情報処理学会論文誌　コンピューティングシステム　Vol.7　No.1　35–45 (Mar. 2014)

AspectJを用いたFault-Injectionによる Hadoop MapReduceの耐故障処理に関する性能評価

中川 洋介[1,†1,a)]　櫻井 孝平[2]　清水 裕亮[1,†1]　山根 智[2]

受付日 2013年7月25日, 採録日 2013年12月18日

概要：近年，ネットワークサービスの利用形態の１つとして，クラウドコンピューティングが注目を集めている．そのクラウドの内部では，複数の要素によって構成される分散システムによって処理が行われており，主にMapReduceなどの分散処理フレームワークにより処理を複数のサーバに分散させている．大規模な分散システムでは，ネットワーク通信障害をはじめとした様々な障害が発生する可能性があるため，システムには高いフォールトトレランス（耐故障性）が要求され，十分な性能評価を行ったうえでシステムを提供する必要がある．しかし分散システムの耐故障処理の動作は複雑なものとなっているため，その性能を評価する効率的な手法として，ランタイムに評価を行う手法が有効であると考えられる．本研究では，Apache Hadoopを対象とし，AspectJを用いて，MapReduceの動作中に考えられる様々な障害を発生させ（Fault-Injection），同時にシステムの動作を監視するモニタを実装する手法を提案する．この手法を用いて，MapReduceの障害発生後の動作をランタイムに解析することにより，耐故障処理の効率的な性能評価を行うことが可能である．今回，障害発生下におけるMapReduceアプリケーションの実行や，モニタにより生成されたトレースにより，Hadoop MapReduceの耐故障性に関する評価実験を行った．

キーワード：分散システム，MapReduce，Hadoop，フォールトトレランス

Fault-tolerant Performance Evaluation of Hadoop MapReduce by Fault-Injection Using AspectJ

Yousuke Nakagawa[1,†1,a)]　Kohei Sakurai[2]　Yusuke Shimizu[1,†1]　Satoshi Yamane[2]

Received: July 25, 2013, Accepted: December 18, 2013

Abstract: Recently, cloud computing has attracted a lot of attention as one of the usage patterns of network services. Within the cloud, the process is conducted by the distributed system that is constituted by plural elements, thereby distribute the process across multiple servers by distributed processing frameworks such as MapReduce mainly. Because there is a possibility of various failure including network communication failure in a large-scale distributed system, high fault tolerance is required for the system, and it is necessary to provide the system after having enough performance evaluation. However, the operation of the fault tolerant processing of the distributed system is complex, therefore it is considered that run time evaluation is effective to evaluate its performance effectively. In this study, we propose a method to inject various failure under processing MapReduce, and to monitor the operation of system at the same time using AspectJ. Using this method, it is possible to analyze the operation after occurring failure of MapReduce at run time, and evaluate fault-tolerant performance effectively. In this paper, by execution of MapReduce application under failure, and traces that are generated by the monitor, we experiment about fault-tolerant performance of Hadoop MapReduce.

Keywords: distributed system, MapReduce, Hadoop, fault tolerance

[1] 金沢大学理工学域
College of Science and Engineering, Kanazawa University, Kanazawa, Ishikawa 920–1192, Japan
[2] 金沢大学理工研究域
Institute of Science and Engineering, Kanazawa University, Kanazawa, Ishikawa 920–1192, Japan
[†1] 現在，金沢大学大学院自然科学研究科
Presently with Graduate School of Natural Science and Technology, Kanazawa University
[a)] ynakagawa@csl.ec.t.kanazawa-u.ac.jp

1. はじめに

1.1 背景と目的

近年，スマートフォンやソーシャル・ネットワーキング・サービスなどの普及にともない，クラウドコンピューティングと呼ばれるネットワーク上でのコンピュータの利用形態が注目されている．クラウドサービスを利用して膨大な

データへのアクセスが容易に行えるようになったことによって，それらのサービスを提供する企業や利用者が急速に増加している．

そのようなサービスを実現しているのは，複数の要素によって構成される**分散システム** [1] による分散処理である．大規模な分散システムでは，主に **MapReduce** [2] と呼ばれる Google が開発した分散プログラミングフレームワークを用いることで処理を複数のサーバに分散させており，タスクを実行するサーバの台数を増やすことで処理性能をスケールアウト [3] させることが可能となっている．この MapReduce を実装した，大規模データの分散処理フレームワークの普及がきっかけで，現在では様々な企業で MapReduce の技術を用いたデータ処理などが行われている．

このスケールアウト性を活かすために大規模な分散環境で処理を実行する場合，ネットワーク障害をはじめとした様々な障害が発生する可能性が高くなる．なぜなら，多数のサーバを用いることで，各プロセスやノード間の通信によりネットワーク負荷が増大してしまうためである．利用者に安定したサービスを提供するためには，システムの高い**フォールトトレランス（耐故障性）**が要求され，十分に性能評価を行ったうえでシステムを提供する必要がある．Hadoop は，MapReduce の動作中に障害が発生すると，各ノード内にあるデータのレプリケーションなどを用いて復旧動作を行い，ある程度の障害には耐えられる設計となっている．しかし，MapReduce を実行する分散システムにおいて，各ノード間での一貫性のあるデータの保持やメッセージ通信などを行う必要があり，システムの障害検知や自己管理の動作は複雑なものとなっている．したがって，そのような耐故障動作には，まだ改善すべき部分が存在していると考えられ，効率的な性能評価手法が求められている．

我々は，分散システムを対象とした耐故障処理に関する性能評価手法として，実際に対象となるシステムに障害を発生させ，実行時の振舞いを監視する手法を提案する．具体的には，大規模分散処理フレームワークである **Apache Hadoop** [4] を対象として，MapReduce の動作における各ノード間のメッセージ通信などに関する例外を発生させることにより，仮想的に障害を発生させ（**Fault-Injection** [5]（以下，FI と略す）），同時にモニタによりシステムの動作の監視を行う．それぞれの実装には，Java を拡張したアスペクト指向言語である **AspectJ** [6] を用いた．この手法を用いて，MapReduce の障害発生後の動作をランタイムに解析することにより，耐故障処理の効率的な性能評価することが可能である．最終的には，優れた並列処理フレームワークである MapReduce の処理のさらなる効率化へつなげることを目的とする．今回，この手法を用いて，モニタから生成されたメソッド実行トレースによる解析や，障害発生下における MapReduce アプリケーションの実行時間の比較などの実験を行うことで，性能評価を行った．

1.2　関連研究

分散システムへの FI に関する既存研究として，ハードウェアとソフトウェアの両方の側面からハイブリッドに障害を発生させる手法 [12] や，障害発生に関するポリシを設定することによって，検証したい項目に合わせて障害を組み合わせるツール [13] などがある．これらは分散システム全体に対して障害を発生させるものであるが，本研究では，MapReduce の耐故障性に着目し，障害を発生させた後の耐故障性能評価に重点を置いている．

