラブルへの初動対応の高速化が行える。

マルチベンダ機器が投入される大規模なイベントネットワークの一例として、Interop Tokyo で構築される ShowNet で収集された syslog を用いて実データをもとに異常状態の分析を行う。

6.1.2　ShowNet における syslog 監視

ShowNet では、監視システムの 1 つとして syslog を収集し分析するシステムが運用される。数年前までは syslog の可視化は行われずに ShowNet 運用が行われていたため、ネットワーク異常が発生した際には、運用メンバーが ping/traceroute/tcpdump などのツールを用いてがトラブルの切り分けを行っていた。ログの可視化を

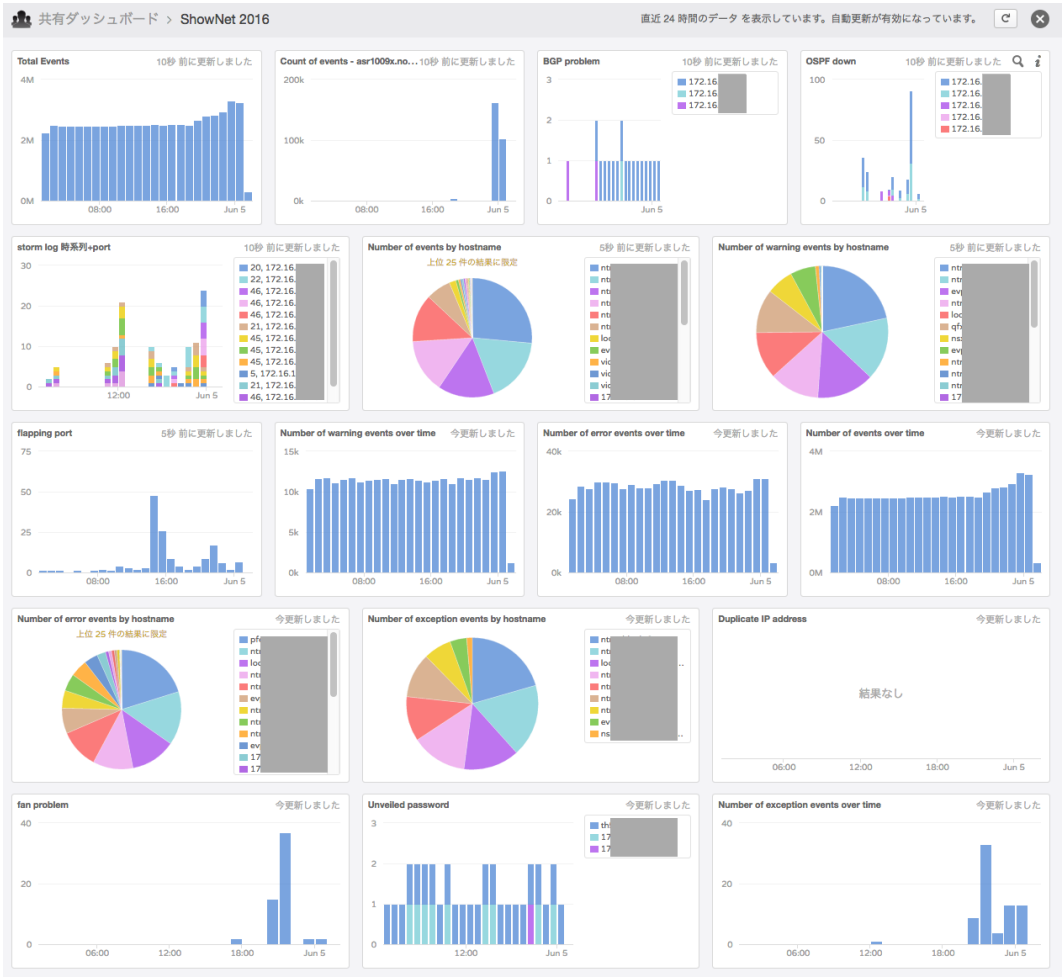

図 6.1: VMware vRealize LogInsight を使った syslog 監視例

行うことで特定時間内に発生したログからキーワードを抽出し、閾値を超えた場合に運用者にアラートを通知することができる。例として図 6.1 に VMware vRealize LogInsight [25]（以下、LogInsight）を使用した syslog 監視をあげる。LogInsight の機能として,

- ダッシュボードによる条件抽出したログの可視化
- 特定キーワード（OSPF down/BGP down/Storm detection など）の出現監視
- 特定キーワードのマッチング回数監視（閾値ベース）

などが提供される。キーワードマッチングをトリガーとしてトラブルの原因究明

を行う訳だが、マルチベンダの機材が多いために運用者は未知のログを扱うことが多く、どのようなキーワードがトラブルの原因になるかを瞬時に判別することは難しい。

ログの意味解析を行い、スコアリングに基づく異常値解析を行うことは可能ではあるが、キーワードの出現頻度だけでは、そのログが正常か異常かの判断を行うことは運用者にとって難しい。機械学習を用いて、教師データを精査し精度を上げ異常検知を行うことも可能ではあるが、イベントネットワークの特徴として開催期間が短すぎネットワーク安定状態における学習データを集めることが難しい。また ShowNet の特性として実験ネットワークの意味合いも強く、debug メッセージや必要のない info メッセージが大量にログ出力され、機械学習を行う上ではノイズとなるデータが多すぎる。

そこで本研究では、ShowNet で集められるマルチベンダ機器から出力される膨大な syslog データの総量を移動平均と標準偏差を用い集計比較することで、計算量が少なく軽量に計算可能なアルゴリズムを利用し、運用者へ異常検知のトリガーとなる事象を通知する手法を提案する。

6.2 提案手法

6.2.1 提案概要

本研究では、計算が軽量なアルゴリズムとして正規分布に基づいたアルゴリズムを採用する。短期間で構築されるために安定状態を定義しづらいイベントネットワークの特徴を再現するため、ShowNet で収集された syslog を対象とすることで、大規模なイベントネットワークの異常を検知する。

ShowNet では以下のようなログの状態が発生した場合に異常状態であると想定している。

- 機材への攻撃によるログの急増

- 機材の不具合によるログの急増
- 機材の不調によるログの急増
- 機材の設定ミスによるログの急増
- 大量な正常アクセスによるログの急増
- ウィルスやワームが発生することによるログの急増

本提案において syslog 総量急増の検出は移動平均と標準偏差を用いる。具体的なアルゴリズムとしてボリンジャーバンド [30] を用い、ログ総量から異常を検知する。また移動平均アルゴリズムの評価として、単純移動平均と指数移動平均を用い両者の比較を行う。

6.2.2 ボリンジャーバンド

ボリンジャーバンドは株式の取引で利用されるテクニカル分析手法の 1 つで 1980 年代前半に John Bollinger により公表された。移動平均を示す線と、その上下に値動きの幅を示す線を加えた指標のことをいい、価格の大半がこのバンドの中に収まるという統計学を応用したテクニカル分析の一つである。

ボリンジャーバンドは正規分布に基づく理論である。統計学の正規分布理論では、図 6.2 で示すように、

- 平均値 $\pm\sigma$（標準偏差）に収まる確率は 68.26%
- 平均値 $\pm 2\sigma$（標準偏差）に収まる確率は 95.44%
- 平均値 $\pm 3\sigma$（標準偏差）に収まる確率は 99.73%。

