情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2013-OS-126 No.10
2013/7/31

AspectJを用いたFault-Injectionによる Hadoop MapReduceのフォールトトレランス検証

中川 洋介[1,†1]　櫻井 孝平　清水 裕亮　山根 智

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浴びている．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．分散システムは大規模な環境で使用されることが多く，ネットワーク通信障害を始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) が要求される．しかし分散システムの耐故障処理は複雑なものとなっているため，効率的な検証手法が求められている．本研究では，Apache Hadoopを対象とし，AspectJを用いて，MapReduceの動作において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．この手法により，MapReduceの障害発生後の動作をランタイムに解析することが可能である．今回，障害発生下におけるMapReduceアプリケーションの実行時間や，モニタにより生成されたトレースにより，Hadoop MapReduceの耐故障性についての評価実験を行った．

1. はじめに

近年，クラウドコンピューティング [1] 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．その背景には，スマートフォンや無線LANなどの普及があり，クラウドサービスを利用してデータへのアクセスが容易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**[2] 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**[3] 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 [4] 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始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．そのため，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，モデル検査などを用いて，十分に検証を行った上でシステムを構成する必要がある．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．したがってシステムのフォールトトレランスを検証するのは容易なことではなく，有効な検証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検証手法として，実際に対象となるシステムに障害を発生させ，実行時の振る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[5] 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の様々な処理に障害を発生させ (**Fault-Injection**(以下，FIと略す)[6])，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**[7] を用いた．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

[1] 金沢大学
Kanazawa Uniersity
[†1] 現在，自然科学研究科
Presently with Natural Science & Technology

ⓒ 2013 Information Processing Society of Japan

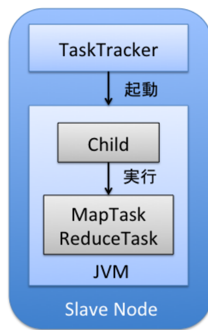

図 1 TaskTracker の動作

ションの実行時間の比較などの実験を行った．この実験により，Hadoop MapReduce の耐故障性について評価を行うことを目的とする．

以下，第 2 章に本手法を実装する上で基となる Hadoop MapReduce とその関連研究について述べ，MapReduce 動作中に発生しうる障害について述べる．第 3 章において，手法の提案とその実装方法について述べ，第 4 章にて手法の評価実験を行い，第 5 章で本研究をまとめる．

2. 基盤技術と関連研究

本章では，2.1 で Hadoop の MapReduce 動作とその障害について述べ，2.2 で関連研究について述べる．

2.1 Hadoop MapReduce

2.1.1 MapReduce の動作

MapReduce は，大規模なデータを複数のノード (コンピュータ) で構成されるクラスタ内で並列分散処理させるフレームワークである．クラスタは，マスターノードとスレーブノード群で構成されており，Hadoop の MapReduce ジョブの動作において根幹を担っているのが，マスターノード内で MapReduce 動作を管理する **JobTracker** とスレーブノード群内において実際にタスクを実行する **TaskTracker** である．JobTracker は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，ジョブを依頼する．そして，タスクを受け取った TaskTracker は，タスクを実行するインスタンスを生成し，子 JVM 内においてタスクを実行する (図 1)．大規模な分散環境で処理を行うために，これらの MapReduce 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両立したものである必要がある．

MapReduce の動作中には，ユーザーの MapReduce アプリケーション中での例外処理や，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，それらの障害を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．

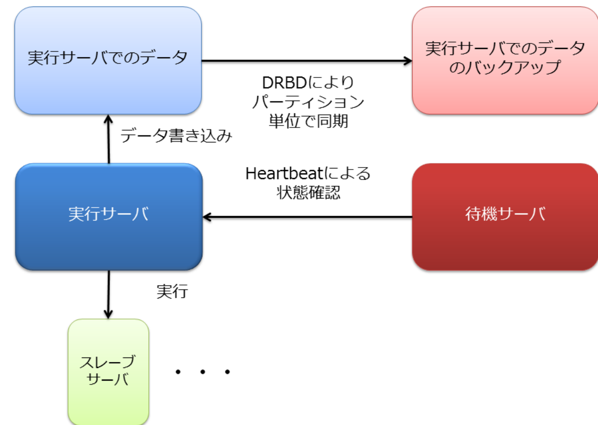

図 2 HA によるマスターノードの二重化

2.1.2 JobTracker における障害

JobTracker は MapReduce ジョブを管理しているマスターノードであるため，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)**[9] によるマスターノードの二重化や，Hadoop のプラグインである **Apache Zookeeper**[10] 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)[11] と Heartbeat[12][*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ことで，高可用性を実現している (図 2)．しかし，この手法は，単一障害点を回避するためだけに新たなサーバを用意する必要があり，障害が発生した際の新たなマスターノードの起動に時間がかかるなどの問題がある．

