クラウド基盤ソフトウェアにおける Failure-Oblivious Computing 導入の検討

杉木章義[†] 奥畑聡仁[††] 加藤和彦[†]

近年，クラウドコンピューティングが注目されている．クラウドは多くの場合，大規模なデータセンターで運用されており，さまざまな障害にもかかわらず，全体のシステムを適切に運用する必要がある．本論文では，Failure-Oblivious Computing の概念をクラウド基盤ソフトウェアに導入し，可用性を向上させることを提案する．本方式では，全体の処理の継続を優先し，一部の障害を見逃すことでシステム全体の可用性の向上を目指す．本研究室で開発しているクラウド基盤ソフトウェア Kumoi に Failure-Oblivious Computing の導入を行い，システムの継続性や副作用による影響などの観点から評価を行った．

Enhancing Cloud Availability Through Failure-Oblivious Computing

Akiyoshi Sugiki ,[†] Akihito Okuhata [††] and Kazuhiko Kato[†]

In recent years, cloud computing has drawn much attention from both industry and academia. A cloud is typically hosted in a large-scale data center that must be well operated despite various failures that may cause severe damage. In this paper, we present a technique to increase availability by introducing a failure-oblivious concept into a software cloud platform. As in the original concept, we made the cloud platform continue its operation even when the presence of failures. To confirm the effectiveness of this approach, we have implemented that concept into our cloud platform, Kumoi, and evaluated it in terms of continued execution and influence from side-effects.

1. はじめに

近年，クラウドコンピューティングが注目されている．クラウドは，計算資源を自身で所有する必要がなく，需要の変動にも対応しやすいなどの利点から広く採用が進んでいる．その利用者からクラウドの実体は見えないが，実際には，クラウドは非常に大規模なデータセンターで運用されている[1)]．近年のデータセンターは数千台から数万台の計算機を収容し，さまざまなハードウェアやソフトウェアの集合で構成されている．

クラウドによるデータセンターへの計算資源の集約が進む一方で，より適切にデータセンターを運用することが求められている．大規模なデータセンターでは，その一部で絶えず障害（failure）が発生しており，これらの障害を利用者から隠蔽し，高い可用性や信頼性を実現する必要がある．この実現のためには，機器の二重化などハードウェアの対応とともに，ソフトウェアによる対応も必要である．

クラウドでは，その実現のために**クラウド基盤ソフトウェア**という並列分散ミドルウェアが用いられている．クラウド基盤ソフトウェアとは，その名の通り，クラウドの中核となるソフトウェアであり，本研究では Infrastructure-as-a-Service (IaaS) 型のクラウドを対象としたミドルウェアを想定する．現在 IaaS では，Eucalyptus[2)]，OpenNebula[3)]，OpenStack[4)] などのさまざまなミドルウェアが用いられている．

データセンターで発生する障害の多くは，このクラウド基盤ソフトウェアで対応することになる．しかしながら，従来の正規の障害対処法—障害の発生を全て検出し，障害の原因や種類に応じて正しく回復処理を行う—では，必ずしも十分に対応できない可能性がある．データセンターでは前述の通り，一部で障害が絶えず発生しており，その全てを正常にするのは難しい．また，クラウドはさまざまなコンポーネントの組み合わせで実現されているため，これら全てが正しく障害

† 筑波大学 システム情報系 情報工学域
Faculty of Engineering, Information, and Systems, University of Tsukuba
†† 筑波大学大学院 システム情報工学研究科 コンピュータサイエンス専攻
Dept. of Computer Science, University of Tsukuba

© 2012 Information Processing Society of Japan

回復処理を行うとは限らない．さらに，クラウドでは機能の充実や変化に対する俊敏性が重要となるため，サービスの信頼性や可用性を高めている間に時代遅れとなる可能性もある．

そこで本論文では，Failure-Oblivious Computing[5] の概念をクラウド基盤ソフトウェアへ導入することを提案する．その概念は Rinard らによって提案され，一部の障害による影響を忘却し，全体の処理の継続を優先する手法である．元々の提案では，プログラム実行時におけるメモリエラーを対象にしていたが，本論文ではこの概念をクラウド基盤ソフトウェアに適用する．

