2018年度　卒 業 論 文

Apache FlinkのBackPressure機構の挙動解析

2019年1月23日

コンピュータサイエンス学科
学生証番号: 444053

有岡 直一

指導教員: 林原 尚浩

京都産業大学コンピュータ理工学部

概 要

昨今，データ通信量の増加によってバッチ処理より, ストリーム処理が必要になってきた. その中で, 連続的に発生し続けるデータをリアルタイムに解析, 分析等の処理を行い続けるための分散ストリーム処理プラットフォームが普及し始めた. 分散ストリーム処理プラットフォームの中の, ストリームプロセッシング系とされるものの中で台頭が見られる Apache Flink は他のプラットフォームと比べて,

- 耐故障性に優れる
- ストリーム処理, バッチ処理を両方サポート
- BackPressure 機構の設定を変更可能

などが挙げられ, 本研究ではデータフロー制御の BackPressure 機構に着目した.

BackPressure 機構は, キューのタスクが溢れないように必要に応じて受け入れ速度を遅めたり, 止めたりする機構である. Apache Flink 内での BackPressure 機構の振る舞いとしては, ジョブマネージャが 50 ミリ秒毎の 100 個のタスクのうちキューに残った個数を Ratio で表し,

- $OK : 0 <= Ratio <= 0.10$
- $LOW : 0.10 < Ratio <= 0.5$
- $HIGH : 0.5 < Ratio <= 1.0$

のようにウェブインターフェースに表示される. これは, 100 個のうち 10 個までがキューに残った場合に OK と表示されるようになっていて, Apache Flink の BackPressure の作動と判断するための設定, 50 ミリ秒の時間や 100 個の個数を変更することができる. しかし, Back Pressure の挙動は通信環境などの動作環境に対してどのような依存関係があるのか明らかになっていないため, 本研究では動作環境による BackPressure 機構の環境依存を Contention Aware Metrics を用いて調査する.

目 次

第1章　はじめに

1.1　研究背景

近年, 機械学習や予測分析など, IoT 導入済みの企業も増え, 膨大なデータを溜め込み, バッチで分析業務を実施している例も少なくない. しかしデータ量の増大に伴ってバッチ処理にかかる時間と容量も増大しており, 夜間バッチや週末バッチなどという言葉ができていて, このままだと業務時間に侵食するのも遠くはない. ストレージ容量の点でいっても厳しくなっており, 解決策が必要になってきた.

そこで, バッチ処理のようにストレージに溜め込み逐次処理をするのではなく, そのまま流動的に処理を行う”ストリーム処理”が必要となってきた. そんな中, 2011 年頃に Apache Storm という分散ストリーム処理プラットフォームがリリースされてから, 現 2018 年にはストリームデータ処理プラットフォーム群雄割拠の時代になった. それでは, 分散ストリーム処理プラットフォームとはどういうものなのか, それによってどう今の状況が変わるのか.

第2章　準備

2.1　分散ストリーム処理プラットフォーム

分散ストリーム処理プラットフォームとは, 連続的に発生し続けるデータをリアルタイムに解析, 分析等の処理を行い続けるプラットフォームのことを指す. 使われる用途としての例として, 課金処理, ライブ費用見積もり, 不正, 異常検出, 不正検出結果復旧などが挙げられる. 構造としては, データ発生元, メッセージパス, ストリーム処理基盤, データ利用先があり, データ発生元から発生したデータを一時的にメッセージパスにデータをバッファリングし, バッファリングされたデータを処理基盤が取得してストリーム処理を行うという流れになる.

そのプラットフォームの中で, データ収集系, データフロー系, ストリームプロセッシング系という様々な種類のプラットフォームが提供されている. データ収集系とは, ログやイベントを収集するようなもので, 代表的なものは Kafka Streams. データフロー系とは, イベントデータに対してリアルタイムに加工をするもので, 代表的なものは Nifi. ストリームプロセッシング系は, 汎用的なストリーム処理が可能なもので, Apache Storm, Spark Streaming, Apache Flink が人気であり, 本研究ではこのストリームプロセッシング系に着目していく.