となる。

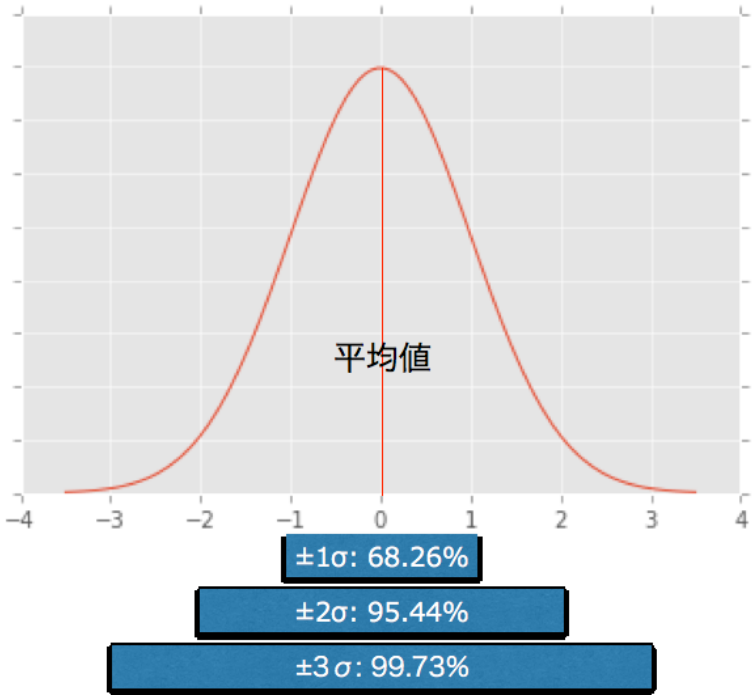

図 6.2: 正規分布と σ の確率

ボリンジャーバンドの考え方では、株価は 99.73%の確率で $\pm 3\sigma$ のバンド幅に収まると仮定され、株価が$+2\sigma$ のラインを超えた場合や$+3\sigma$ のラインにタッチした場合には株が買われ過ぎであると判断し、逆に-2σ のラインを株価が下げた場合や-3σ のラインにタッチした場合には株が売られ過ぎであると判断され、適切な価格へ戻ることが予想できる。

つまり、

- 移動平均 + 3σ（標準偏差）を UpperLimit（上限値）
- 移動平均 - 3σ（標準偏差）を LowerLimit（下限値）

として上限値と下限値を採用し、それらの値と株価の比較により適正価格かどうかの判定を行う。

移動平均（単純移動平均）は以下の式として表すことができる。

$$\overline{x} \quad = \quad \frac{1}{n}\sum_{i=0}^{n-1} x_i$$

新しい値を移動平均の計算に加えたい場合には、現在の移動平均の値に対し、新しい値を加え古い値を除くことで求めることができ、総和を求め直す必要はないので軽量な計算で済む。

また標準偏差（σ）は以下の式で求めるられる。

$$\begin{aligned}\sigma &= \sqrt{\frac{1}{n}\sum_{i=0}^{n-1}(x_i-\overline{x})^2} \\ &= \sqrt{\frac{1}{n^2}\left\{n\sum_{i=0}^{n-1}x_i^2-\left(\sum_{i=0}^{n-1}x_i\right)^2\right\}}\end{aligned}$$

標準偏差（σ）に ± 3 をかけた値を、UpperLimit, LowerLimit として用いる。

6.2.3 syslog 総量計算への応用

本提案では syslog の総量の推移に関して株式取引と同様に統計的な推移が存在すると仮定し、移動平均 $\pm 3\sigma$（標準偏差）を超えた場合、もしくは下回った場合には異常状態であると判定する。syslog の総量はシステムが安定していれば株式と同等に一定範囲（$\pm 3\sigma$ の間）で推移すると仮定すると、99.73%の確率で総量は UpperLimit と LowerLimit の間であるバンド内を推移ことを意味し、バンドの上限を超える/下限を下回る確率である 0.27%を異常値として検出する。

6.2.4 アラート発生率と誤検知

ネットワーク運用者は発生したアラートが適切なアラートであるか判断を行う必要がある。アラートが少なければ少ないほど個々の事象を調査する時間は増え、調査の精度を上げることができる。アラートが頻発した場合には、調査を行う人員の数に依存するが調査に費やされる総時間は増加する。故に、実時間で調査を

表 6.1: 実験環境

OS	CentOS 7.2
開発言語	Python 3.5.1
CPU	Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz
メモリ	128GB

完了させるために本当に通知したいアラートのみ通知するというオペレーションが運用上望ましい。

しかしながら、過度なアラート抑制は誤検知を招く恐れがある。本来通知されなければならないアラートが抑制されるフォールスネガティブが発生した場合に、ネットワーク運用者はアラートに気がつくことができない。逆に、本来通知される必要がないフォールスポジティブなアラートが通知された場合には、必要のない運用オペレーションが発生することになる。ボリンジャーバンドのアルゴリズムでは基準を超えたかどうかのみを判定するため、基準を超えた場合にはボリンジャーバンド的に常に正しいアラート通知となる。ボリンジャーバンドのアルゴリズムのみでは、通知されたアラートが誤検知か否かを判定することは難しい。逆にアラートの抑制を行う場合には、本来通知すべきアラートが通知されないためフォールスネガティブは発生する。

本研究で目標とする 0.27%が異常値として適切な値かどうかは、監視運用者の判断に任せられることになるが、アラートの発生回数が初期の気づきとして運用対応を行う現実的な回数内に収まれば運用に適用可能と仮定する。

6.3 評価

6.3.1 実験概要

実験環境を表 6.1 に示す。本実験では、ShowNet に接続された機材が出力した管理ログを収集して、ShowNet 終了後に実験環境において Python を用いて解析

を行った。解析対象の syslog の容量は約 6.4GB で、行数にして約 4,350 万件を対象とした。なお可読できないバイナリ形式で出力されたログに関してはノイズとして扱い除外してから解析を行った。

6.3.2 手法

本提案では、ログの意味分析を行わずに時間あたりのログ行数の出現回数を集計分析する手法を用いた. ログの総数を分析するにあたり、syslog から分析には必要ではない情報の削除を行った. syslog フォーマット [35] は大きく、ヘッダー部とメッセージ部からなる。ヘッダー部は、

- タイムスタンプ
- デバイス名

を必ず含み、タイムスタンプはローカル時刻でフォーマットは”Mmm dd hh:mm:ss”となる。また、デバイス名はホスト名または IP アドレスとして定義される。メッセージ部は、フリーフォーマットでテキストメッセージが出力される。

ログの総数から分析を行うために本提案では、タイムスタンプとデバイス名の 2 カラムを用いて分析を行った。計算速度を速めるため、メッセージ部の情報を削除したファイルをフィルタプログラムにより csv データとして作成した。事前処理を行った csv ファイルを、Python のデータ解析ライブラリである pandas[41] を用いてログの総量を時系列に解析した。読み込んだ csv データは pandas で定義される DataFrame 形式で処理される。DataFrame は、pandas で定義されるデータ構造の一つで、二次元のテーブルとしてデータを定義できる。各行、各列にはラベルをつけることができ、1 行を 1 つのデータとして表計算ソフトウェアの様に処理できる。DataFrame 形式で読み込んだデータに対して、基本的な算術計算をメソッドとして呼び出すことができる。本提案では、移動平均と標準偏差を扱うがこれら

```
1 import pandas as pd
2 df = pd.read_csv('./2016-06-06-syslog.log', delim_whitespace
     =True, ...)
3 count = df.groupby(pd.TimeGrouper('1Min')).count()
4 mean  = count.rolling(window=60).mean()
5 std   = count.rolling(window=60).std()
6 std_plus  = std.apply(lambda x: x * 3)
7 std_minus = std.apply(lambda x: x * -3)
8 upper_limit = mean.add(std_plus)
9 lower_limit = mean.add(std_minus)
```

図 6.3: サンプルコード

に関しても、DataFrame に対するメソッド呼び出し（移動平均: mean メソッド、標準偏差: std メソッド）を行うことで実装する。

処理を行う前提は以下とする。

- 1 日ごとのログ総量を計算/描画対象とする
- 1 分単位でデータをグルーピングする
- 過去 1 時間分のログ総量を移動平均に利用する（0 時台の計算には前日の 23 時のデータを読み込む）
- 1 分単位のグルーピングにより、移動平均/標準偏差のウィンドウサイズは 1 時間で 60 とする