最初の試みとして，本研究室で開発しているクラウド基盤ソフトウェア Kumoi[6),7)] に Failure-Oblivious Computing の導入を行った．Kumoi はオブジェクト指向と関数型言語を融合したモデルを採用しているため，Failure-Oblivious Computing 導入による効果や副作用をある程度予測しやすい．実装では，オブジェクトと関数それぞれに対応する実装を行い，全体として Failure-Oblivious Computing 対応とした．

評価では，Virtual Machine (VM) のライフサイクル管理において標準的と思われるスクリプトに対して，実験を行った．その結果，小規模な障害を避けながらスクリプト実行を最後まで継続できること，また副作用が小規模に留まることを確認した．

2. 動　　機

本章では，まず従来の障害対応の限界について述べ，Failure-Oblivious Computing 導入の必要性を示す．

2.1 障害対応の難しさ

近年のデータセンターは非常に大規模に運用されており，その規模は年々拡大している．このような環境において，全ての計算機が正しく機能することを期待するのは難しい．また，さまざまなハードウェア，およびソフトウェアによるコンポーネントが関係するため，これらが正しく障害に対処することを期待するのは難しい．

特に，クラウドでは安価なハードウェアと，信頼性よりも機能性を優先したソフトウェアコンポーネントを複雑に組み合わせて運用しており，障害は起こりやすい状況にあると言える．ここでは，2 つ示す．最初に示す例は，**図 1** の Libvirt の Unknown Failure の例である．一般に VM を操作する場合，Xen や KVM などのハイパーバイザだけではなく，Libvirt などの VM 操作用のライブラリがよく使用される．この場合，Libvirt が上位のコンポーネントに十分な情報を提供しないため，障害が発生したことは分かるものの，上位で障害の種類に応じて対応を切り替えることは難しい．このように実行パス上のさまざまなコンポーネントのいずかが十分な対応を行わなければ，その障害に応じて処理を切り替えることは難しい．

次に示す例は，**図 2** のタイミングに関係した障害の例である．Xen では VM を起動した後，直ちにディスクなどの統計情報を取得することはできない．この場合，VM の起動とバックグラウンドの課金処理のタイミングが重なったため障害が発生しているが，このようなタイミング障害はコンポーネントの関係が複雑になるほど容易に発生しうる．

以上の障害は，従来の障害対応でも時間やコストをかければ対応可能である．しかしながら，脆い基盤の上で，しかも限られた制約の中で実現していくには限界がある．そこで本研究では，Failure-Oblivious Computing の導入による改善を目指す．

2.2 Failure-Oblivious Computing

Failure-Oblivious Computing[5] は Rinard らによって元々，Safe-C コンパイラによってメモリ境界を動的にチェックするコードを挿入し，境界を越えたメモリアクセスや不正なポインタ参照などのメモリ操作における障害の影響を軽減する手法として提案されている．従来の手法であれば，これらの障害を検出すると直ちにプログラムの実行を終了させるが，Rinard らは不正なメモリアクセスによる影響をできる限り無害化し，プログラムの実行を続けることで，サーバソフトウェアにおけるセキュリティや可用性の低下を防ぐ手法として提案されている．

この Failure-Oblivious Computing の動作は，メモリの書き込み操作と読み込み操作の場合で異なる．まず，書き込み操作の場合には，境界を越えた書き込みを検出すると，プログラムに書き込みを続けさせ，その後書き込まれた内容を破棄する．一方で，読み込み操作の場合には，境界を越えた読み込みを行おうとすると，0 でパディングされた領域など，影響の少ない値を模造（manufacture）し，読み込みの結果とする．

3. 提　　案

本論文では，Failure-Oblivious Computing の概念をクラウド基盤ソフトウェアに適用する．まず，実行モデルや障害の仮定について述べ，その後，実装の詳細について説明する．