MapReduce 動作の検証における既存研究として，MapReduce の処理がネットワークに対して与える負荷の分析 [3] があげられる．この研究において，MapReduce によるネットワークリソースの競合や，トランスポートプロトコルの挙動などが，MapReduce 自体の性能に影響を与えていると述べられている．しかし，障害が発生した場合，処理性能にどのような影響が生じるのかについては言及されておらず，システムの耐故障動作に関する解析も行う必要があると考えられる．

また Hadoop には，アプリケーションのロジックに障害を発生させる FI フレームワークが備わっている．これは，AspectJ によって実装されており，ユーザが作成した MapReduce アプリケーションに対して耐故障性の評価を行うものである．我々は，アプリケーションではなく MapReduce のフレームワーク自体に対して性能評価を行うことを目的としており，新たに AspectJ を用いて，FI とシステムを監視するためのモニタの実装を行った．

1.3　論文の構成

以下，2 章に本手法を実装するうえで基となる Hadoop MapReduce について述べ，MapReduce の動作中に発生しうる障害について述べる．3 章において，耐故障性能の評価手法の提案とその実装方法について述べ，4 章で評価実験を行い，5 章で本研究をまとめる．

2.　Hadoop MapReduce

本章では，2.1 節で，MapReduce の耐故障性能の評価を行ううえで基盤技術となる，Hadoop の MapReduce 動作について簡単に述べたうえで，2.2 節で MapReduce 動作中に発生しうる障害について述べる．

2.1　Hadoop の MapReduce 動作

MapReduce とは，データの処理を Map フェーズと Reduce フェーズの 2 つのフェーズに分け，大規模なデータであっても，クラスタ群内の TaskTracker によって並列分散させて処理を行うことができるフレームワークである

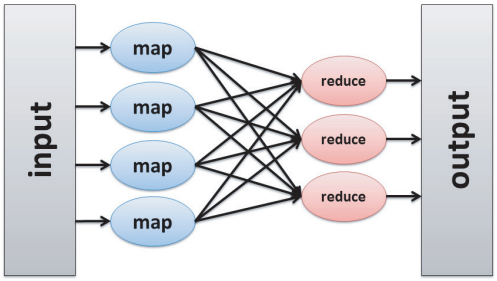


図 1　Map/Reduce 処理
Fig. 1　Map/Reduce process.

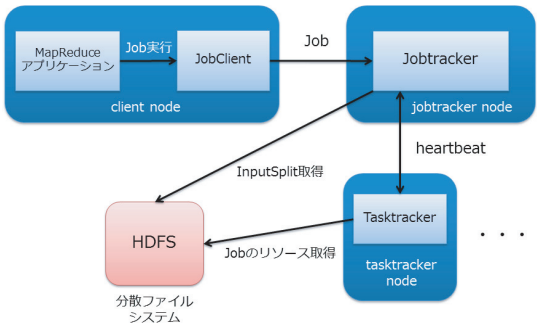


図 2　MapReduce の動作の概要
Fig. 2　Overview of the operation of MapReduce.

(図 1).

Hadoop の MapReduce 動作は，主に以下の部品により構成されており，それぞれの関係を図 2 に表す．

- **JobClient**
  MapReduce ジョブをマスタノードに投入する．
- **JobTracker**
  クラスタのうち指定したスレーブノードへ MapReduce タスクを送る．
- **TaskTracker**
  ネットワーク環境下において，MapReduce タスクを起動し，実行する．
- **HDFS（分散ファイルシステム）**
  ジョブに関するファイルを各要素と共有するためのファイルシステム．

これらの中で MapReduce 動作の根幹を担っているのが JobTracker と TaskTracker である．

JobTracker は分散環境下において TaskTracker に MapReduce ジョブを投入し，管理する中心的存在（**マスタノード**）である．JobTracker は submitJob メソッドにおいて，データに近く，使用可能なスロットを持つ TaskTracker ノードを選択し，ジョブを実行させる．また，タスク失敗率の高い TaskTracker をブラックリストに登録する．

スレーブノード群に属する各 TaskTracker は，JobTrakcer からタスクを受け取ると，HDFS 上からジョブの jar ファイルや，アプリケーションの実行に必要なファイルなどをローカルファイルシステムにコピーし，TaskRunner と呼ばれる，タスクを実行するインスタンスを生成する．そして，その TaskRunner は JVM を起動し，各 Map/Reduce タスクを実行させる（図 3）．JVM は，タスクの実行中に，親である TaskTracker にタスクの進行状況を定期的に送っている．

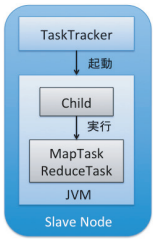


図 3　TaskTracker ノードの内部動作
Fig. 3　Internal operation of TaskTracker node.

また，TaskTracker 自身もループ動作によって定期的に heartbeat シグナルを JobTracker に送ることによって，JobTracker に生きていることを知らせている．JobTracker は，TaskTracker から定義された時間以内に heartbeat シグナルが送られて来なければ，その TaskTracker に割り当てたタスクが失敗したものと見なし，他の TaskTracker に依頼する．

大規模な分散環境で安定した処理を行うために，これらの MapReduce の動作やタスクスケジューリングは，優れた耐故障性と高効率性を満たしたものである必要がある．

2.2　MapReduce 動作中に発生しうる障害

2.1 節で説明した MapReduce の動作中には，ユーザ作成の MapReduce アプリケーション中のバグによる障害や，各ノード間との通信障害など，様々な障害が発生する可能性がある．Hadoop には，それらの障害に対処するフレームワークが備わっており，ある程度の障害が発生してもジョブを完了させることができる．以下，JobTracker および TaskTracker において発生しうる障害と，その後の Hadoop の動作について述べる．

2.2.1　JobTracker における障害

JobTracker は MapReduce ジョブを管理しているマスタノードであるため，単一障害点[*1]である．そのため，JobTracker に障害が発生すると，ジョブを完了させることができない．これを回避するため，これまでに **Hadoop HA**（**high availability**）[8] によるマスタノードの二重化や，Hadoop のプラグインである **Apache Zookeeper** [9] による冗長化などの手法が考えられてきた．

Hadoop HA は，DRBD（Distributed Replicated Block Device）[10] と Heartbeat [11] [*2] を組み合わせた構成をしており，処理を実行するサーバとは別に待機サーバを設ける

[*1] Single Point of Failure．障害が発生すると，システム全体に障害が発生してしまう箇所．
[*2] JobTracker，TaskTracker 間の heartbeat とは別．

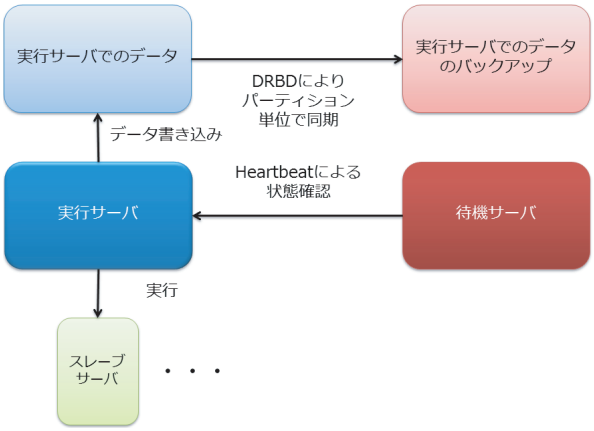


図 4　HA によるマスタノードの二重化
Fig. 4　Duplication of the master node by HA.