総量、移動平均、標準偏差を求めるサンプルコードは図 6.3 となる。

各行の処理は以下を意味する。

1) pandas ライブラリの読み込み

2) ログファイル（csv ファイル）の読み込み

3) DataFrame のデータを 1 分単位でまとめて集計

4) mean メソッドで移動平均値を計算

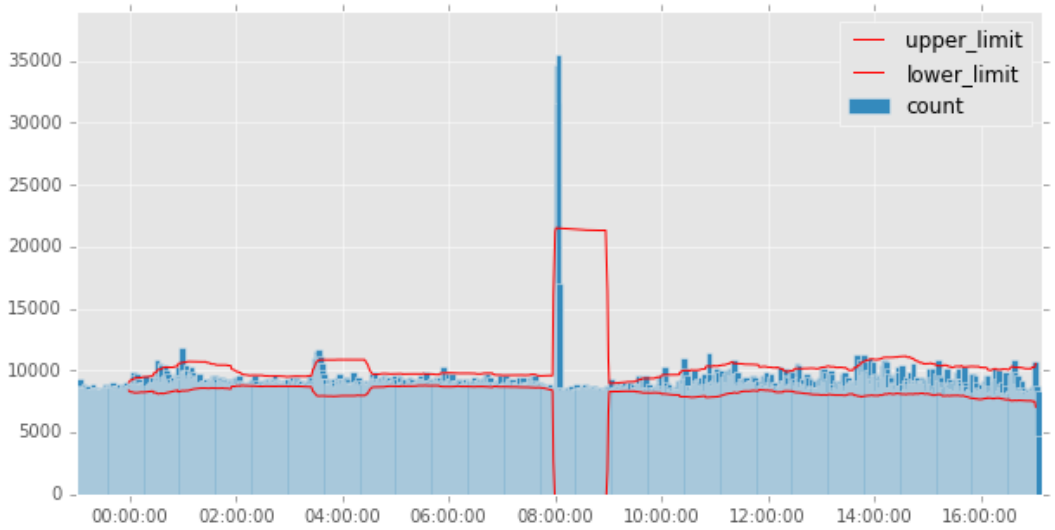

図 6.4: 1 日のログ総量分析例

5) std メソッドで標準偏差を計算

6) 標準偏差の 3 倍を計算（$+3\sigma$）

7) 標準偏差の-3 倍を計算（-3σ）

8) 移動平均と標準偏差の足しあわせ（UpperLimit）

9) 移動平均と標準偏差の足しあわせ（LowerLimit）

入力データの例外として、5/27 のみ初日ということで前日からの学習データは存在しない。

これらの計算データから画像を生成した例が図 6.4 となる。青い棒グラフである count が 1 分ごとにまとめられたログの総量を表す。総量の上方の赤い折れ線グラフが UpperLimit を意味し、下方の赤い折れ線グラフが LowerLimit を表す。ログの総量が UpperLimit と LowerLimit の範囲を推移する場合には、ログの総量は異常状態ではないと判断する。

本提案では異常値を見つける方法として、ボリンジャーバンドの上限のみにまずは着目した。つまり移動平均$+3\sigma$ を指す UpperLimit が総量の合計を超えた回数

表 6.2: アラートレベル

Level	異常値の範囲
Low	3σ 以上 4σ 未満
Middle	4σ 以上 5σ 未満
High	5σ 以上

を集計した。この時に UpperLimit を超える確率は、正規分布の$+3\sigma$ のみとなり約 0.14%(0.27%の半分) となる。UpperLimit にのみ着目する理由は、一般的に機器の異常やネットワークの異常時には syslog が大量に出力されるという運用者の経験則に基づく。また正常アクセスであっても、DoS（Denial of Service）のような攻撃を受けた場合にはアクセスログが大量に出力され、上限値超えに気がつくことで正常ではない状態であると判断できる.

さらに異常値判定の精度を考慮しアラート通知の頻発を抑えることを目的として、異常値が$+3\sigma$ を超えた値の範囲に着目し、アラートのレベル分けを行うこととした (表 6.2)。アラートレベルは動的に計算され、Low=3σ 以上 4σ 未満、Middle=4σ 以上 5σ 未満、High=5σ 以上という形で出力される。先行実験 [54] では Low, Middle, High を固定値の幅として計算していたが、本提案では動的にアラートレベルを計算する仕組みを取り入れた。また、移動平均のアルゴリズムの種類により異常値の検出がどのように変化するかの実験を行うため、単純移動平均と指数移動平均の 2 種類の移動平均アルゴリズムを用い、異常検知数がどのように変化するか比較実験を行った。

6.4 結果

6.4.1 アラート発生率

ShowNet の期間中に発生した syslog の総量とボリンジャーバンドの上限を超えたアラートの発生率を表 6.3 に示す。

表 6.3: アラート発生率

日付	ログ総数	時間スロット数	アラート回数	発生率
5/27	192	617	12	1.94%
5/28	181,285	1,440	46	3.19%
5/29	552,579	1,440	50	3.47%
5/30	821,363	1,440	48	3.33%
5/31	617,368	1,440	27	1.88%
6/1	917,368	1,440	47	3.26%
6/2	1,949,738	1,440	38	2.64%
6/3	1,771,956	1,440	28	1.94%
6/4	2,108,661	1,440	38	2.64%
6/5	3,177,122	1,440	24	1.67%
6/6	3,297,654	1,440	24	1.67%
6/7	2,702,382	1,440	24	1.67%
6/8	3,186,363	1,440	24	1.67%
6/9	12,769,834	1,440	24	1.67%
6/10	9,446,694	1,083	22	2.03%
合計	43,500,499	20,420	476	2.33%

表 6.3 は、

1. 日付
2. ログ総数
3. 時間スロット数
4. アラート回数
5. 発生率

の 5 項目からなる。ログ総数は、日付ごとに集計した syslog の総数を表す。時間スロット数は、 ログの集計単位 (1 分単位:60 秒) を 1 日 (86400 秒) で割った数 (86400/60=1,440) を表す。5/27 の時間スロット数が 1,440 より少ないのは、ログ収集が開始された当日なので、1 分ごとの集計が 192 回しか行われなかったからである。syslog の収集機構は、5/27 の午後には動作していたが、ラックへと積載さ

れた各機器の syslog 出力の設定時間が違うため、収集開始時刻も各機器により異なる. そのため ShowNet 全体として syslog を収集開始した時刻を定めることはできない. また 6/10 の時間スロット数が 1,440 より少ないのは、17 時に ShowNet がシャットダウンされ撤収が開始されたためである。

アラート回数は、ボリンジャーバンドの UpperLimit を超えた回数を表し、発生率は時間スロット数に対して UpperLimit を超えた回数をパーセンテージで表したものとなる。ログ総量の合計から算出した ShowNet 期間中の全アラート発生率は 2.33%となり、集計されたログの 97.67%は UpperLimit を超えなかった。

ShowNet の開催期間は主に準備期間 (Hotstage) と会期準備期間、会期の 3 つに分けられる。Hotstage は 5/27 から 6/3 まで, 会期準備期間は 6/4 から 6/7 まで, 会期は 6/8 から 6/10 までとなる。Hotstage 期間中は、新しい機材の繋ぎこみやネットワークの構築、ソフトウェアの稼働などが進められ、syslog の総量が日を追うごとに増加する。会期準備期間には、出展者が出展準備のために会場に持ち込んだ端末を ShowNet へと接続し始める. 会期準備期間、並びに会期中には 200 万行から 1,200 万行ほどでログの総量が推移しているが、ログの総量が増加してもアラートの発生率は 1%台から 2%台の間で推移している。

6.4.2 アラートの回数とアラートレベルの割合

次に表 6.2 に示したアラートレベルごとのアラート割合を集計した。表 6.4 がアラートレベルごとに分類した集計結果となる。全アラート回数のうち、Low アラートが 50.63%, Middle アラートが 22.89%, High アラートが 26.47%という割合を占める。Low アラートと Middle アラートがアラートの大部分である約 75%を占めている。