3.1 実行モデル

本研究室で開発しているクラウド基盤ソフトウェア Kumoi[6),7)] は，ミドルウェアの機能をカーネルとシェルに分割し，カーネル側は必要最小限な機能だけに留

ⓒ 2012 Information Processing Society of Japan

```
java.rmi.UnexpectedException: unexpected exception; nested exception is:
     org.libvirt.LibvirtException: Unknown failure
     at org.libvirt.ErrorHandler.processError(Unknown Source)
     at org.libvirt.Connect.processError(Unknown Source)
     at org.libvirt.Domain.processError(Unknown Source) ...
```

図 1 障害に関する情報が提供されない場合の例

```
Xen: [DEBUG] start centos5
kumoi.impl.ps.ProcessActor@1bcdbf6: caught org.libvirt.LibvirtException: internal error read_bd_stats:
    Failed to read any block statistics
org.libvirt.LibvirtException: internal error read_bd_stats: Failed to read any block statistics
     at org.libvirt.ErrorHandler.processError(Unknown Source)
     at org.libvirt.Connect.processError(Unknown Source) ...
```

図 2 タイミング障害の例

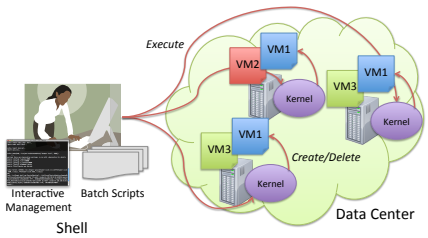

図 3 システムの概要

```
scala> pms.dfilter(_.cpuRatio > 0.9).dmap(_.name)
```

図 4 簡単なスクリプトの例

め，その他の多くのクラウドに必要な機能をスクリプトとして実現していく（**図 3**）．そのため，拡張性やカスタマイズ性に優れていると考えられる．

シェル環境は Scala[8] をベースとしているため，オブジェクト指向と関数型言語を融合したプログラミングモデルでクラウドの構築や操作を行っていく．**図 4** は最も簡単なスクリプトの例である．このスクリプトでは，CPU の使用率が 90 % を超えている物理計算機の名前を抽出している．スクリプト中の `pms` は現在，システムに参加している物理計算機のオブジェクトのリストであり，これらを始めとするさまざまなデータセンター上の資源が分散オブジェクトとして抽象化されている．次の `dfilter()` や `dmap()` は，関数型言語で有名なリスト操作関数である．ただし，これらの関数は自動的に並列分散で計算される．

より複雑なスクリプトの例が，**図 5** の VM の配置圧縮である．この例では，`compacti()` 関数で物理計算機のリスト `pms` を順に走査し，各計算機の VM の

```
1:  def compact(pms: List[HotPhysicalMachine]) {
2:    def firstFit(v: HotVM,
3:                 rest: List[HotPhysicalMachine]) {
4:      rest match {
5:        case h :: rs
6:          if h.cpuAvailable > v.cpuRatio =>
7:            v.migrateTo(h)
8:        case h :: rs => firstFit(v, rs)
9:        case List() =>
10:     }
11:   }
12:   def compacti(pms: List[HotPhysicalMachine]) {
13:     pms match {
14:       case h :: rest =>
15:         h.vms.foreach(v
16:           => firstFit(v, rest.reverse))
17:         compacti(rest)
18:       case List() =>
19:     }
20:   }
21:   compacti(pms.reverse)
22: }
```

図 5 VM の集約スクリプト

リストを `vms` として取得する．`vms` 上のそれぞれの VM に対して，First-Fit 方式で移送先を `firstFit()` 関数で計算する．以上の結果として，VM を少数の物理計算機に集約することができる．

3.2 障害の仮定

本手法は全ての障害に対して適用できないため，いくつかの点について仮定を置く．

まず，管理者が行う作業をシェル上の対話的な操作か自動的に実行されるバッチ処理とする．それぞれの例は図 4 および図 5 に対応している．これらの場合，障害に対応できればよいのは，対話的な一つの操作が完了するまでか，バッチ処理スクリプトの最後に到達するまでである．これらの範囲を超えた障害について