ことで，高可用性を実現している（図 4）．これにより，マスタノードに障害が発生しても，待機サーバにジョブの実行を引き継がせることで，ジョブを継続して実行することが可能である．

2.2.2　TaskTracker における障害

TaskTracker は，実際に MapReduce タスクを実行し，ノード間などの通信をかなりの頻度で行っているため，JobTracker と比べて通信障害などの影響を受けやすい．ユーザが作成した MapReduce プログラムのコードが実行時に例外を投げた場合は，子 JVM が親の TaskTracker にタスクの失敗を報告し，TaskTracker はそのタスクの分のスロットを他のタスクの実行のために空ける．また，タスクを実行する子 JVM が突然終了してしまったり，TaskTracker との間の通信に障害が生じることによって，タスクの進行状況の報告ができなくなってしまったりした場合には，TaskTracker はその子 JVM が実行していたタスクが失敗したものと見なす．しかし，多少のタスクの失敗があってもジョブを継続して行うことができる．

TaskTracker に障害が発生し，JobTracker にブラックリスト登録されてしまった場合，JobTracker は対象の TaskTracker をジョブのスケジューリングから除外し，そのタスクを異なる TaskTracker に依頼することで，タスクを再開させる．また障害が発生していない TaskTracker であっても，タスクの失敗率が高ければ，ブラックリストに登録されることがある．

MapReduce の動作の中で，クラスタに属する各 TaskTracker どうしは，発生した障害などに関する情報を通信によって交換しあっていないことが，特徴としてあげられる．特に大規模な MapReduce アプリケーションの場合，各 TaskTracker どうしは各ジョブやタスクに関する膨大な情報を交換する必要があるため，ネットワークトラフィックが急激に増加してしまう．そのため，各 TaskTracker はマスタである JobTracker を通して，情報交換を行っている．

3.　耐故障性能評価手法の提案と実装

本章では，3.1 節で Hadoop における MapReduce 処理の有効な性能評価手法として，AspectJ を用いて FI およびシステムを監視するモニタによる手法を提案し，3.2 節でその実装について述べる．

3.1　提案手法

Hadoop の耐故障性を評価するうえで，実際に障害を発生させ，ランタイムに解析を行うことが有効であると考えられるが，その際にシステムの動作に与える影響を可能な限り少なくすることが重要である．本研究では，AspectJ を用いて，例外を投げることで仮想的に障害を発生させる．ソフトウェアへの Fault-Injection は，Compile-time Injection と Runtime Injection の 2 つに分けられるが，本手法はコンパイル時にアスペクトを織り込むことで障害を発生させるため，Compile-time Injection に分類される [5]．また，AspectJ を用いることにより，システムを監視するモニタの実装も行う [14]．モニタにより，各ノードで実行トレースが生成され，トレースには，実行されたデーモンごとに呼び出されたメソッドがタイムスタンプとともに出力される．Hadoop では，log4j[*3]によりログを取得できるが，このログは各ノードごとに生成されるため，ノード数が増加するに従い障害の原因特定などといった作業は非常に困難となる．しかし，モニタを用いることにより，各ノードのトレースを 1 カ所に集約させることが可能となるため，効率的に解析を行うことができる．

AspectJ のようなアスペクト指向言語を用いることで，ロギングなどといったオブジェクト指向言語だけでは分離できないような処理（横断的関心事）が，モジュール化により実現できる．例外を投げるといった処理もその 1 つである．AspectJ では，プログラムの特定の実行時点（**ジョインポイント**）の中から追加の処理を適用させる時点を**ポイントカット**と呼ぶ記述によって決定する．そして，ポイントカットで選んだジョインポイントに対して，ロギングやデバッグなどの追加的な処理を**アドバイス**として記述することで，対象となるコードに手を加えることなく処理を実行する．

本手法では，以下のすべての条件に合うジョインポイントを選択する．

(1) MapReduce を動作させるうえで中核を担っている JobTracker と TaskTracker の処理．
(2) 障害が発生しやすいと考えられるメッセージ通信を行う実行時点．
(3) 例外処理を行っている．

そして，アドバイス内でコードに **IOException** などの

[*3] Java のロギングユーティリティ
http://logging.apache.org/log4j/1.2/

表 2　発生させる障害とそれぞれのポイントカットとアドバイス
Table 2　Failures and each pointcut and advice.

障害名	ポイントカット	投げる例外
SubmitJobFailure	JobTracker.submitJob(..) 内の addJob(..)	IOException
ValidateJVMFailure	TaskTracker.statusUpdate(..) 内の validateJVM(..)	RemoteException
JvmlaunchTaskFailure	mapred.JvmManager.JvmManagerForType. JvmRunner.run() 内の runChild(..)	RemoteException
LocalizeJobTokenFileFailure	TaskTracker.localizeJobTokenFile(..) 内の FileSystem.getFileStatus(..)	FileNotFoundException
MapOutputServletFailure	TaskTracker.MapOutputServlet.doGet(..) 内の SecureIOUtils.openForRead(..)	RemoteException
TransmitHeartBeatFailure	TaskTracker.transmitHeartBeat(..) 内の InterTrackerProtocol.heartbeat(..)	IOException

表 1　使用したソフトウェアのバージョン
Table 1　Version of the software used.

OS	CentOS release 6.3
Java	1.7.0_09-b05
Hadoop	1.0.3
AspectJ	1.7.1

例外を投げることで耐故障動作に関する評価を行う．これにより，例外処理という実際には実行される機会が少ない部分を検証することが可能となる．また，本手法では実際の Hadoop のソースコードから例外処理を探し出しポイントカットを決定するため，本研究で評価を行う耐故障動作の範囲は，Hadoop 本体での例外発生後の動作であり，言語レベルで認識される障害を対象とする．そのため，(a) 通信遅延やビザンチン障害などの例外発生に至らないような障害や，(b) データ化けやバグなどのユーザのアプリケーション側で対処する必要のあるソフトウェア障害は，本手法では対象としない．なお，(a) についての対応は将来的な課題とする．

モニタは，MapReduce における JobTracker と TaskTracker のメソッド実行トレースを生成するものとし，障害が発生した場合にもトレースが生成される．このトレースと FI を組み合わせることにより，障害が発生した回数や時間などの情報をランタイムに取得し，MapReduce の耐故障処理に関する性能評価を効率的に行うことが可能となる．