表 6.4: アラートレベル集計

日付	アラート回数	Low	Middle	High
5/27	12	2	4	6
5/28	46	22	14	10
5/29	50	26	13	11
5/30	48	19	10	19
5/31	27	10	8	9
6/1	47	21	11	15
6/2	38	16	13	9
6/3	28	15	4	9
6/4	38	23	4	11
6/5	24	16	2	6
6/6	24	12	4	8
6/7	24	12	8	4
6/8	24	15	5	4
6/9	24	21	2	1
6/10	22	11	7	4
合計	476	241	109	126
割合		50.63%	22.89%	26.47%

6.4.3 アラート閾値判定との比較と指数移動平均との比較

動的なアラート判定を行う前に、閾値を基準とした判定に関しても実験を行った [54]。閾値ベースの判定では High アラートは「2σ との差が 1,000 以上離れた値」と定めた。アラートを閾値と比較して求めたアラート発生回数と、本提案で動的に計算をした 2 つの移動平均 (単純移動平均と指数移動平均) のアラート回数を比較した図が図 6.5 となる。単純移動平均と比較して、指数移動平均はデータに対して指数関数的に重みを減少させるという特徴がある。重みは指数関数的に減少するので、最近のデータを重視するとともに古いデータを完全には切り捨てないという特徴がある。つまり、指数移動平均は直近のデータの動きに早く反応するということであり、単純移動平均よりも異常を素早く発見できる可能性がある。

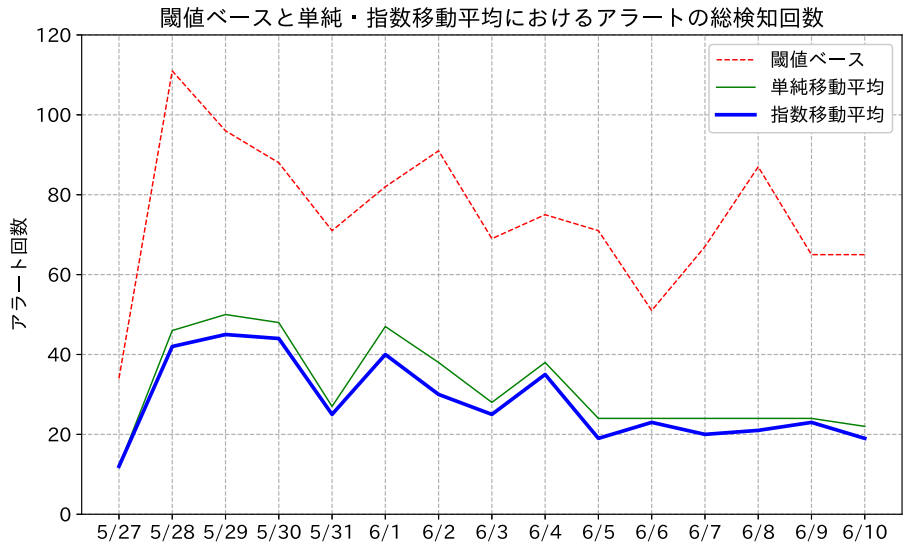

図 6.5: アラート総検知数比較

6.5 考察

6.5.1 アラートの発生率について

6.4.1 で示したアラート発生率の結果より、ShowNet における syslog の総量はボリンジャーバンドで仮定した $\pm 3\sigma$ 以内である 0.27%を上回る水準となっており、統計予想回数を上回りアラートが発生していた。つまり株式市場ほどイベントネットワークは安定しておらず、純粋なボリンジャーバンドのアルゴリズムでは、正規分布の予想値から外れることを意味する。しかしながら、会期が近づくにつれログの総量は純増しているがアラートの発生率はログの総量に比例して増加はしていない。表 6.5 に示すように、全体のログ総量は約 97.60%の確率で $\pm 3\sigma$ の範囲に収まっている。結果として異常通知回数はボリンジャーバンドの統計範囲には収まらなかったが、syslog の総量は $\pm 2.2\sigma$（97.22%）から $\pm 2.3\sigma$（97.86%）の間の分布となり、ボリンジャーバンドアルゴリズムの分布に近似している。σ のパラメータを調整することで syslog 総量による異常検知を行うことは有効であると考

表 6.5: $\pm 3\sigma$ の範囲に収まる確率

日付	時間スロット数	$+3\sigma$ 以上	-3σ 未満	± 3 の範囲
5/27	617	12	0	98.06%
5/28	1,440	46	2	96.67%
5/29	1,440	50	0	96.53%
5/30	1,440	48	6	96.25%
5/31	1,440	27	3	97.92%
6/1	1,440	47	0	96.74%
6/2	1,440	38	1	97.29%
6/3	1,440	28	0	98.06%
6/4	1,440	38	1	97.29%
6/5	1,440	24	0	98.33%
6/6	1,440	24	0	98.33%
6/7	1,440	24	0	98.33%
6/8	1,440	24	0	98.33%
6/9	1,440	24	0	98.33%
6/10	1,083	22	1	97.88%
合計	20,420	476	14	97.60%

えられる。

次に、閾値でのアラート数の結果と比較してアラート発生回数の総数は、単純移動平均では 1,123 回から 476 回へと大きく減少した。指数移動平均を用いることで、単純移動平均で求めたアラート発生回数の総数である 476 回から 423 回へとさらに減少した。図 6.5 が示すとおり、日々のアラート発生率は閾値ベース、単純移動平均、指数移動平均の順に減少している。指数移動平均を異常検出に採用することによりアラート検知回数が大きく減少したことから、より精査された異常をネットワーク運用者へ通知することが可能となる。

6.5.2 レベル分けの有効性

次に 6.4.2 で示したアラート回数とアラートレベルの割合については、Low アラートと Middle アラートが約 75%を占めており、これらのアラートを取り除き

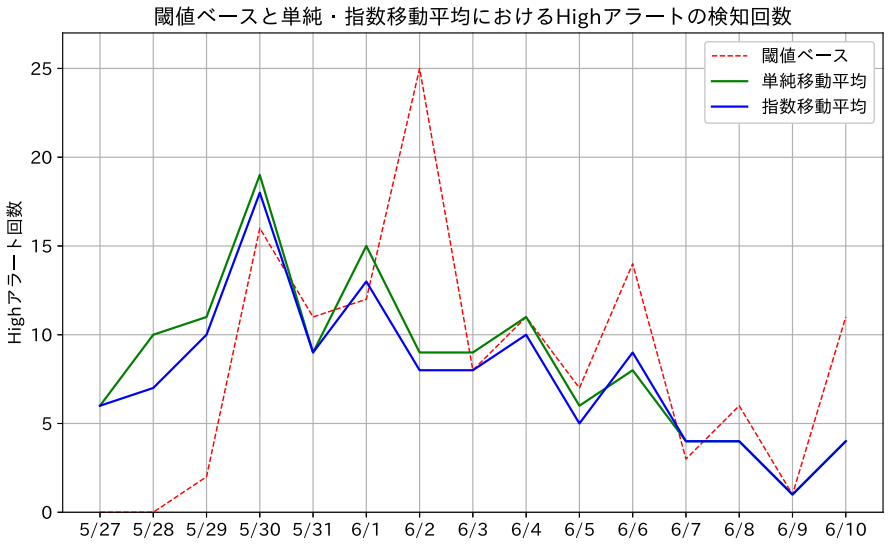

図 6.6: High アラート検知数比較

High アラートのみを通知すれば、異常通知を減らす意味では有益なフィルタとなりうる。

本研究では、閾値でのアラート数と比較して、アラートレベルを固定値の幅で求めるのではなく、動的にレベル分け計算を行う手法を用いた。図 6.6 が示すように、結果として High アラートの総数は閾値ベースの判定と比較して大きく減少していない。閾値ベースの判定では High アラートは「2σ との差が 1,000 以上離れた値」と定めたが、本研究では 5σ を超えた値とした。閾値の 1,000 という数値に根拠がなかったわけだが、本研究では 3σ よりもさらに低い確率で発生しうる異常値として 5σ を基準値とした。High アラートの総数は閾値ベースと単純移動平均ではほぼ差がない状態となり、指数移動平均を利用した場合に微減という結果となった。アラートのレベル分けでは閾値に固定値を用いるよりも、指数移動平均を用いた方がより効果的に大きな変動を捉えられるという結果となった。閾値ベースの判定では、本来運用者が気づきたい 1,000 以下の値の中での大きな変動を見つけられないことを意味するが、本研究では 1,000 以下の値の中でも起こりうる大きな