複数選択肢がある中で, プラットフォームを選ぶポイントとして

- 性能と耐故障性
- Single-Event か Micro-Batch
- Streaming+Batch
- プログラミングモデル
- 運用性

の五つの要素が挙げられる. 性能と耐故障性とは, リアルタイム処理を行うストリーム処理には性能が重要視されるのはいうまでもない. 加えて, 耐故障性も重要で, 故障が起こった際に, At Least Once(少なくとも一度は処理する) なのか, Exactly Once(必ず一度だけ処理する) といったメッセージが到達する信頼性の違いがプラットフォームによって存在する. 障害が発生した時のプログラミングモデルもプロダクトによって異なるが, データ収集や保存の方式によってどこで信頼性を担保するかも変わるので, システム全体のアーキテクチャを踏まえて検討する必要がある. Single-Event か Micro-Batch は, イベントサイズのことで, Single-Event がイベントを 1 つずつ処理するので 1 メッセージ毎の遅延が少なくなり, Micro-Batch は小さいバッチを連続実行することでストリーム処理を実現したもので, 短い時間スパンで集計処理など行うことが可能, という違いがある. Streaming + Batch は, ストリーム処理とバッチ処理を両方サポートしているかどうかを指す. システム全体を考えた時にストリーム処理とバッチ処理の両方を利用するケースは多いた

め, バッチ処理もサポートしているかは他のプラットフォームとの違いになる. プログラミングモデルは, 高レベルの API か低レベルの API かのことで, 高レベルのものは分散処理をあまり意識せずに実行を行うことができる. しかしその分, 障害発生時の切り分けは難しくなる. 運用性は, 大量のデータをストリーム処理するとなると, ログを出力して動作を確認するということが難しいため, 管理コンソール画面でスループットやエラー情報などを確認することができるか, 他に障害発生時に他システムへ通知を行えるかなどの機能が充実しているかが重要になる.

ウインドウという概念も実装されていて, 固定長ウインドウ (Tumbling Window), スライディングウインドウ (Sliding Window), セッションウインドウ (Session Window) がある. それぞれ, 固定長ウインドウは 1 時間ごとなどの一定の時間ごとに区切った範囲のウインドウ, スライディングウインドウは毎分, 過去 5 分間分の結果を集計して出力するといった範囲が移動するウインドウ, セッションウインドウは, 一定時間以内にアクセスが連続した場合にそのアクセスを紐づけるという長さが固定されないウインドウを意味する.

常に実行し続けるという性質上, バッチ処理以上に性能の安定性や, いきなりデータ流量が増大した時に死なないことが求められていて, これによって BackPressure という機構が採用されるようになった.

2.1.1 Apache Storm

Twitter 社が公開した OSS で, 分散ストリーム処理エンジンの火付け役的存在である. 実装言語は Clojure で, Clojure が読み書きできないと核心の問題は追えない. 現バージョンは, Storm 1.2.2(04. Jun. 2018)

初期のプロダクトのため, 初期はメッセージ単位の処理であり, レイテンシは低いがスループットが悪く, データ取得側の性能がいいと溢れて失ったりなど, 問題が多かった. [2]

Yahoo や Spotify で利用されていて, 特徴としては. Single-Event として処理する Spout/Bolt 構成の Storm Core と, Micro-Batch として動作する Storm Trident の 2 タイプがある. [1] 故障メッセージの到達信頼性については, Storm Core は At Least Once, Storm Trident は Exactly Once とタイプによって変わる. [4]

2.1.2 Spark Streaming

カルフォルニア大学バークレー校で開発され, 2013 年に公開された Apache Spark のリアルタイム処理エンジン部分のこと. 実装言語は Scala. 現バージョンは Spark 2.4.0(Nov 02, 2018) [3]

バッチ処理フレームワーク Spark 上で小バッチを連続実行し, ストリーム処理を実現する Micro-Batch を実現した. [2]

Hadoop というビッグデータ処理プラットフォームと互換性があり, Hadoop ファミリとしての利用例が増えている. メッセージの信頼性は, Exactly Once に対応しているとあるが, 耐故障性を考慮すると At Least Once に当たる. [1] 特徴として, Spark エコシステム上で機械学習ライブラリの利用, SQL によるデータ操作, 開発手法も同様のものが利用可能が挙げられる. [2]