3.2　実装

今回実装において用いたソフトウェアのバージョンを，**表 1** に示す．以下，実装を行った FI について述べる．Hadoop の JobTracker と TaskTracker を中心とした MapReduce 処理の中から，実際に例外を投げる箇所とその例外を決定し，ポイントカットおよびアドバイスとした．また，障害を発生させるノードと障害の種類，障害発生確率（0.0–1.0）を，設定ファイルを読み込むことで，任意に設定できるようにし，各アドバイス内で，発生させた障害の種類とそのタイムスタンプをモニタによるトレースとして出力されるようにする．

表 2 に，実装した障害一覧と，それらのポイントカットおよびアドバイスで投げる例外を示す．たとえば **SubmitJobFailure** では **JobTracker** クラスの **submitJob(..)** メソッド内の **addJob(..)** メソッドの呼び出しをポイントカットとし，そのアドバイス内でファイルの入出力関係の例外である **IOException** を投げている．以下，実装を行ったそれぞれの障害について述べる．

3.2.1　JobTracker での FI

- **SubmitJobFailure**
  JobTracker での障害としてまず考えられるのが，TaskTracker へのジョブ投入の失敗である．この障害が発生することにより，JobTracker は TaskTracker にジョブを依頼することができなくなるため，ジョブは失敗に終わることが予想される．
  JobTracker は単一障害点であるため，現状の Hadoop では，JobTracker での障害はそのままジョブ失敗につながると考えられる．しかし 2.2.1 項で述べたように，現在ではこのような問題はほぼ解決されつつある．今回は，マスタサーバの冗長化を行っていない Hadoop を評価の対象としているため，マスタノードに関する耐故障性の評価は行わない．

3.2.2　TaskTracker での FI

図 5 に，TaskTracker における障害実装の概要図を示す．

- **ValidateJVMFailure**
  TaskTracker での障害では，まずはじめにタスクの失敗が考えられる．**ValidateJVMFailure** でのアドバイスでは，**org.apache.hadoop.ipc** で定義されている **RemoteException** を投げている．この例外は **IOException** のサブクラスであり，通信障害が発生した場合に使用される．ポイントカットの **validateJVM** メソッドは，TaskTracker が定期的に JVM から

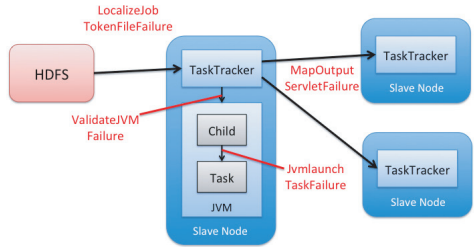


図 5　TaskTracker における障害注入の概要
Fig. 5　Overview of Fault-Injection in TaskTracker.

タスクの進捗報告を確認する際に呼び出されるため，この障害が発生することにより，TaskTracker はタスクの進行状況を取得することができなくなる．

- **JvmlaunchTaskFailure**
  TaskTracker 内で実際にタスクを実行する子 JVM において障害が発生する場合も考えられる．**JvmlaunchTaskFailure** は，TaskTracker が子 JVM を起動するときにメッセージ通信障害を発生させるものである．
- **LocalizeJobTokenFileFailure**
  TaskTracker は JobTracker からジョブを投入された際，そのジョブに関するデータ類を Hadoop の分散ファイルシステムである **HDFS** から取得し，ローカルディレクトリに保持する．**LocalizeJobTokenFileFailure** では，アドバイスで **FileNotFoundException** を投げることで，データをローカライズする際に障害を発生させる．この障害により，TaskTracker はジョブデータを取得することができなくなる．
- **MapOutputServletFailure**
  MapReduce では，Map の出力をそのまま同じノード内の Reduce の入力にするのではなく，他のノードに分散させて処理を行う．そのため，TaskTracker は他の TaskTracker からの Map 出力を受け取る必要があり，その際に障害が発生することが考えられる．**MapOutputServletFailure** では，他の TaskTracker からの Map 出力を得るメソッドをポイントカットとし，通信障害を発生させる．これにより対象の Map タスクは再実行されることが予想できる．

4.　評価実験

本章では，実際に Hadoop に対して 3 章で述べた実装を行ったうえでの，実装した手法の性能評価および MapReduce の耐故障性に関する評価実験を行う．まず 4.1 節で，実験を行う際に用いた環境について述べる．そして 4.2 節で提案手法の評価について述べ，4.3 節で実験について述べる．

4.1　実験環境

実験で用いた計算機のスペックを**表 3** に示す．今回はこの計算機を 6 台用い，1 台をマスタノード，5 台をスレーブノードとした．また，Hadoop を動作させるうえでの，設定ファイルをそれぞれ**表 4**，**表 5** に示す．

表 3　使用した計算機のスペック
Table 3　Specifications of computer used.

CPU	Intel® Core™ i5-3470 Processor
動作周波数	3.20 GHz
コア数	4
RAM 容量	8 GB
ディスク	1 TB SATA HDD（7,200 回転）

表 4　core-site.xml
Table 4　core-site.xml.

設定項目	設定値
io.file.buffer.size	65,536
io.sort.factor	20
io.sort.mb	600
fs.inmemory.size.mb	200

表 5　mapred-site.xml
Table 5　mapred-site.xml.

設定項目	設定値
mapreduce.client.submit.file.replication	2
mapred.tasktracker.map.tasks.maximum	4
mapred.tasktracker.reduce.tasks.maximum	4
mapreduce.reduce.maxattempts	4
mapreduce.reduce.shuffle.connect.timeout	60,000
mapreduce.reduce.shuffle.read.timeout	180,000
mapreduce.task.timeout	600,000
mapred.tasktracker.expiry.interval	60,000

4.2　手法の評価

実装を行った FI およびモニタのアスペクトが，実際に MapReduce の処理に与える影響を調べ，手法の評価を行う．評価には Hadoop のサンプルアプリケーションである TeraSort を用いる．これは任意サイズのデータのソートを行うプログラムであり，その入力には，同じく Hadoop のサンプルアプリケーションである TeraGen を用いて作成したデータを与える．評価手法として，FI およびモニタのアスペクトを織り込んだ場合とそうでない場合において，TeraSort の実行にかかる時間を測定し，比較を行う．入力ファイルのデータサイズは 1 GB，10 GB，100 GB の 3 種類を用いる．

表 6 より，アスペクトを適応した場合とそうでない場合の実行時間の比率（実装しなかった場合/実装した場合）は，1 GB，10 GB，100 GB それぞれにおいて，0.915，0.975，0.942 となっている．この結果から，いずれのファイルサイズにおいてもアスペクトによるオーバヘッドが 10%未満であり，またコンパイル時に静的にアスペクトを織り込ん

表 6　TeraSort にかかる実行時間とスループット
Table 6　Throughput and execution time required for TeraSort.