変動を検知し通知することが可能となる。

6.5.3　誤検知への対応と実運用性

本手法ではアラートのレベル分けを行うことで、本来通知すべきアラートを通知しないフォールスネガティブが発生する可能性が高まる。しかしながらレベル分けを行わない場合には、通知されるアラートの総数は、「日に最大 50 件ほど」になる場合があり、ネットワーク運用者が運用を行う上でアラートの調査負荷が高まる。本提案で用いた指数移動平均を利用したレベル分けを行うことで、High アラートは日に 18 件以下の検知数を記録した。このアラート件数は、レベル分けを行わない「日に 50 回」というアラート数と比較して、少数のネットワーク運用者でも現実的にハンドリング可能な回数である。

ボリンジャーバンドアルゴリズム単体では、誤検知を発見することはできないが、他の監視システムと連携し相関を取ることで誤検知かどうかを判断することは可能である。また、アラート発生時間のログのみを意味解析し、機械学習を行って誤検知か否かを判定することも可能性の一つである。しかしながら、本提案はあくまで syslog の総量を用いた異常検知とネットワーク運用者が実オペレーション可能なアラート通知回数の両立を目標としており、本提案ではその両方が実現可能となっている。

6.5.4　他の研究との比較

本研究では syslog のヘッダーのみを処理対象とし集計を行ったが、他の研究では通常必要なログの意味解析を行い、異常を検知する。これにより意味解析を行う計算を省略できるばかりか、集計処理を行う場合にも対象となるデータ量が少なくなり計算を高速に行える。またボリンジャーバンドに必要な計算は移動平均と標準偏差の計算のみであるため、全体的なアルゴリズムの計算量を少なく押さ

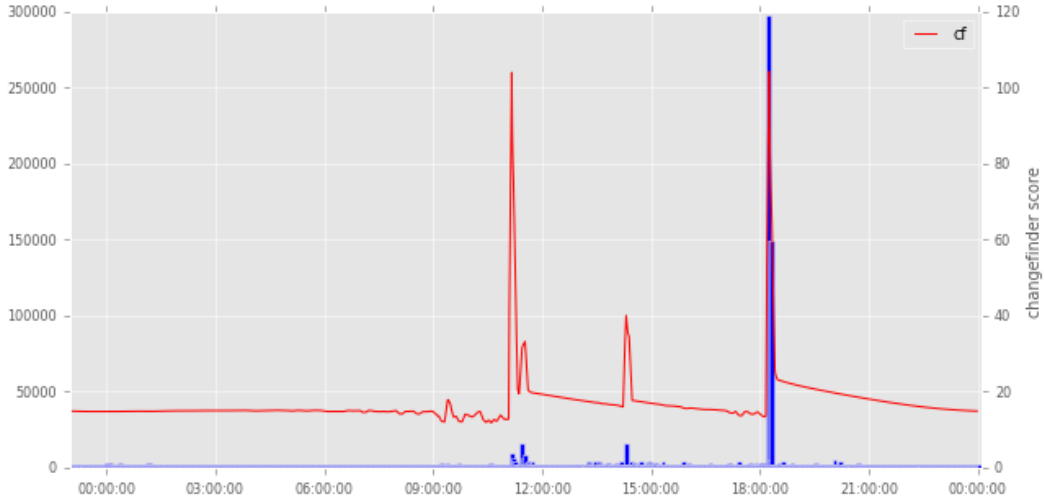

図 6.7: 6/6 の変化点検出

えられる。これは、他の手法に対して計算量が少なく異常検知が行えることを意味する。

本手法と比較するため、ChangeFinder アルゴリズムを適用した計算を行った. 図 6.7 は 6/6 のデータから ChangeFinder の計算結果を描画したものであり、2 つの大きなスコアの増加が見られる. 6/6 では 11 時過ぎと 18 時過ぎにログの総量が大きく変動したことを意味する高いスコアの値となっている。18 時過ぎの大きな変動の原因は、SNMP GET の request と response がついとなり 18:10-18:16 の間に数十万行のログが出力されていた。実際には、11 時過ぎの変化よりも 18 時過ぎの変化の方が何倍も大きく、運用者への通知を行いたい事象は 18 時過ぎの変化のはずだが、スコア的にはどちらの時間の変化もほぼ同じ値になっている. ChangeFinder ではログ総量の変化点を見つけることが可能だが、ログ総量がどの程度の割合で増加したか判断することはできず、アラートのレベル分けを実装することは難しく、結果としてネットワーク運用者へアラートの重要度を通知することが困難となる。

6.5.5　イベントネットワーク特有の問題への対応

イベントネットワーク運用はサービス提供を行うネットワークであり、トラブル対応の速度がネットワーク品質に大きく影響する。ログの意味解析を行う手法の場合、ログの急増により処理しなければならない対象ログも急増し、解析処理自体が遅延してしまい運用者への異常通知が遅れる可能性がある。本手法のようにログの総量にのみ着目し、かつ統計的手法によって異常検出計算も軽量なアルゴリズムを用いることで運用者への異常通知が高速化し、対応速度を重視した運用を行うことが可能となり、イベントネットワーク以外の様々なネットワークやサーバ運用等にも適用することで運用の品質を向上することができる。

一般的に開催されるカンファレンスやシンポジウムの開催期間は長くて一週間程度や短くて 2-3 日という中でイベントネットワークが構築されるが、ShowNet は 2 週間という比較的長い時間をかけ構築運用される。本提案手法では、1 時間分のウィンドウ幅でデータが学習され計算される. 短期間のイベントネットワークおいても、学習に必要なデータを短時間で蓄積でき計算することができ本提案アルゴリズムが有効に働く.

6.5.6　実データの利用

ShowNet の syslog 運用では事前に運用メンバーが把握している特定キーワードによる閾値ベースの監視が行われている。syslog 可視化が行われるまでは、ループが発生した場合には解決に数十分から数時間かかる場合もあった。syslog の実データが監視に用いられることにより、異常検知までの時間が数分まで大幅に減少した。閾値ベースの syslog 監視は、発生した事象がイベント回数として可視化され通知されるため運用者にとって有用に働く。しかしながら、運用メンバーが把握している特定キーワードにのみ対処可能であり、それ以外のログ異常に関してはやはり運用メンバーの勘によるシステム運用がなされている。検知されたか

らと言ってそれが本当に検知したいトラブルなのか、ネットワーク機器へ設定を行ったため発生した記録ログなのか判断がつきにくく、最終的には運用者が確認を行うこととなる。また、特定キーワードはイベントごとに変化する可能性があり、さらに監視を行う閾値の適切な調整が必要となり、運用者の経験に頼るシステムでは自動化は困難である。

ShowNet という実際に運用構築される大規模なイベントネットワークにおける、マルチベンダ機材が混在する特殊な環境において収集された syslog の総量を監視し、異常を自動検知することが、ShowNet 運用の属人性の排除を進める一歩となる。さらに本実験が実データを用いて評価されたということは、本実験がシミュレーションではなく、実際の ShowNet の運用自動化を進める大きなアドバンテージとなる。