© 2012 Information Processing Society of Japan

は，本手法では対応しない．また，もし対話的な操作の結果が Failure-Oblivious Computing によって多少おかしくなっても，後の対話的な操作によって修正することができる．バッチ処理については，定期的な同一スクリプトの実行によって修正することができる．例えば，図 5 のスクリプトは一度実行されればよい訳ではなく，負荷の変動に応じて何度も実行される必要がある．よって，Failure-Oblivious Computing によって一時的に正常な状態から離脱しても，次回のスクリプト実行によって修正される．この考え方は，自己安定アルゴリズムのものと共通している．

障害については，スクリプト実行中に発生した障害のみを対象とし，一時的な障害（transient failure）か発生直後の恒久的な障害（permanent failure）を仮定する．よって，スクリプトの実行以前に発生していた障害は対象としない．例えば，`pms` は参加している物理計算機を動的に管理しているため，障害が以前から発生していたのであれば，`pms` から自動的に取り除かれる．ここで問題となるのは，`pms` の値を取得してから，スクリプトの実行が完了するまでの物理計算機などの障害である．本提案は，これらの障害の影響をできるだけ避けながら，スクリプトの実行完了を目標とする．

3.3 導入の概要

Kumoi に Failure-Oblivious Computing の概念を導入するには，分散オブジェクトおよび関数のそれぞれに対して対応の実装を行えばよい．分散オブジェクトについては 3.4 節で説明し，関数については 3.5 節で説明する．また，障害の発生を帯域外で通知する方法を 3.6 節で説明する．

3.4 オブジェクトの Failure-Oblivious 化

分散オブジェクトの Failure-Oblivious Computing 対応では，分散オブジェクトのメソッド呼び出しをフックし，例外が発生した場合にその影響をできるだけ無害化することで実装する．Kumoi は分散オブジェクトの実装に Java Remote Method Invocation (RMI) を使用していることから，**図 6** のように RMI 呼び出しを内部でフックし，`try-catch` 節を用いて例外を捕捉することで実装する．

オブジェクトの対応では，オリジナルの提案と同様に，書き込みの場合と読み込みの場合に分割して対応する．クラウド環境においては，前者は VM の起動など，データセンター環境に何らかの変化を及ぼす更新操作に対応し，後者は VM の状態や統計情報の取得など，環境の情報取得操作に対応する．

分散オブジェクトにおいてどちらの操作であるかは，

```
try {
  invoke(proxy, method, args) // RMI invoke
} catch {
   case e: Exception =>
     method.getReturnType match {
       // write op.
       case c if c == classOf[Unit] =>
       // read ops.
       case c if c == classOf[Int] => 0
       case c if c == classOf[Boolean] => false
       case c if c == classOf[String] => ""
       case c if c == classOf[List[_]] => List()
       case c if c == classOf[Option[_]] => None
       ...
     }
}
```

図 6 メソッド呼び出しの疑似コード

メソッドの戻り値を見て判定することができる．戻り値が Scala の `Unit` 型であれば，書き込み操作と判定し，何らかの戻り値があれば読み込み操作と判定する．

書き込み操作の Failure-Oblivious 化は非常に簡単である．例外を捕捉し，何も行わない．以上によって，操作が正常に完了したように見せかけることができる．例えば，`vm.start()` などの VM を起動するメソッドであった場合，メソッドの呼び出しは正常に完了し，実際には VM の起動は行わない．

読み込み操作の場合は，メソッドの戻り値に応じてふさわしい値を模造する必要がある．本論文では次のルールに従って，値の生成を行う．

- **プリミティブ型の場合：** この場合，各型の `0` を返却する．また，`Bool` 型においては，`false` を返す．代案として，最小値または最大値を返す方法，中央の値を返す方法，ある範囲の値をランダムに返す方法，以前にキャッシュしておいた値を返す方法など，さまざまな方法があり得るが，これらは全てアドホックな手法のため，保守的に `0` としておく．
- **一般オブジェクトの場合：** 問題はオブジェクトの場合である．戻り値として `null` を返却してしまうと，後々の操作の際に `NullPointerException` が発生し，スクリプトの実行継続が難しくなる．よって，`String` 型の場合に空の文字列`""`を返却するとともに，その他のオブジェクトの場合もできる限りオブジェクトの模造を試みる．なお，分散オブジェクトについては，次で説明する手法を使用する．
  生成する過程で，`List` や `Set` 型などはうまく（graceful）戻り値を生成できるため，特殊扱いする．この 2 つの場合，空のリストや空の集合