2.1.3 Apache Flink

ベルリン工科大学およびヨーロッパのいくつかの大学とともに発足した成層圏研究プロジェクトが元となっており, 2014 年に OSS として公開された [5]. 実装言語は Scala. 現バージョンは Apache Flink 1.7.1(21 Dec 2018)

特徴としては, バッチおよびストリーム処理の両方をサポートする. 他に耐故障性に優れ, Accurual Failure Detector が採用されている. 障害対応のための状態保存として, 効率のいい分散スナップショット方式を使用しており, 自動的に各コンポーネントの状態を保存するため, 障害が発生しても自動復旧して処置を継続可能になっている. [2] 加えて, 高レベルの API と低レベルの API の両方を提供しているため, 簡易なものは簡単に組むことができる. メッセージの到達信頼度は Exactly Once. [4] 他のプラットフォームと明確に違う点として, データのフロー制御の BackPressure の設定値を変えることができる.

2.1.4 3 つのプラットフォームの比較

同じストリームプロセッシング系のプラットフォームとして Apache Storm, Spark Streaming, Apache Flink の 3 つはよく比較される. わかりやすく表にしたものを表 2.1 に示す.

Yahoo!がこの 3 つを対象としたベンチマークを行なっており, テスト対処のアプリケーションは広告に関するもので, 100 のキャンペーンがそれぞれ 10 の広告を持つ構成で, 5 つの Kafka ノードを使用して生成された JSON イベントがデシリアライズされ, フィルタを通過し, 関連するキャンペーンと統合された上で Redis ノードに格納されるというものだった. Kafka は生成するイベント数を 50K/秒から 170K/秒まで, 10 刻みに変更可能になっていた. それぞれのイベント送出率においてタプルが完全に処理されるのに必要な遅延率を比較したところ, Flink と Storm の動作には類似点があり, いずれもレイテンシが指数的に増加する場合の遅延率は 99 パーセントまでほぼ直線的に変化したそうだ. Storm0.10.0 は, 135K/秒のイベントレート以降のデータを処理できず, また, Ack を備えた Storm0.110 では, 150K/秒のデータ処理時に重大な問題があったらしい. Ack を無効にした場合の Storm0.11.0 のパフォーマンスは良好で, Flink を凌駕した. しかし, Yahoo!によると, ”Ack を無効にした状態では, タプルエラーの通知や処理も無効になる”そうだ. Yahoo!のテストでの Spark の成績は BackPressure なしで 70 秒, BackPressure ありで 120 秒と, いずれも 1 秒未満であった Flink と Storm に比較するとかなり見劣りするものであったそう. [6]

夏まで, Spark Streaming を触っていたが, 日本語の資料が少なく, サンプルもわかりずらく, うまく動かせているのかもよくわからなかった. その後, Apache Flink になったが, Flink は日本語の資料もまだ多く, チュートリアルも比較的わかり易かったので, 動かしやすかった. 難点があったとすれば, コンパイルの時間が非常に長いということである. Storm は触ったこともない.

スループットに関しては Apche Flink が一番であるというものが調査段階では多く見られた. それぞれ長所があるが, Apache Flink が人気があるのも触ってから頷ける. やはり, チュートリアルのわかりやすさ, 実装のしやすさは開発者にとっては重要なファクターになるのだろうと, 今回を通して学んだ.