6.6 まとめと今後の課題

6.6.1 まとめ

本研究では、ShowNet という大規模でマルチベンダ機材が大量に投入されるイベントネットワークでの異常検知を、syslog の総量を集計してボリンジャーバンドアルゴリズムを用いて解析し検知する手法を提案した。これによりネットワーク運用者の経験と勘に頼る属人性を排除した自動的な異常検知を行い、ネットワーク運用者へ通知するというシステムの実現が可能となる。また、指数移動平均をベースとした動的なアラートのレベル分けを追加することで、ネットワーク運用者に対して現実的な異常発生回数を通知する機構を実現した。結果として、閾値ベースのアラート判定よりも良い精度でのアラート検出とレベル分けを行うことが可能となり、ネットワーク運用者へのトラブル対応の高速化の可能性を示した。

本提案は、ログの意味解析をあえて行わず本来は意味解析が有効である機械学習や統計分析では実現できない、軽量で高速に計算できる単純なアルゴリズムが

有効に働いた。高速な計算による自動的な異常通知の結果として、トラブルへの初期対応を素早く実行することが可能となり、属人的なイベントネットワーク運用に大きく貢献できる可能性がある。また、本手法はイベントネットワークのみならず、企業や大学などで行われるネットワーク運用やサーバ運用に対しても有効に働き、多くのネットワークのトラブル解決のための一手法として貢献できる。

結果として、6.5.1 で述べたようにボリンジャーバンドの幅である $\pm 3\sigma$ の幅で、syslog の総量は推移しなかった。しかしながら、本実験で syslog の総量は $\pm 2.2\sigma$（97.22%）から $\pm 2.3\sigma$（97.86%）の間の分布となり、ボリンジャーバンドアルゴリズムの分布に近似した。大規模で不安定なイベントネットワークでの分布がある一定の σ の幅で収まるということは、実際の安定したネットワークであればボリンジャーバンドの幅でのログの推移は実運用に適用可能であると予想する。

6.6.2 今後の課題

本研究ではイベントの事後に分析処理を行ったが、ログをリアルタイムに処理することで本研究を実運用へと組み込むことが可能となり、ネットワーク管理者へリアルタイムでアラートを通知することが可能となる。また、総量にのみ着目するのではなく送信元ホスト別に異常を検出する仕組みを作ることで、ネットワーク管理者に対して異常発生源が送信元ホストであるという直接的な問題解決の提示を行える syslog 監視システムを提案できる。さらに、セキュリティ機器のログを対象にすることにより、セキュリティインシデントに対する異常検知という点で貢献可能である。

第7章　結論

本研究ではまず、大規模なネットワークを安定的に管理するための問題点を提議した。また、大規模ネットワークでの異常検知の問題とデータ量の問題、そこで収集される時系列データについて述べた。そして、障害が発生した場合にどの程度の損失が発生するかについて考察した。

次に、大規模な時系列ログデータを処理する一つの実装として、Hayabusa を設計、評価し、スタンドアロン環境で高い検索性能を発揮するアーキテクチャを実現した（第 3 章）。さらに、第 2 章で実現したスタンドアロン版の Hayabusa の性能を高めるため、スタンドアロン版の Hayabusa のシンプルさを崩さずに、分散版の Hayabusa を設計、評価し、高い検索性能を実現した（第 4 章）。そして、分散版 Hayabusa の課題として挙がったストレージの問題とリクエスト処理の問題点について改良した新アーキテクチャを設計し評価を行い、その実装を OSS として公開した（第 5 章）。最後に、高速に集計データを取得可能な Hayabusa のアーキテクチャ特性を生かすため、統計データを用いた軽量なアノマリ検知機構を提案し、大規模なイベントネットワークで取得した実データを用いて評価を行い、異常検出のための仕組みを実装し実現した（第 6 章）。

7.1　本研究の成果

第 1 章では、大規模なネットワークを安定的に管理するための問題点として、ネットワーク管理者の視点にたち安定的なネットワーク運用のためのデータ収集の難しさと、その収集したマルチベンダ機材からの大量なデータ群を実際のイン

シデント対応やトラブルシューティングへとフィードバックさせるために、必要な情報を高速に抽出する必要性について述べた。また、大規模ネットワークでの異常検知について、大量のデータかつマルチベンダ機器のログの扱いの難しさについて述べた。そして大量データの扱いとそのデータをフィルタせずに保持する難しさについて考察し、収集される時系列データの重要性について解説を行った。さらに、システム障害による損失を考察し、安定したネットワーク管理の重要性とトラブルの即応性の重要性について論じた。

第 2 章では、大量のログを収集、検索するための汎用的なシステムについての解説と、時系列データの収集に特化した時系列データベースについての分析と考察を行った。また、異常検知手法に関して古くから研究される統計を用いた手法に関して考察を行った。

第 3 章では、巨大なネットワークで発生する大量の時系列ログデータを処理するための新しい提案として、シンプルでかつ高速なアーキテクチャである Hayabusa を設計・実装した。Hayabusa は分散化した Aapache Spark よりも高速に動作し、システム構造や運用に関してもシンプルさを保ったままシステムを実現することできた。Hayabusa は、ログ検索を行いたいユーザに対し、高い検索性能と運用の容易性を提供するプラットフォームとなり得ることを証明した。そして、ログを時系列データとして扱うために適した形で保存し、高速に処理をするためのアーキテクチャを実現した。

第 4 章では、他のスケールアウトする分散システムと比較した場合に、スタンドアロン版の Hayabusa では超えられない性能を打破するために、Hayabusa 自体の分散システム化を行った。各 Worker ノードへとデータを複製し、クライアントからのリクエストを ZeroMQ を用いることで均等に各 Worker ノードへと分配することで、スタンドアロン版の Hayabusa のシンプルさを崩さずに分散版の Hayabusa を設計、実現することができた。結果として、144 億行のログを約 7 秒でフルスキャンし集計可能なシステムとして分散システム化することに成功した。高速な

ログ検索速度の実現とシンプルなシステムの設計は、分散 Hayabusa を利用するユーザや運用者に恩恵をもたらす可能性を与えた。

第 5 章では、分散版 Hayabusa の課題として残っていたストレージの複製とエラーハンドリング部分に関して、NFS を用いたネットワークストレージと、リクエスト管理機構を導入する設計と実装を行った。これにより、全ての Worker ノードへとデータを複製していた処理はなくなり、ストレージが集中管理可能となった。また、NFS を冗長化することで、Primary/Backup の 2 箇所へのデータ複製を行うことで耐障害性を高めた。次に、リクエスト機構を RequestBroker として実装することで、クライアントが投入したリクエストがどの程度処理されたか・エラーが発生したかなどを把握することが可能となった。これら改良を加えた分散版 Hayabusa の新アーキテクチャを OSS として幅広く公開したことにより、全世界のユーザに利用される可能性を実現した。

第 6 章では、ネットワーク全体の異常を検知するために、大規模なネットワークイベントで収集した実データを用いて、異常検知アルゴリズムを設計した。アルゴリズムには金融取引で利用されるボリンジャーバンドを用い、さらにアラートの判定に独自なアルゴリズムを追加することで、高い異常検知性能を実現した。結果として、大規模なイベントネットワークであったとしても、ログの統計値はある一定範囲で推移することがわかった。さらに安定したネットワークであれば、ログの総量はボリンジャーバンドに近い統計値の幅で推移することが予想され、異常検知は十分に可能であると推測される。本実装は、ログの総数を時間単位で集計した時系列データの集計値として用いることから、Hayabusa の高速な集計性能とマッチするアルゴリズムとなり、高速な異常検知システムの実現を示唆した。

7.2 本研究の社会的な意義

本研究では、大量に収集されるスキーマレスなログデータを時系列データとして最適な形で蓄積・検索可能なアーキテクチャとして Hayabusa を設計し、特定条件においてわずか 10 台の小規模な分散処理クラスタを用いることで高い検索処理性能を実現した。これにより、低い運用コストで自由に利用可能な分散処理基盤として世の中のネットワーク管理者が広くトラブルシューティングやインシデントレスポンスへと利用可能となる。また、Hayabusa のシステムが OSS として公開されていることによりシステムがブラックボックス化せずに、データセンターやオンプレミス環境、クラウドサービス上であっても誰もが望む場所にシステムを構築し改良することが可能となった。