2.1.3 TaskTracker における障害

TaskTracker は，実際に MapReduce タスクを実行し，メッセージ通信を高頻度で行なっているため，JobTracker と比べて多くの障害が発生する可能性がある．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たりすることによって，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みなす．しかし，ある程度の規模のあるジョブの場合は，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

[*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 間のものとは別

ⓒ 2013 Information Processing Society of Japan

ができる．TaskTracker に障害が高頻度で発生した場合，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 に投げることで，タスクを再開させる．

注目すべき点として，クラスタに属する各 TaskTracker 同士は，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挙げられる．特に大規模な MapReduce アプリケーションの場合，各 TaskTracker 同士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 はマスターである JobTracker を通して，情報交換を行なっている．

2.2 関連研究

以上により，MapReduce の動作中には様々なレベルにおける障害の発生が考えられ，その後障害検知などに関する動作を行なっていることが分かる．しかし，分散システム自体の構造が複雑であるため，そのようなフォールトトレランスに関する動作には，改善すべき部分も存在しているのが事実である．Hadoop のこのような動作を効率的に検証するためには，実際に様々な障害を発生させることが有効である．障害を発生させた上で検証を行なっていくことで，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に繋がると考えられる．本研究では，MapReduce の動作に重点を置き解析を行うことで，分散システムの安全性の改善及び，動作の効率化のための検証を行うことを目的とする．

分散システムに障害を発生させること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 [13] や，障害発生に関するポリシー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 [14] などがある．しかし，これらは MapReduce に特化したものではなく，分散システム全体に対するものであるため，MapReduce のさらに細かい部分にまで解析を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や例外を発生させる Fault-Injection 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いるものであり，0.0〜1.0 の数値を指定した設定ファイルを読み込むことで，障害発生確率を制御することができる．しかし，ユーザーが作成した MapReduce アプリケーションを障害下で実行し，検証を行うことを目的としており，Hadoop 自体の動作解析を行うためのものではない．

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を行うことも可能である [15]．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 は，Java のロギングユーティリティである log4j によりログが取得できるが，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したが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 箇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．

3. 手法の提案と実装

本章では，3.1 にて Hadoop における MapReduce 動作の有効な検証法として，AspectJ により FI 及びシステムを監視するモニタを用いる手法を提案し，3.2 にてその実装について述べる．

3.1 手法の提案

2 章で述べたように，Hadoop のフォールトトレランスを検証する上で，実際に障害を発生させる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できるだけ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，MapReduce の動作を解析する手法を提案する．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 [6]．AspectJ を用いることで，実際のコードに手を加えることなく様々な障害を任意の場所で発生させることが可能であり，同時にモニタから生成されるメソッド実行トレースにより，MadReduce の動作を解析することができる．

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可能となる．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応させる時点 (**ポイントカット**) を決定する．そして，ポイントカットで実行するロギングやデバッグなどの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させることができる．

今回，ポイントカットとして，MapReduce を動作させる上で中核を担っている JobTracker と TaskTracker を中心に選択し，アドバイス内でコードに例外を投げることで障害を発生させる．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，障害が発生させた場合にもトレースが生成される．トレースを用いることにより，障害が発生した回数や時間などの情報を得ることができる．

ⓒ 2013 Information Processing Society of Japan

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

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

表 1 使用したソフトウェアのバージョン

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

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 MapReduce 処理の中から，発生させる障害及び発生させる場所，タイミングを決定し，ポイントカット及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及びアドバイスで投げる例外を示す．例えば **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繋がると考えられる．しかし 2.1.2 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．

3.2.2 TaskTracker での FI

図3に，TaskTracker における障害実装の概要図を示す．

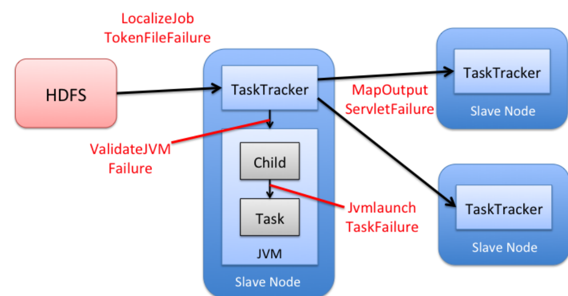

図 3 TaskTracker における障害実装

- **ValidateJVMFailure**
  TaskTracker での障害では，まず始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 を起動する時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**