表 2.1: ストリームプロセッシング系プラットフォームの性能比較 [4]

	Apache Storm Core	Apache Storm Trident	Spark Streaming	Apache Flink
実装言語	Clojure	Clojure	Scala	Scala
開発言語	Java,Clojure, Scala,Rython,Ruby	Java,Clojure, Scala,Python	Java,Scala, Python, R	Java,Scala, Python
レイテンシ	とても低	低	低	とても低
スループット	高	高	中	高
到達性保証	At Least Once	Exactly Once	At Least Once	Exactly Once
Single-Event or Micro-Batch	Single-Event	Micro-Batch	Micro-Batch	Single-Event
Streaming + Batch	Streaming	Streaming	Streaming + Batch	Streaming + Batch
API	低レベル Compositional	高レベル Declarative	高レベル Declarative	高レベル Declarative
ウインドウ処理	Sliding, Tumbling	Sliding, Tumbling	Sliding	Sliding, Tumbling
BackPressure	変更不可	変更不可	変更不可	変更可

2.2 データフロー制御

フロー制御は, 受信ホストの処理能力に合わせて, 発信ホストのデータ量を調整する機能を指す. フロー制御がないと, 送信側は受信側の許容量を考えずにデータを流し続けるため, 受信側がデータを取りこぼす恐れが生じる. そのため, TCP ではウインドウ制御により, 受信側が受信可能なデータ量を送信側へ通知してデータ量を調整する仕組みが採用されている.

TCP でのデータ転送にはコネクションの確立が必要となる. コネクションの確立には, スリーウェイハンドシェイクと呼ばれる手法が用いられる. スリーウェイハンドシェイクは最初に SYN を送り, 次に ACK を送り互いに確認が取れたらコネクションの確立となる. クライアント側から順に【(クライアント)SYN を送る→ (宛先)ACK+SYN を送る→ (クライアント)ACK を送る】といった順に応答確認を取る.

TCP ではスリーウェイハンドシェイクでコネクションが確立でき次第, セグメントの送信をするが, セグメントは分割して送られる. [7] その分割したものを一気に送ってもデータを保留できるバッファが溢れてしまうことがある. これをバッファオーバーフローという. このバッファオーバーフローを防ぐためにウインドウ制御というものを使う.

2.2.1 スライディングウインドウ

スライディングウインドウは, ウインドウ制御の種類の一つ. 「一つのメッセージを送信したらそれへの確認応答を待つ」という最もな単純な手法では, ネットワークの伝播遅延が大きいと送信側に待ち時間が必要なため, 転送のスループットが大幅に下がる. この欠点を回避するため, スライディングウインドウは, それぞれの応答確認を待たずに複数のセグメントを送信する.

バッファの容量はそれぞれ宛先によって異なる. スリーウェイハンドシェイクでは, ACK 送信後の SYN が帰ってきたときに付随したウインドウサイズ (バッファの容量) を知ることができる. その容量によって送るセグメントを変えて送信していく. [7] そうなると宛先のウインドウサイズ分のデータしか送られず, 処理が終わったらまたウインドウサイズいっぱいまでデータが来るので, まるでウインドウがスライドしているように見えることからスライディングウインドウという.

応答確認を都度実施していると遅くなるので, ウインドウサイズ分だけまとめて確認応答をすることで, オーバーヘッドを少なくして高速化することができる.

2.2.2 BackPressure

BackPressure とは, 3com が開発した LAN スイッチ向けフロー制御で, LAN スイッチ向けフロー制御での BackPressure の仕組みは, トラフィックの集中によってスイッチングハブの受信バッファが溢れそうになった場合, データ送信元ノードにコリジョン信号を送ることでデータ送信の送信速度を低下, あるいは一時中断させるというものになっている.[10]

これをデータのフロー制御として応用した. 非同期のメッセージパッシングでコンポーネント間の通信を行うとき, 処理が早い Producer が処理が遅い Consumer に向けてデータを送信し続けた場合, Consumer 側のバッファが溢れてデータの損失の恐れがある. よって, BackPressure は, Consumer から Producer に対して, 受け入れ可能な個数を通知し, Producer に当たるコンポーネントは, Consumer の速度に合わせてメッセージを送信したり, 間に合わない分は捨てたりなどの

対策を行う. [9] これによって各コンポーネントが速度を調整してバランスをとって, 上流のスループットを犠牲にして, システム全体の安定を図るような機構が BackPressure のシステムになる.

Apache Flink 内でも BackPressure 機構は採用されており, デフォルトの設定では, ジョブマネージャが 50 ミリ秒ごとにキュー中の 100 個のタスクを見て, 100 個のタスクのうちどの程度処理されずに残ったかで BackPressure の強度を判定する. 残った個数/100 が Ratio になる.