アスペクトの有無	データサイズ（GB）	実行時間（s）	スループット（MB/s）
アスペクトなし	1	75	13.33
	10	312	32.05
	100	3,541	28.24
アスペクトあり	1	82	12.34
	10	320	31.25
	100	3,758	26.61

でいるため，本手法により動作中に想定しない障害が発生するなどの耐故障性の評価を行ううえでの不具合は発生しないと考えられる．

4.3　障害の発生による耐故障処理の評価

実装を行った FI とモニタを用いて，MapReduce の耐故障処理に関する実験を行う．4.3.1 節では，障害を発生させたうえで生成されたトレースにより，MapReduce の動作解析を行い，4.3.2 節と 4.3.3 節では，障害発生確率および障害を発生させるノード数を変化させることによる実行時間などの比較を行う．これらの実験により，各障害が MapReduce 処理の性能にどのような影響を与えているのか検証する．

4.3.1　トレースによる解析

実装した各障害を障害発生確率を 1.0（必ず障害が発生する）として入力 1 GB の TeraSort を実行し，モニタから生成されたトレースをもとに各障害が発生した前後における MapReduce 動作を監視する．以下では，実際に障害を発生させたノードにおけるメソッド実行トレース（付録参照）を用いて説明する．

- **SubmitJobFailure**
  JobTracker はジョブを実行するにあたり，getQueueAdmins により投入すべきジョブについての情報を取得している．そしてその後にジョブを各 TaskTracker に投入するのだが，その前に実装した SubmitJobFailure が発生しているのが分かる．しかし，障害が発生した後は特別な動作を行っているわけではなく，すぐにクラスタや TaskTracker の情報取得などの通常動作を行っていることが分かる．これはジョブの投入に失敗すると，そのジョブをすぐに放棄し，次のジョブに備えるためであると考えられる．
- **ValidateJVMFailure**
  TaskTracker はタスクを実行するために，まずローカルディレクトリを参照してタスクに関する情報を取得している．そして，タスクをローカルディレクトリから取得した後に，ValidateJVM によって JVM にタスクを実行できるか確認をとっている．ここで障害が発生すると再びタスクに関する情報を取得しながら，何度か JVM への確認を行っていることが分かる．
- **JvmlaunchTaskFailure**
  TaskTracker はタスク実行のための準備を行いタスクを立ち上げているが，そのときに障害が発生すると，TaskTracker はタスク立ち上げを何度も試みていることが分かる．しかし最終的には，そのタスクを実行不可能と判断し，kill している．
  大規模なデータ処理においては，タスク実行に関する障害は多少なりとも発生してしまうものであるため，1 つのタスクの障害に執着してしまうと，全体の処理が遅くなる原因となってしまう．そのため，タスク切り捨てを行うバランスの設定が重要となる．
- **LocalizeJobTokenFileFailure**
  これは，TaskTracker がジョブに関するファイルをローカルディレクトリにコピーする際の障害であるので，最初に障害が発生している．何度かローカライズを試みた後に，タスクは，タスクの進捗状況を管理する TaskInProgress 内の reportDiagnosticInfo によって診断情報を TaskTracker に送っている．TaskTracker は，該当のタスクを失敗と見なし，その後そのタスクは JobTracker によって他の TaskTracker に割り当てられる．
- **MapOutputServletFailure**
  TaskTracker は，JVM での Map タスクの実行を終えた後，その出力を MapOutputServlet により他の TaskTracker に分配する．その際に障害が発生してしまうと，LRU キャッシュの取得などを行いながら，Map 出力の分配をかなりの頻度で行っていることが分かる．そして，何度か行った後，mapOutputLost により出力分配に失敗した後の処理を行っている．失われた Map 出力は，当然 Reduce の入力に用いることができないため，そのタスクは再スケジューリングされる．

このように，モニタから出力されたトレースにより各メソッドの実行状況を監視することで，障害発生前後の MapReduce に関する動作を確認することができる．

4.3.2　障害発生確率の変化による比較

まず，障害を発生させるノードを 1 つに固定し，障害発生確率を変化させていくことによってジョブの実行時間の比較を行う．TaskTracker での各障害において，障害発生確率を 0.0，0.2，0.4，0.6，0.8，1.0 の 6 段階で変化させ，

表 7　各障害の発生回数とメソッド実行回数
Table 7　Method execution times and the number of occurrences of each failure.

障害名	発生回数	メソッド実行回数					
		Master	Slave1	Slave2	Slave3	Slave4	Slave5
障害なし		425	296	273	273	291	274
ValidateJVMFailure	12	649	57	387	358	419	447
JvmlaunchTaskFailure	10	496	33	365	340	338	336
LocalizeJobTokenFileFailure	4	445	5	378	356	378	429
MapOutputServletFailure	83	633	292	412	368	369	444

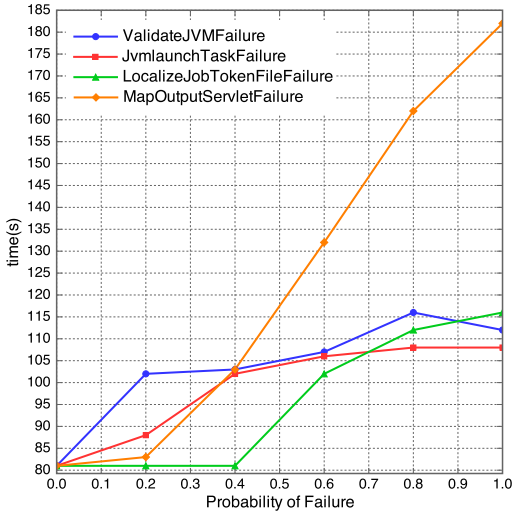


図 6　障害発生確率の変化による実行時間の比較
Fig. 6　Comparison of the execution time due to changes in the fault probability.

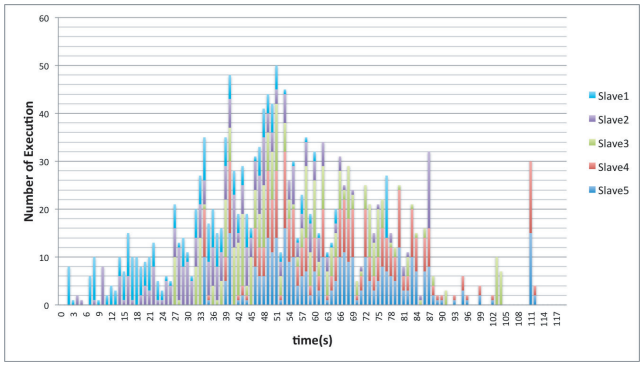


図 7　障害なしでの時間ごとのメソッド実行回数
Fig. 7　Method execution times of each time with no failure.