ⓒ 2013 Information Processing Society of Japan

表 3　使用した計算機のスペック

CPU	Intel® Core^{TM} i5-3470 Processor
動作周波数	3.20GHz
コア数	4
RAM 容量	8GB
ディスク	1TB SATA HDD(7200 回転)

表 4　core-site.xml

設定項目	設定値
io.file.buffer.size	65536
io.sort.factor	20
io.sort.mb	600
fs.inmemory.size.mb	200

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上での，実装した手法の性能評価及び MapReduce のフォールトトレランスに関する実験を行う．まず 4.1 にて，実験を行う際に用いた環境について述べる．そして 4.2 にて提案手法の評価について述べ，4.3 にて実験を行う．

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 台をマスターノード，5 台をスレーブノードとした．また，Hadoop を動作させる上での，設定ファイルをそれぞれ表 4，表 5 に示す．

4.2 手法の評価

実装を行った FI 及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 及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 1GB，10GB，100GB の 3 種類

表 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	60000
mapreduce.reduce.shuffle.read.timeout	180000
mapreduce.task.timeout	600000
mapred.tasktracker.expiry.interval	60000

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，1GB，10GB，100GB 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，FI やモニタの実装が Hadoop の MapReduce 処理に与える負荷は非常に小さなものとなっていることが分かり，MapReduce 処理のフォールトトレランスを検証する上で問題はないと考えられる．

4.3 実験

実装を行った FI とモニタを用いて，MapReduce のフォールトトレランスに関する実験を行う．4.3.1 では，障害発生確率を変化させることによる実行時間などの比較を行い，4.3.2 では，障害を発生させるノード数を変化させることにより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 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，1GB 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 4 に示す．また，表 7 に，各障害の障害発生確率を 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及び各障害を発生させた場合でのマスターノードと 5 つのスレーブノードで生成された JobTracker 及び TaskTracker のトレースファイルサイズを示す．なお，マスターノードと各スレーブノード間のネットワーク距離は全てほぼ等しく，表 7 において，障害を発生させたノードは Slave1 である．

図 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，マスターノードや障害発生対象でないノードでのトレースのファイルサイズが大き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ために，各メソッドの実行回

表 6 TeraSort にかかる実行時間とスループット

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)
アスペクト無し	1	75	13.33
	10	312	32.05
	100	3541	28.24
アスペクト有り	1	82	12.34
	10	320	31.25
	100	3758	26.61

表 7 各障害の発生回数とトレースファイルサイズ

障害名	発生回数	トレースファイルサイズ (byte)					
		Master	Slave1	Slave2	Slave3	Slave4	Slave5
障害なし		67860	53566	52979	77916	76891	53322
ValidateJVMFailure	12	83343	11734	96260	107894	105669	72143
JvmlaunchTaskFailure	10	92016	5611	64962	87977	91269	69722
LocalizeJobTokenFileFailure	4	74338	1548	83932	97210	97209	77982
MapOutputServletFailure	83	112197	58901	80466	103578	95625	71159