- OK : $0 <= Ratio <= 0.10$
- LOW : $0.10 < Ratio <= 0.5$
- $HIGH$: $0.5 < Ratio <= 1.0$

このように 100 個のうち, 10 個以下の場合, ウェブインターフェースに"OK"という表示とともに, BackPressure なしで処理される. 同様に 11 個以上 50 個以下の場合は"LOW"という表示で受け入れ速度を遅めるように BackPressure をかけ, 51 個以上 100 個までの場合は"HIGH"と表示され, 上流へ送信を一旦止めるよう BackPressure をかける.

そして, デフォルトでは, といった通り, この BackPressure をかける判定をするサンプル時間, サンプル個数, そしてリフレッシュするインターバルの設定が変更可能になっている. 他のプラットフォームでは, BackPressure を使うか使わないかの TRUE/FALSE しか設定できない.

設定の変更をする場合は, "flink-1.6.1/flink-core/src/main/java/org/apache/flink/configuration/WebOption.java"の, "BACKPRESSURE_REFRESH_INTERVAL = ... "でリサンプリングのためのリフレッシュをする間隔を変更でき, "BACKPRESSURE_NUM_SAMPLES = ..." でサンプリングするタスクの個数を変更することができる. その下の"BACKPRESSURE _DELAY = ..." では, BackPressure を作動させるか判断するためのサンプリング間隔を変更することができる. それぞれデフォルト値は, 1 分, 100 個, 0.05 秒となっている. そこの部分のソースを付録に載せておく.

第3章　問題点と研究目的

3.1　問題点

関連研究で Apache Flink を用いて BackPressure に関する研究はあるが, Apache Flink の BackPressure 機構の挙動が, 動作環境に対してどのような依存関係があるのかはまだ明らかになっていない. 例えば, 何かプログラムを動かしたときに BackPressure が HIGH になっていても, 対応できない環境があるということが考えられる.

3.2　目的

本研究の目的として, 動作環境によって BackPressure がどのような挙動をするのかの解析を行うこと, が目的となる. 具体的には, Apache Flink 1.6.1 を使用して, Apache Flink がチュートリアル用として提供している”SocketWindowWordCount.jar”に変更を加える. そして Source → Flow → Sink の系を作り, そこの Flow 部分で Delay を実装し, Contention Aware Metrics[11] を用いて, Delay を起こさないときと起こしたときのスループットを計測する実験を行い, ネットワークに対して Flow 部分の BackPressure は, どのような影響を与えるのかを明らかにする.

第4章　実験

4.1　実験環境

実験環境として, Apache Flink(flink-1.6.1), Apache Maven(apache-maven-3.5.4) を使用した.

4.2　実験概要

Apache Flink のチュートリアルで使う”SocketWindowWordCount.jar”に変更を加えて, Contention Aware Metrics[11] を使って BackPressure をコントロールした時のスループットを計測する.

まず Apache Flink のソースダウンロードする. このパッケージを本研究では Java で扱うので, そのプロジェクト用管理ツールである Apache Maven の準備をしておく. Apache Maven は pom.xml というプロジェクトファイルを作り, そこで宣言されている実行が行える. Apache Flink では pom.xml ファイルも用意してあるので, ダウンロードした”flink-1.6.1”のディレクトリに移り, pom.xml があるのを確認して, コンパイルをする. ”$ mvn clean package -DskipTests”でコンパイルを始める. -DskipTests を入れることで多少コンパイル時間を短くできるが, それでも 40 分ほどかかる. コンパイルが完了したら, ディレクトリ内に”/build-target”というエイリアスの実行用フォルダができる.