1 GB 入力の TeraSort の実行時間を記録した．その結果の各実行時間の関係をグラフにしたものを**図 6** に示す．また，**表 7** に，障害発生確率を 1.0 とした場合に生成されたトレース内での各障害の発生回数と，障害を発生させない場合および各障害を確率 1.0 で発生させた場合での，マスタノードと 5 つのスレーブノード内の JobTracker および TaskTracker のメソッド実行回数を示す．なお，マスタノードと各スレーブノード間のネットワーク距離はすべてほぼ等しく，表 7 において，障害を発生させたノードは Slave1 である．

表 7 より，どの障害の場合においても，障害を発生させない場合と比較して，マスタノードや障害発生対象でないノードでのメソッド実行回数が多くなっていることが分かる．これは，障害が発生することによって，他のノードへの負荷が増えたことが原因として考えられる．

ValidateJVMFailure や JvmlaunchTaskFailure は，確率を変化させても実行時間にはあまり変化が見られないことが図 6 で確認できる．TaskTracker での JVM へのタスクの進捗確認や，タスクの起動といった動作において，障害が発生した回数はどちらも 10 回程度であり，これぐらいであればジョブの実行時間にはそこまで影響しないため，このような結果になったと考えられる．またこの結果から，これらの障害が発生した後のタスクの再スケジューリングもスムーズに行われていることが分かる．

また，LocalizeJobTokenFileFailure では，障害発生確率が 0.4 まで，障害がジョブの実行に影響をほとんど与えていないことが分かる．この要因として，ジョブファイルのローカライズは，各ノードの TaskTracker においてジョブのはじめの段階に 1 度だけ行われる動作であることがあげられる．そのため，失敗してもすぐに再実行を行い，それが成功すればその後のタスク実行には影響されていない．しかし，確率が上がると，ローカライズに失敗した後の再実行も失敗する確率が高くなるため，ジョブの実行により時間がかかる結果となっている．

MapOutputServletFailure の場合は，確率を上げるにつれて，実行時間が大きく変化している．特に確率 1.0 の場合は障害なしの場合の実行に約 2.25 倍の時間がかかっており，他の障害と比較しても，特に高い発生確率において実行に 2 倍以上の時間がかかっていることが分かる．また表 7 より，この障害が確率 1.0 で発生した場合は，マスタノードでのメソッド実行回数が，障害なしの場合に比べて約 1.49 倍となっており，JobTracker でのメソッド実行回数も大きく増加してしまっていることが確認できる．**図 7** と**図 8** に，障害を発生させない場合と MapOutputServletFailure を確率 1.0 で発生させた場合の，各時間ごとの TaskTracker のメソッド実行回数を示す．図 7，図 8 より，Map 出力分配の際に障害が発生した場合，障害なしの場合と比較してメソッドの実行回数は多くないが，前半部分に実行が集中していることが分かる．これは，Map の出力を他の

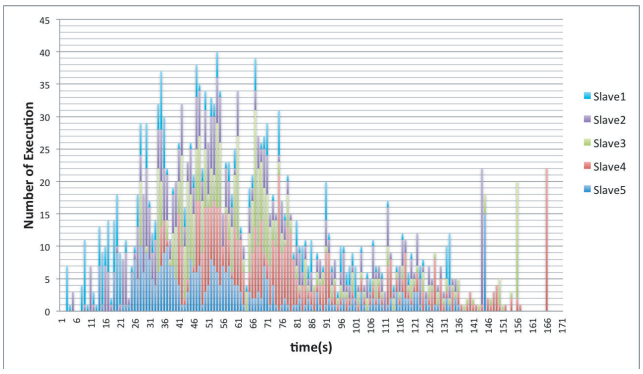


図 8　MapOutputServletFailure 下での時間ごとのメソッド実行回数

Fig. 8　Method execution times of each time under MapOutputServletFailure.

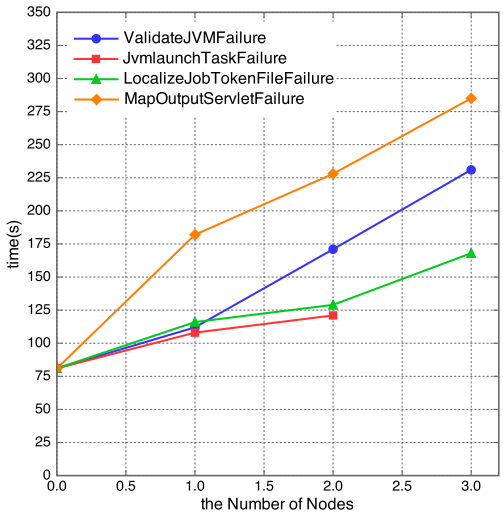


図 9　障害を発生させるノード数の変化による実行時間の比較

Fig. 9　Comparison of the execution time due to the change in the number of nodes that are faulted.

ノードに配布するという，MapReduce の Map フェーズから Reduce フェーズに移る際のシャッフルフェーズでの動作が，障害により多数回繰り返されることによって，高頻度で通信が発生していることが原因としてあげられる．それによって，Map フェーズから Reduce フェーズにスムーズに移行することができず，実行時間が長引いてしまう結果になったと考えられる．

このように，ジョブファイルのローカライズといった少ない回数しか行われない動作と比較し，Map 出力分配のような多数回行われる動作での障害の方が，ジョブの実行時間により影響を与えることは明らかである．

4.3.3　障害を発生させるノード数の変化による比較

分散システムでの障害は，複数のノードで発生する可能性があることも考える必要がある．今回は，障害発生確率を 1.0 に固定し，障害を発生させるノード数を増やすことにより，MapReduce の実行にどのような影響が出るのか調べる．この実験には，1 GB 入力の TeraSort を用いた．その結果のグラフを**図 9** に示す．

JvmlaunchTaskFailure は 3 ノードで，その他の障害では 4 ノードでジョブは失敗に終わっており，図 9 のグラフには，ジョブが成功した場合のみの結果を記している．どの障害においても，発生させるノード数を増やすことによって実行時間が大きく変化しており，障害が複数ノードで発生することで，MapReduce の動作に深刻な影響を与えてしまうことが分かる．また，今回も特に MapOutputServletFailure が発生した場合が実行に時間がかかっていることが確認できる．これらの結果により，MapReduce がより効率的で耐故障性に優れたシステムになるためには，何度も繰り返し実行する必要のある処理におけるタスクスケジューリングや耐故障動作を改善していく必要があるといえる．

5.　まとめと今後の展望

本研究では，分散システムを実現している Hadoop の最も重要な処理の 1 つである MapReduce を対象として，アスペクト指向言語である AspectJ を用いて障害を発生させる手法により，耐故障処理に関する性能評価を行った．Hadoop の障害検知や回復動作は複雑なものとなっており，ソースコードを解析するだけでは検証が困難である．しかし，実際にシステムに障害を発生させ，システムの動作を監視するモニタを実装することにより，効率的に MapReduce の障害発生後の動作を解析することができる．この手法を用いることで，元の Hadoop のソースコードに手を加えず，また実際のシステムの動作に負荷をほとんど与えることなく性能評価を行うことが可能である．