そして、ボリンジャーバンドを用いた異常検出アルゴリズムを実現し、有効性を評価した。これにより、ネットワーク管理者は大規模な実ネットワークにおいて、本手法を適用しネットワークの異常を発見することができる。通常の大規模ネットワークは、ShowNet のようなイベントネットワークほど不安定ではなく安定稼働するものと推測できるため、ShowNet のような不安定なネットワークでの異常検出結果は、実ネットワークへの応用を考える上では十分な実績と言える。

シンプルで高速な時系列ログデータ検索基盤と異常検知手法を用いることで、ネットワーク管理者はネットワーク管理のための本質的な作業時間を作り出すことができ作業に集中することができるようになるため、自身が管理するネットワークをさらに安定的に運用可能な状況へと導くことができる。

7.3 本研究の学術的な意義

本研究では時系列ログデータに対するシステム最適化という実装を実現した。特定の問題解決（本研究では時系列ログデータの蓄積と検索）へと特化したアーキテクチャを実現することで性能向上が望めるという事例を示した。また、コン

ピュータのアーキテクチャの変化は高速で、アーキテクチャに大きなパラダイムシフトが発生した場合に追従可能なシステムアーキテクチャを構築することは難しい。Hayabusa は、大量の時系列データを CPU のメニーコアに適した形で処理可能であり、今後訪れるであろうさらなるメニーコア時代に追従可能なアーキテクチャの一つを実現した。

そして、Hayabusa を OSS として全世界に公開したことにより、小規模な研究組織での低コストかつ低い運用負荷での導入が実現可能となり、広くアカデミアの研究に役立つと推測する。

異常検出における統計学を用いた手法の提案と実現、ならびに実データを用いた評価に関しては、特に実データの有用性を示す結果となり、本来運用データとしてしか用いられないデータをうまく研究課題として取り込んだ一例として示すことが可能である。

7.4 本研究分野の課題と展望

Hayabusa のストレージ実装として、本研究ではデータの複製と NFS によるデータの集中という 2 つのパターンを用いた。どちらの実装も検索性能かストレージ容量かのトレードオフとしての選択を行ったが、どちらのパターンも選択可能なアーキテクチャ適用事例が考えられる。例えば、ファーストクラスの高速性を求めるならばコストを度外視した複製パターンのアーキテクチャを選択することは可能であり、コスト意識が高ければストレージ容量を重視したそこそこの処理速度を問題解決に選択できる。さらに、時系列にデータを保存することに特化した Hayabusa 専用の分散ファイルシステムの実現などが考えられる。それら課題の解決や要件にあった適用手法を考えることで、Hayabusa のさらなる分散処理システムとしての性能向上が見込まれる。

次に、Hayabusa は基盤として動作するために、それ単体ではただの検索エンジ

ンと同等である。しかしながら、本研究で実装した異常検知の仕組みや機械学習ライブラリとの融合、さらにエッジコンピューティングとしての IoT 機器のリアルタイムログ解析エンジンや大規模な監視システムへの組み込みなど、上位で動作するアプリケーションやミドルウェアと融合することでシステムを統合することが可能となる。

多くのネットワーク管理者は、ログ蓄積を検索・可視化をすでに行ってネットワーク運用を行っていると考えられる。異常検知に関しても監視情報を元に閾値ベースのものや異常監視を行う製品の導入を行っている場合もある。それらの一部を本研究での成果で置き換える、または組み込むことでさらなる運用自動化を促進し、ネットワーク管理者の負荷を下げ、ネットワークを安定稼働させるために本来やらなければいけないことへとさらに時間が割けるようになることを望む。

謝辞

本研究を進め、論文を執筆するにあたり終始御指導賜りました篠田陽一教授に深く感謝致します。また、本学、丹康雄教授、知念賢一准教授、京都大学 岡部寿男教授、慶應義塾大学 植原啓介准教授には、審査員をお引き受けいただき、多くの助言を頂けたことを深く感謝いたします。また、副テーマをご指導いただいてインターネットと運用シンポジウムへの投稿を勧めていただき、結果として複数の賞をいただくことになりました高知工科大学 敷田幹文教授に論文指導等多くのご指導をいただいたことに深く感謝いたします。

次に、本学 篠田研究室の宇多 仁助教には、研究に関して様々なご助言をいただき深く感謝いたします。また篠田研究室の修了生、現役メンバー、東京サテライトオフィスのメンバーにもたくさんの議論に付き合っていただき感謝いたします。特に実験をお手伝いいただいた情報通信研究機構北陸 StarBED 技術センターのみなさまにおかれましても、多大なご助力を頂けたことを心より感謝いたします。

また、株式会社インターネットイニシアティブの皆さま、株式会社 IIJ イノベーションインスティテュート技術研究所の長健二朗所長、島慶一副所長、和田英一顧問、その他たくさんの方々に多大なるご助力を頂けましたことを心より感謝いたします。そして株式会社レピダムの皆さま、ならびにココン株式会社の皆さまにおかれましても、貴重な時間をいただきご助力いただけたことを心より感謝いたします。

WIDE プロジェクトの皆さまには、合宿や研究会などで研究を進める上で様々な有益なご意見をいただきましたことを感謝いたします。特に「サーバー脅威ビッグデータの解析とリアルタイム攻撃検知と予測」を行う NML(Network Muscle Learning)

プロジェクトのメンバーである、関谷勇司博士、石原知洋博士、岡田和也博士、中村遼博士、宮本大輔博士、松浦知史博士、北口善明博士には論文の添削だけではなく、たくさんの議論にお付き合いいただき多大なるご助力を頂けましたことを心より感謝いたします。また、株式会社 Preferred Networks の宇夫陽次朗博士、浅井大史博士、土井裕介博士に多大なご助力を頂けたことを心より感謝いたします。

また、株式会社クルウィットの井澤志充さま、清原智和さま、フリーランスエンジニアの黒川仁さまにおきましては、OSS 版 Hayabusa の実装と性能測定に関してご助力をいただけたことを心より感謝いたします。

ShowNet NOC チームメンバーの皆様にはデータの取得や解析を行う上で多大なるご支援をいただきました。ShowNet のデータなしには今回の研究が実現しなかったため、心より感謝いたします。

最後に、研究や生活を支えてくれた家族である、妻・陽子、父・源内、母・トヨコにはたくさんの迷惑をかけました。たくさんの支援をいただけたことを心より感謝いたします。また、愛すべき子供達である、娘・希、息子達・和史、惇にはたくさんの我慢をさせてしまったことをここに謝罪するとともに、君たちがいたからこそ頑張れたことを感謝いたします。心から感謝を表し謝辞と致します。

本研究に関する発表論文

論文誌（査読あり）

- 阿部博, 敷田幹文, 篠田陽一, ”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”, 情報処理学会論文誌, 59 巻 3 号, pages 1006-1015, mar 2018.
- 阿部博, 島慶一, 宮本大輔, 関谷勇司, 石原知洋, 岡田和也, 中村遼, 松浦知史, 篠田陽一, ”時間軸検索に最適化したスケールアウト可能な高速ログ検索エンジンの実現と評価”, 情報処理学会論文誌, 60 巻 3 号, pages 728-737, mar 2019.

国際会議（査読あり）

- Hiroshi Abe, Keiichi Shima, Yuji Sekiya, Daisuke Miyamoto, Tomohiro Ishihara, Kazuya Okada, ”Hayabusa: Simple and Fast Full-Text Search Engine for Massive System Log Data”, Conference on Future Internet (CFI’17), June 16, 2017.
- Keiichi Shima, Daisuke Miyamoto, Hiroshi Abe, Tomohiro Ishihara, Kazuya Okada, Yuji Sekiya, Hirochika Asai, Yusuke Doi, ”Classification of URL bitstreams using Bag of Bytes” , First International Workshop on Network Intelligence (NI2018), February 20-22, 2018.