”SocketWindowWordCount.jar”はどういうプログラムかというと, 5 秒ごとにローカルサーバーに入っている文字を読んでカウントして, 文字列ごとにその数を出力するものになっている. ”SocketWindowWordCount.jar”の使い方は, ”./flink-1.6.1/build-target/” に移動し, ”./bin/start-cluster.sh”でローカルのクラスタを起動する. 別窓でカウントする文字を入れる用のローカルサーバーを”$ nc -l 9000”で立ち上げる. そして今回の実験では, なるべく大きい処理を見たいので, このローカルサーバーに 2GB ほどあるテキストファイルを入れる. そのため, nc -l 9000 ¡ asfi.txt”という風に asfi.txt を突っ込む. なので, この作業は/bluid-target ディレクトリの外で構わない. そして, WordCount のプログラムを先ほどのポートと合わせて実行する. ”$./bin/flink run examples/streaming/SocketWindowWordCount.jar –port 9000” これで実行できるのだが, 今回は BackPressure の挙動を確認したいので, ”$./bin/flink run -p 2 examples/streaming/SocketWindowWord Count.jar –port 9000” と, -p 2 を加える. これは Parallelism の p でこれを 2 にすることで SocketWindowWordCount が 2 つ同時に動くことになる. こうすることで処理が重くなり, BackPressure の作動を確認できる. あと, ここでパラレルにするため, ローカルクラスタをもう一つ”./bin/start-cluster.sh”で起動されておかなくてはならない. この実行中の様子を localhost:8081 にアクセスすると, ウェブインターフェースで確認することができる. そこで走っているジョブをクリックすると,

Source: Socket Stream → Flat Map → Window(TumblingProcessingTimeWindows(5000), ProcessingTimeTrigger, ReduceFunction$1, PassThroughWindowFunction) → Sink: Print to Std. Out

という図が見える.

4.2.1 Contention Aware Metrics

Akka で用いられるストリーム処理のパイプラインを表す Source, Flow, Sink というものがある. Source は一つの出力を持つ処理単位で, 処理における入力のノードに当たる. Sink は一つの入力を持つ処理単位で, 処理における出力のノードに当たる. Flow は, Source と SInk の間で, 1 つの入力と 1 つの出力を持つ処理単位で, 入力ノードと出力ノードの中継処理ノードに当たる. Contention Aware Metrics[11] では, この Source → Flow と Flow → Sink のネットワークの速さを 1 として, Flow に λ というパラメータを入れる. この λ は, ネットワークに対する Flow の相対処理速度を表し, λ が 1.5 など, 1.0 より大きい場合は, ネットワークに対して処理が遅い中間ノードが存在することを表し, λ が 0.5 など 1.0 より小さいと場合は, 中間処理ノードに対して, ネットワークの速度が遅いことを表す. これを実装するため, "SocketWindowWordCount.jar"のソースである"WordCount.java"に Thread.sleep() を入れて, Flat Map で入力された文字を整えて, WordCount に渡すところで Delay を発生させてる. それによるスループットを測り, Delay を発生させない時のスループットと比較する. 発生させる Delay は BackPressure が作動を決定する 0.05 秒にする. λ の計算方法は,

1 : λ = Thread.sleep(0) の時のスループット : Thread.sleep(0.05) の時のスループット

第5章　まとめ

5.1　まとめ

分散ストリーム処理プラットフォーム群雄割拠の時代の中, データフロー制御の BackPressure 機構の設定を変更できる Apache Flink に着目した. その BackPressure 機構は動作環境に対してどのような依存関係がるのかが明らかになっていなかったため, Apache Flink のサンプルとして提供される”WordCount.java”に処理を遅らせる Delay を実装することによって, Contention Aware Metrics の λ というパラメータを使って BackPressure によるネットワークと中間処理ノードの関係を計測した.

5.2　今後の課題

せっかく BackPressure の設定を変更できる Apache Flink を選択したにも関わらず, BackPressure の設定値を変更せずに計測したため, BackPressure の設定値を変えることによってどこまでスループットを高めることができたり, 高いスループットの状態での環境依存を見ていくことが今後の課題となる. 他に今回は Flow の中身が FlatMap と WordCount だけだが, もっと複雑な処理を行うものになった場合のスループットの変化も見たい.

謝 辞

林原先生, 不出来な自分を 3 年も面倒見てくださって感謝しています. 先生の研究室の卒業として何か貢献できるよう社会に入っても精進してまいります,

今江さん, 佑さん, 嶋田さん, 杉原さん, アスフィーさん, 中川さん, 研究やスライド作りなどたくさんお世話になりました. 本当にありがとうございました.