今回対象とした JobTracker と TaskTracker は，MapReduce の中でも重要な位置にあるデーモンであるが，それだけにそれらの動作には高い耐故障性が求められる．JobTracker は，最近の研究で高可用性を実現しているが，TaskTracker においては，データの破損やメッセージ通信障害などの様々な規模の障害が発生する可能性がある．本研究における評価実験により，シャッフルフェーズにおける処理に見られるような，通信が多数回行われる処理において障害が発生すると，ジョブの実行に大きく影響を与えてしまうことが確認できた．このとき，マスタノード内の JobTracker でメソッド実行回数が増加しているため，効率的にジョブを実行するうえでのボトルネックとなってしまっていることが分かる．またこの事実は，既存研究 [3] においても確認されている．そのため，MapReduce の耐故障処理での安定性や効率を高めていくためには，TaskTracker での通信頻度が高い処理をさらに改善していく必要があると考えられる．

また，障害を発生させる場所を決定する際に，対象となる処理をソースコードの中から探す必要があるが，障害発生後に行われる処理まで見る必要はなく，任意の場所を選択するだけで障害を発生させることができる．今回

は MapReduce 動作における障害をいくつか実装したが，Hadoop において MapReduce と同等に重要なシステムである HDFS にも障害を発生させることで，様々な側面から Hadoop の耐故障性を評価することが可能であり，本手法は拡張性を有しているといえる．

今後の展望として，本手法を拡張し，障害発生タイミングを操作したり，例外発生に至らないような障害についても対応したりすることによって，さらに幅広い評価を行う方法が考えられる．また，モニタから生成されたトレースを機械学習によってリアルタイムに解析することによって，障害検知システムや動的なパラメータチューニングを行う手法の開発なども考えられる．

参考文献

[1] Tanenbaum, A.S. and van Steen, M.（著），水野忠則，佐藤文明，鈴木健二，竹中友哉，西山　智，峰野博史，宮西洋太郎（訳）：分散システム 原理とパラダイム，第 2 版，ピアソン桐原 (2009).

[2] Dean, J. and Ghemawat, S.: MapReduce: Simplified data processing on large clusters, *Commun. ACM*, Vol.51, No.1, pp.107–113 (2008).

[3] 森　達哉，木村達明，池田泰弘，上山憲昭：MapReduce システムのネットワーク負荷分析，経営の科学，Vol.56, No.6, pp.331–338, 社団法人日本オペレーションズ・リサーチ学会 (2011).

[4] Apache Software Foundation: Apache Hadoop (online), available from ⟨http://hadoop.apache.org/⟩.

[5] Hsueh, M.-C., Tsai, T.K. and Iyer, R.K.: Fault Injection Techniques and Tools, *IEEE Computer*, Vol.30, No.4, pp.75–82 (1997).

[6] AspectJ Project: AspectJ, (online), available from ⟨http://www.eclipse.org/aspectj/⟩.

[7] White, T.（著），玉川竜司，兼田聖士（訳）：*Hadoop 2nd edition*, pp.183–192, オライリージャパン (2011).

[8] Wang, F., Qiu, J., Yang, J., Dong, B., Li, X. and Li, Y.: Hadoop High Availability through Metadata Replication, *CloudDB '09*, pp.37–44, ACM (2009).

[9] Apache Software Foundation: Apache ZooKeeper, (online), available from ⟨http://oss.infoscience.co.jp/hadoop/zookeeper/⟩.

[10] LINBIT: DRBD, (online), available from ⟨http://www.linbit.com/en/products-and-services/drbd⟩.

[11] Linux-HA: Heartbeat, (online), available from ⟨http://linux-ha.org/wiki/Heartbeat⟩.

[12] Trodhandl, C. and Weiss, B.: A Concept for Hybrid Fault Injection in Distributed Systems, *TAIC PART 2008*, Fast Abstract, IEEE Computer Society (2008).

[13] Joshi, P., Gunawi, H.S. and Sen, K.: PreFail: A Programmable Failure- Injection Framework, EECS Department University of California, Berkeley Technical Report No.UCB/EECS-2011-30, pp.171–173 (2011).

[14] 清水裕亮, 櫻井孝平, 山根　智：AspectJ を用いた Hadoop の監視とプロファイリング手法の提案，*ComSys2012*, pp.70–78, 情報処理学会 (2012).

付　　録

A.1　トレースファイル

トレースは，
タイムスタンプ-ホスト名-トレース (ノード) 名={ メソッド名や障害発生 }
で構成されている．ホスト名およびトレース名はそれぞれの障害において，すべて同じものであるため 2 回目からは省略する．

A.1.1　SubmitJobFailure

```
130417133658-sirius.csl.ec.t.kanazawa-u.ac.jp-jobtrackertrace
={ClusterStatus org.apache.hadoop.mapred.JobTracker.
 getClusterStatus(boolean),[false]}
130417133700={AccessControlList org.apache.hadoop.
mapred.JobTracker.getQueueAdmins(String),[default]}
130417133700={SubmitJobFailure: injected fault point at
JobStatus org.apache.hadoop.mapred.JobTracker.
addJob(JobID, JobInProgress)}
130417133701={ClusterStatus org.apache.hadoop.
mapred.JobTracker.getClusterStatus(boolean),[false]}
130417133701={TaskTracker org.apache.hadoop.
mapred.JobTracker.getTaskTracker(String),[...]}
130417133701={TaskTrackerStatus org.apache.hadoop.
mapred.JobTracker.getTaskTrackerStatus(String),[...]}
130417133701={ClusterStatus org.apache.hadoop.
mapred.JobTracker.getClusterStatus(boolean),[false]}
130417133704={TaskTracker org.apache.hadoop.
mapred.JobTracker.getTaskTracker(String),[...]}
130417133704={ClusterStatus org.apache.hadoop.
mapred.JobTracker.getClusterStatus(boolean),[false]}
```

A.1.2　ValidateJVMFailure

```
130417142558-procyon.csl.ec.t.kanazawa-u.ac.jp-tasktrackertrace
 ={String org.apache.hadoop.mapred.TaskTracker.
  getLocalTaskDir(String, String, String), [...]}
130417142558={String org.apache.hadoop.mapred.
TaskTracker.getJobCacheSubdir(String),[hadoop]}
130417142559={JvmTask org.apache.hadoop.mapred.
TaskTracker.getTask(JvmContext), [...]}
130417142603={ValidateJVMFailure: injected fault point at
 void org.apache.hadoop.mapred.TaskTracker.validateJVM
 (TaskTracker.TaskInProgress, JvmContext, TaskAttemptID)}
130417142603={ValidateJVMFailure: injected fault point at
 void org.apache.hadoop.mapred.TaskTracker.validateJVM
 (TaskTracker.TaskInProgress, JvmContext, TaskAttemptID)}
130417142603={ValidateJVMFailure: injected fault point at
 void org.apache.hadoop.mapred.TaskTracker.validateJVM
 (TaskTracker.TaskInProgress, JvmContext, TaskAttemptID)}
```