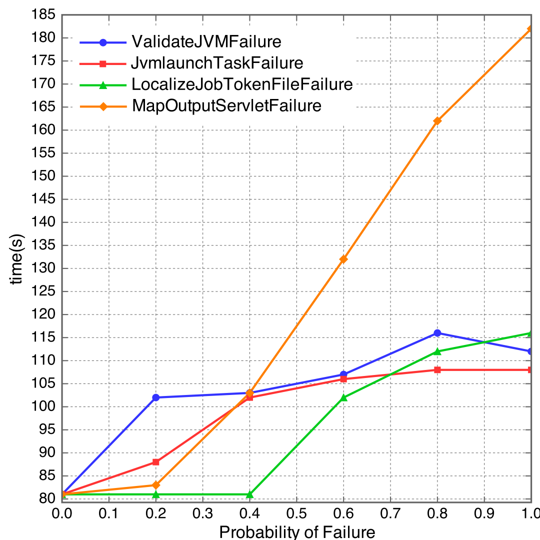

図 4 障害発生確率の変化による実行時間の比較

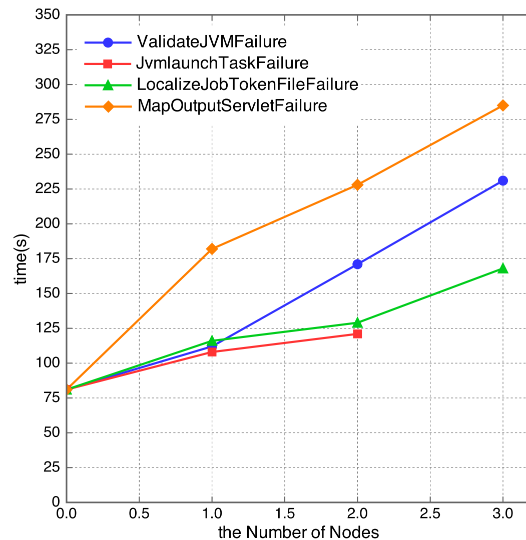

図 5 障害を発生されるノード数の変化による実行時間の比較

数が増加し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 4 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依存しないため，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一度だけ行われる動作であることが挙げられる．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．他の障害と比べても，特に高確率で障害を発生させた場合は，実行に倍以上の時間がかかっていることが分かる．この原因として，障害発生回数や対象となるノードのトレースファイルサイズの大きさからも分かるように，Map の出力を他のノードに配布するという，MapReduce のシャッフルフェーズでの動作において，高頻度で通信が発生していることが挙げられる．

このように，ジョブファイルのローカライズといった少

ⓒ 2013 Information Processing Society of Japan

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．

4.3.2 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，1GB 入力の TeraSort を用いた．その結果のグラフを図 5 に示す．

JvmlaunchTaskFailure は 3 ノードで，その他の障害では 2 ノードでジョブは失敗に終わっており，図 5 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言える．

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を提案した．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ー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検証を行うことが可能である．

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，さまざまな側面から Hadoop の耐故障性を検証していくことが可能であり，本手法は拡張性を有していると言える．

今回検証の対象とした 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が実現しているが，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．また本研究により，シャッフルフェーズにおける処理に見られるような，多数回行われる処理において障害が発生すると，ジョブの実行にかなり影響を与えてしまうことが確認できた．そのため，MapReduce 動作の安定性や効率を高めていくためには，TaskTracker の動作をさらに改善していく必要があると考えられる．

今後の展望として，本手法を拡張し，MapReduce の動作に障害を発生させたり障害の発生を止めたりするタイミングを自由に制御することにより，さらに幅広い検証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解析することによる障害検知システムの開発なども考えられる．

参考文献

[1] Michael Armbrust, Armando Fox, Rean Grifth, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia: *Above the Clouds: A Berkeley View of Cloud Computing*, EECS Department University of California, Berkeley Technical Report No. UCB/EECS-2009-28 (2009).

[2] Andrew S. Tanenbaum and Maarten van Steen: 分散システム 原理とパラダイム 第 *2* 版, 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, ピアソン桐原 (2009).

[3] Jeffrey Dean and Sanjay Ghemawat: *MapReduce: simplified data processing on large clusters*, Comm.ACM (2008).

[4] 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, オペレーションズ・リサーチ：経営の科学 56(6), pp.331-338 (2011).

[5] Apache Software Foundation. *Apache Hadoop* (online), `http://hadoop.apache.org/`

[6] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer: *Fault Injection Techniques and Tools*, Computer, Volume: 30, Issue: 4 (1997).

[7] AspectJ Project. *AspectJ* (online), `http://www.eclipse.org/aspectj/`

[8] Tom White: *Hadoop* 第 *2* 版, 玉川 竜司, 兼田 聖士訳, オライリージャパン, pp.183-192 (2011).

[9] Feng Wang, Jie Qiu, Jie Yang, Bo Dong, Xinhui Li, and Ying Li: *Hadoop High Availability through Metadata Replication*, CloudDB’09 (2009)

[10] Apache Software Foundation. *Apache ZooKeeper* (online), `http://oss.infoscience.co.jp/hadoop/zookeeper/`

[11] LINBIT. *DRBD* (online), `http://www.linbit.com/en/products-and-services/drbd`

[12] Linux-HA. *Heartbeat* (online), `http://linux-ha.org/wiki/Heartbeat`

[13] Christian Trodhandl and Bettina Weiss: *A Concept for Hybrid Fault Injection in Distributed Systems*, TAIC PART 2008 (2008).

[14] Pallavi Joshi, Haryadi S. Gunawi and Koushik Sen: *PreFail: A Programmable Failure- Injection Framework*, EECS Department University of California, Berkeley

ⓒ 2013 Information Processing Society of Japan

Technical Report No. UCB/EECS-2011-30, pp.171-173 (2011).

[15] 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 (2012).

ⓒ 2013 Information Processing Society of Japan