参 考 文 献

[1] Qiita「分散ストリーム処理エンジンあれこれ」(最終閲覧日：2019 年 1 月 19 日) https://qiita.com/takanorig/items/aaa4f116d1564ec20dd3

[2] Qiita「ストリーム処理とは何か？＋ 2016 年の出来事」(最終閲覧日：2019 年 1 月 19 日) https://qiita.com/kimutansk/items/60e48ec15e954fa95e1c

[3] Apache Spark Streaming「Spark Streaming makes it easy to build scalable fault-tolerant streaming applications.」(最終閲覧日：2019 年 1 月 19 日) https://spark.apache.org/streaming/

[4] Slide Share「IoT 時代におけるストリームデータ処理と急成長の Apache Flink」(最終閲覧日：2019 年 1 月 19 日) https://www.slideshare.net/takanorig/iot-apache-flink

[5] Apache Flink「Apache Flink - Stateful Computations over Data Streams」(最終閲覧日：2019 年 1 月 19 日) https://flink.apache.org/

[6] 著者:Abel Avram 訳:吉田 英人「Yahoo!が Apache Flink, Spark, Storm のベンチマークを実施」InfoQ 投稿日:2016 年 2 月 2 日 (最終閲覧日：2019 年 1 月 19 日) https://www.infoq.com/jp/news/2016/02/yahoo-flink-spark-storm

[7] Qiita「OSI 参照モデルまとめ」(最終閲覧日：2019 年 1 月 19 日) https://qiita.com/tatsuya4150/items/474b60beed0c04d5d999

[8] ASCII.jp × TECH「帯域を効率的に利用する TCP の仕組みとは？」(最終閲覧日：2019 年 1 月 19 日) http://ascii.jp/elem/000/000/444/444152/

[9] SlideShare「Reactive Systems と Back Pressure」(最終閲覧日：2019 年 1 月 19 日) https://www.slideshare.net/zoetrope/reactive-systems-back-pressure

[10] weblio 辞書「Back Pressure」(最終閲覧日：2019 年 1 月 19 日) https://www.weblio.jp/content/Back+Pressure

[11] P.Urban, X. Defego, A.Shiper,“Contention-aware metrics for distributed algorithms: comparison of atomic broadcast algorithms”, ICCCN 2000

付 録A　WebOption.java

”flink-1.6.1/flink-core/src/main/java/org/apache/ flink/configuration/WebOption.java”の, Back-Pressure 機構の設定部分

```
/**
 * Time after which cached stats are cleaned up if not accessed.
 * .defaultValue(10 * 60 * 1000) 10min
 */
public static final ConfigOption<Integer> BACKPRESSURE_CLEANUP_INTERVAL =
key("web.backpressure.cleanup-interval")
.defaultValue(10 * 60 * 1000)
.withDeprecatedKeys("jobmanager.web.backpressure.cleanup-interval");

/**
 * Time after which available stats are deprecated and need to be refreshed (by resampling).
 *  .defaultValue(60 * 1000) 1min
 */
public static final ConfigOption<Integer> BACKPRESSURE_REFRESH_INTERVAL =
key("web.backpressure.refresh-interval")
.defaultValue(60 * 1000)
.withDeprecatedKeys("jobmanager.web.backpressure.refresh-interval");

/**
 * Number of stack trace samples to take to determine back pressure.
 *  .defaultValue(100)
 */
public static final ConfigOption<Integer> BACKPRESSURE_NUM_SAMPLES =
key("web.backpressure.num-samples")
.defaultValue(100)
.withDeprecatedKeys("jobmanager.web.backpressure.num-samples");

/**
 * Delay between stack trace samples to determine back pressure.
 * .defaultValue(50) 0.05sec
 */
public static final ConfigOption<Integer> BACKPRESSURE_DELAY =
key("web.backpressure.delay-between-samples")
.defaultValue(50)
.withDeprecatedKeys("jobmanager.web.backpressure.delay-between-samples");
```