A.1.3　JvmlaunchTaskFailure

```
130417135128-procyon.csl.ec.t.kanazawa-u.ac.jp-tasktrackertrace
 ={String org.apache.hadoop.mapred.TaskTracker.
  getPrivateDistributedCacheDir(String),[hadoop]}
130417135128={void org.apache.hadoop.mapred.TaskTracker.
TaskInProgress.launchTask(TaskTracker.RunningJob),
[org.apache.hadoop.mapred.TaskTracker$RunningJob@2dbb97d4]}
130417135128={String org.apache.hadoop.mapred.TaskTracker.
 getLocalJobDir(String, String),[...]}
130417135128={JvmlaunchTaskFailure: injected fault point at
 void org.apache.hadoop.mapred.JvmManager.JvmManagerForType.
  JvmRunner.runChild(JvmManager.JvmEnv)}
130417135128={JvmlaunchTaskFailure: injected fault point at
 void org.apache.hadoop.mapred.JvmManager.JvmManagerForType.
  JvmRunner.runChild(JvmManager.JvmEnv)}
130417135128={boolean org.apache.hadoop.mapred.
TaskTracker.TaskInProgress.wasKilled(),[]}
130417135128={JvmlaunchTaskFailure: injected fault point at
 void org.apache.hadoop.mapred.JvmManager.JvmManagerForType.
```

```
  JvmRunner.runChild(JvmManager.JvmEnv)}
130417135128={JvmlaunchTaskFailure: injected fault point at
 void org.apache.hadoop.mapred.JvmManager.JvmManagerForType.
  JvmRunner.runChild(JvmManager.JvmEnv)}
130417135134={JvmlaunchTaskFailure: injected fault point at
 void org.apache.hadoop.mapred.JvmManager.JvmManagerForType.
  JvmRunner.runChild(JvmManager.JvmEnv)}
130417135137={void org.apache.hadoop.mapred.TaskTracker.
TaskInProgress.launchTask(TaskTracker.RunningJob),
[org.apache.hadoop.mapred.TaskTracker$RunningJob@2dbb97d4]}
130417135137={JvmlaunchTaskFailure: injected fault point at
 void org.apache.hadoop.mapred.JvmManager.JvmManagerForType.
  JvmRunner.runChild(JvmManager.JvmEnv)}
130417135155={boolean org.apache.hadoop.mapred.TaskTracker.
FetchStatus.fetchMapCompletionEvents(long),[1366174315425]}
130417135225={void org.apache.hadoop.mapred.TaskTracker.
TaskInProgress.jobHasFinished(boolean),[false]}
130417135225={void org.apache.hadoop.mapred.TaskTracker.
  TaskInProgress.kill(boolean),[false]}
```

A.1.4 LocalizeJobTokenFileFailure

```
130417140501-procyon.csl.ec.t.kanazawa-u.ac.jp-tasktrackertrace
 ={LocalizeJobTokenFileFailure: injected fault point at
  FileStatus org.apache.hadoop.fs.FileSystem.getFileStatus(Path)}
130417140501={LocalizeJobTokenFileFailure: injected fault point at
  FileStatus org.apache.hadoop.fs.FileSystem.getFileStatus(Path)}
130417140501={LocalizeJobTokenFileFailure: injected fault point at
  FileStatus org.apache.hadoop.fs.FileSystem.getFileStatus(Path)}
130417140501={LocalizeJobTokenFileFailure: injected fault point at
  FileStatus org.apache.hadoop.fs.FileSystem.getFileStatus(Path)}
130417140501={void org.apache.hadoop.mapred.TaskTracker.
  TaskLauncher.addFreeSlots(int),[1]}
130417140501={void org.apache.hadoop.mapred.TaskTracker.
TaskInProgress.reportDiagnosticInfo(String),[Error initializing
   attempt_201304171404_0001_m_000016_0:
java.io.FileNotFoundException: ..... ]}
130417140501={void org.apache.hadoop.mapred.TaskTracker.
  TaskLauncher.addFreeSlots(int),[1]}
```

A.1.5 MapOutputServletFailure

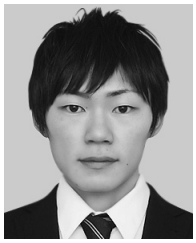
```
130417140956-procyon.csl.ec.t.kanazawa-u.ac.jp-tasktrackertrace
 ={MapOutputServletFailure: injected fault point at FileInputStream
 org.apache.hadoop.io.SecureIOUtils.openForRead(File, String)}
130417140956={long org.apache.hadoop.mapred.TaskTracker.
  getProtocolVersion(String, long),
  [org.apache.hadoop.mapred.TaskUmbilicalProtocol,19]}
130417140957={void org.apache.hadoop.mapred.TaskTracker.
  MapOutputServlet.doGet(HttpServletRequest,
  HttpServletResponse),[org.apache.hadoop.http.HttpServer$
  QuotingInputFilter$RequestQuoter@5577f5b5,HTTP/1.1 200
  Content-Type: text/html; charset=utf-8]}
130417140957={MapOutputServletFailure: injected fault point at
 FileInputStream org.apache.hadoop.io.SecureIOUtils.
 openForRead(File, String)}
130417140957={String org.apache.hadoop.mapred.TaskTracker.
  getLocalTaskDir(String, String, String, boolean),[...]}
130417140957={MapOutputServletFailure: injected fault point at
 FileInputStream org.apache.hadoop.io.SecureIOUtils.
 openForRead(File, String)}
130417140957={String org.apache.hadoop.mapred.TaskTracker.
  getJobCacheSubdir(String),[hadoop]}
130417140957={MapOutputServletFailure: injected fault point at
 FileInputStream org.apache.hadoop.io.SecureIOUtils.
 openForRead(File, String)}
130417140957={Object org.apache.hadoop.mapred.TaskTracker.
LRUCache.get(Object),[...]}
130417140957={MapOutputServletFailure: injected fault point at
FileInputStream org.apache.hadoop.io.SecureIOUtils.
openForRead(File, String)}
130417140959={TaskCompletionEvent[] org.apache.hadoop.mapred.
TaskTracker.FetchStatus.getMapEvents(int, int),[37,10000]}
```


中川 洋介

1990 年生．2013 年金沢大学理工学域電子情報学類卒業．同年同大学大学院自然科学研究科修士課程入学．分散並列処理基盤の耐故障性に関する研究に従事．

櫻井 孝平　（正会員）

1981 年生．2006 年芝浦工業大学大学院工学研究科修士課程修了．2009 年東京大学大学院総合文化研究科博士課程修了．同年より芝浦工業大学大学院工学研究科ポスドク研究員．2011 年より金沢大学理工学域電子情報学系助教．学術博士．アスペクト指向プログラミング言語やソフトウェアテスト・デバッグの研究に従事．ソフトウェア科学会，ACM 各会員．



清水 裕亮

1990 年生．2012 年金沢大学理工学域電子情報学類卒業．同年同大学大学院自然科学研究科修士課程入学．分散並列処理に関する研究に従事．

山根 智　（正会員）

1984 年京都大学大学院修了．現在，金沢大学理工研究域電子情報学系教授．博士（京都大学）．リアルタイムハイブリッドシステム等の形式的検証の研究に従事．EATCS，日本ソフトウェア科学会等会員．