国内会議（査読あり）

- 阿部博, 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”, 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.

- 阿部博, 篠田陽一, ”スケールアウト可能なログ検索エンジンの実現と評価”, インターネットと運用技術シンポジウム 2017 論文集, volume 2017, pages 73-80, nov 2017.

- 野村孔命, 阿部博, 菅野哲, 力武健次, 松本亮介, Web アプリケーションテストを用いた SQL クエリのホワイトリスト自動作成手法, インターネットと運用技術シンポジウム 2018 論文集, volume 2018, pages 106-113, nov 2018.

- 坪内佑樹, 脇坂朝人, 濱田健, 松木雅幸, 阿部博, 松本亮介, HeteroTSDB: 異種混合キーバリューストアを用いた自動階層化のための時系列データベースアーキテクチャ, インターネットと運用技術シンポジウム 2018 論文集, volume 2018, pages 7-14, nov 2018.

国内研究会

- 阿部博, 井上朋哉, 篠田陽一, ”隠蔽された分散処理環境を透過的に利用可能な基盤の提案”, マルチメディア、分散協調とモバイルシンポジウム 2016 論文集, volume 2016, pages 293-300, Jul 2016.

- 長谷川幹人, 阿部博 , 関谷勇司, ”PTP(Precision Time Protocol) の相互接続実証実験の現状と方向性 (Phase2) 〜 Interop Tokyo 2016 ShowNet における結果からの考察 〜”, 電子情報通信学会 インターネットアーキテクチャ研究会, 信学技報,vol.116,no.203,IA2016-13,pp.1-6,2016 年 8 月.

- 淺葉祥吾, 北口善明, 石原知洋, 高嶋健人, 阿部博, 篠田陽一, ”階層的ネットワーク計測における計測項目の相関分析”, マルチメディア、分散、協調とモバイル (DICOMO2018) シンポジウム論文集, pp. 1407-1412, July 2018

その他

- 阿部博, 島慶一, ”Hayabusa：高速に全文検索可能なログ検索エンジン”, Internet Infrastructure Review, vol.38, 2018 年 3 月

参考文献

[1] Amazon Web Service. https://aws.amazon.com/.

[2] Ansible. https://www.ansible.com/.

[3] Apache Hadoop. http://hadoop.apache.org/.

[4] apache-loggen. https://github.com/tamtam180/apache_log_gen.

[5] Cassandra. http://cassandra.apache.org/.

[6] Cloudera. https://www.cloudera.com/.

[7] Cloudera's open source software distribution. https://www.cloudera.com/products/open-source/apache-hadoop/key-cdh-components.html.

[8] Distributed Hayabusa. https://github.com/hirolovesbeer/hayabusa2.

[9] Elasticsearch. https://www.elastic.co/products/elasticsearch.

[10] FortiAnalyzer. https://www.networld.co.jp/product/fortinet/pro_info/fanalyzer/.

[11] Gentelella. https://github.com/puikinsh/gentelella.

[12] GNU Parallel. https://www.gnu.org/software/parallel/.

[13] Hortonworks. https://hortonworks.com/.

[14] Influxdb. https://www.influxdata.com/time-series-platform/influxdb/.

[15] Interop Tokyo. http://www.interop.jp/.

[16] JANOG Meeting. https://www.janog.gr.jp/meeting/.

[17] KairosDB. https://kairosdb.github.io/.

[18] MapR. https://mapr.com/.

[19] Panorama. https://www.paloaltonetworks.jp/products/management/panorama.

[20] ShowNet. http://www.interop.jp/2016/shownet/.

[21] Splunk. https://www.splunk.com/.

[22] SQLite. https://www.sqlite.org/.

[23] syslog-ng. https://syslog-ng.org/.

[24] UDP Samplicator. https://github.com/sleinen/samplicator.

[25] VMware vRealize Log Insight. https://www.vmware.com/products/vrealize-log-insight.html.

[26] Zabbix. https://www.zabbix.com/.

[27] An inside look at google bigquery. 2013.

[28] 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.

[29] H. Y. . H. Adachi. Automation of rolling upgrade for hadoop cluster without data loss and job failures. Technical report, Yahoo Japan, may 2017.

[30] J. A. Bollinger. *Bollinger on Bollinger Bands*. McGraw-Hill Education, 2001.

[31] G. E. P. Box and G. Jenkins. *Time Series Analysis, Forecasting and Control*. Holden-Day, Inc., San Francisco, CA, USA, 1990.

[32] Chang F, et al. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4:1–4:26, 2008.

[33] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[34] George L. *HBase: The Definitive Guide: Random Access to Your Planet-Size Data*. O'Reilly Media, Inc., 1st edition, 2011.

[35] R. Gerhards. The syslog protocol. RFC 5424, RFC Editor, March 2009. `http://www.rfc-editor.org/rfc/rfc5424.txt`.

[36] P. Hintjens. 0mq - the guide, 2011.

[37] IPA. 「企業における情報システムのログ管理に関する実態調査」報告書について. Technical report, IPA, Jun 2016. https://www.ipa.go.jp/security/fy28/reports/log_kanri/index.html.

[38] P. S. Kalekar. Time series forecasting using holt-winters exponential smoothing. *Kanwal Rekhi School of Information Technology 4329008*, pages 1–13, 2004.

[39] J. Kleinberg. Bursty and hierarchical structure in streams. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 91–101, New York, NY, USA, 2002. ACM.

[40] A. Martínez, M. Yannuzzi, V. López, D. Lopez, W. Ramírez, R. Serral-Gracià, X. Masip-Bruin, M. Maciejewski, and J. Altmann. Network management

challenges and trends in multi-layer and multi-vendor settings for carrier-grade networks. *IEEE Communications Surveys & Tutorials*, 16:2207–2230, 2014.

[41] W. McKinney. pandas: a foundational python library for data analysis and statistics.

[42] 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.

[43] O'Neil P, et al. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996.

[44] Pelkonen T, et al. Gorilla: A fast, scalable, in-memory time series database. *41st International Conference on Very Large Data Bases (VLDB)*, 8(12):1816–1827, 2015.

[45] 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.

[46] J.-i. Takeuchi and K. Yamanishi. A unifying framework for detecting outliers and change points from time series. *IEEE Trans. on Knowl. and Data Eng.*, 18(4):482–492, Apr. 2006.

[47] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another

resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[48] L. Winkler. Scinet: 25 years of extreme networking. In *Proceedings of the Second Workshop on Innovating the Network for Data-Intensive Science*, INDIS '15, pages 3:1–3:9, New York, NY, USA, 2015. ACM.

[49] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[50] 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.

[51] 中村和敬, 當仲寛哲. ユニケージ開発手法に基づく unix ファイルシステムとシェルを用いたデータベースの構築と操作. Technical Report 16, ユニバーサル・シェル・プログラミング研究所, ユニバーサル・シェル・プログラミング研究所, may 2017.

[52] 中西一生, 井ノ上攻, 深野隆行, 長島祐之. マルチベンダ l2nw 環境における nw 管理手法の一検討. **電子情報通信学会技術研究報告**. *ICM,* **情報通信マネジメント** *: IEICE technical report*, 109(463):99–103, mar 2010.

[53] 阿部博, 島慶一, 宮本大輔, 関谷勇司, 石原知洋, 岡田和也, 中村遼, 松浦知史, 篠田陽一. 時間軸検索に最適化したスケールアウト可能な高速ログ検索エンジンの実現と評価. **情報処理学会論文誌**, 60(3):728–737, mar 2019.

[54] 阿部博, 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.

[55] 阿部博, 篠田陽一. スケールアウト可能なログ検索エンジンの実現と評価. In **インターネットと運用技術シンポジウム** *2017* **論文集**, volume 2017, pages 73–80, nov 2017.