ⓒ 2012 Information Processing Society of Japan

を生成する．また，Scala の Option 型の場合も None を返却すれば，うまく扱える．
これ以外の一般オブジェクトの場合には，コンストラクタをリフレクションで順に取得し，生成をできる限り試みる．コンストラクタの引数は，ルールを再帰的に適用し，生成する．例として，UUID 型のオブジェクトは以上の方法で生成に成功している．また，InetAddress 型などはコンストラクタを持たないため，特別扱いしてクラスメソッドから生成する．

- **分散オブジェクトの場合：** この場合，Java の Proxy と InvocationHandler の機能を利用し，オブジェクトをプロキシで模造する．このプロキシは全てのメソッド呼び出しをフックし，書き込み操作のメソッドの場合は何も行わず，読み込み操作の場合はこれまでのルールを再帰的に適用し，元のオブジェクトらしい振る舞いを行う．

3.5 関数の Failure-Oblivious 化

本研究では，リスト操作関数を中心に関数も Failure-Oblivious 化する．これらの関数は Kumoi のスクリプトにおいて頻繁に使用される．

リスト操作関数の Failure-oblivious 化は，基本的に障害が発生した要素を取り除き，残りの正常な要素のみで計算するように実装している．現在，Failure-oblivious 化の概念の検証を目的としているため，それぞれの関数を手動で実装している．また，後の 3.6 節で述べるように，障害が発生した要素もイベントとして通知するため，計算中の値と障害が発生した要素のリストの 2 つを管理しながら，計算を行う．現在，Failure-oblivious 化されているのは，dmap()，dfiler()，dreduce()，dcount()，dforeach()，dexists()，dforall() の 7 つの関数である．これらは同じような計算処理であるため，内部で fold() 関数に置き換え，手動で実装されている．

関数の Failure-oblivious 化の有効性は，図 4 の例で確認することができる．dfilter() の一部の要素で障害が発生したとすれば，それらを述語（predicate）にマッチしなかったと見なし，結果のリストから取り除けば，障害の影響を避けながらスクリプトの実行を続けることができる．また，dmap() 関数で障害が発生したとすれば，その計算機を要素から取り除いて名前を取得する．この場合，リストの長さが変化し，オリジナルの map() 関数の定義と異なるが，dmap() 関数には filter() 関数が組み合わされており，正常に計算できた要素だけ抽出されていると考えると直感的に分かりやすい．

再度，図 4 の例を確認すると，オブジェクトと関数の実装を組み合わせた場合に次の問題があることが分かる．この場合，pms の一部の物理計算機で障害が発生すると，オブジェクト側で障害がマスクされ，意味を活用してよりうまく対処できる関数側で障害対応が行われない．そこで，オブジェクト側でスタックトレースを参照し，リスト操作関数から呼ばれていれば，例外を通常通り発生させ，関数側で対処させる．

3.6 イベント通知

これまでのオブジェクトと関数の実装によって，ほとんどのスクリプトは実行を継続することができる．しかしながら，Failure-Oblivious Computing 実行が成功すれば，管理者も障害の対応に必要な情報が得られない．そのため，通常のスクリプト処理とは別に，障害の発生をメッセージとして管理者に通知する．

図 7 は，そのメッセージの例である．Kumoi では，Java RMI による分散オブジェクトと Scala Actor によるアクターの 2 つの分散プログラミング・モデルを組み合わせている．障害の発生は予め指定しておいたアクターにメッセージとして通知される．

4. 議　　論

本章では，Failure-Oblivious Computing 導入にともなうさまざまな点について議論を行う．

4.1 導入の効果

本方式は VM や物理計算機などの障害を直接解決しない．障害の発生したコンポーネントは依然として障害の状態のままである．本方式で実現できるのは，全体のスクリプト実行の継続である．従来であれば，正常に処理が完了した計算機，途中まで適用し，障害が発生した計算機，処理が全く適用されていない計算機の 3 種類が発生していたのを，処理が完了した計算機，障害が発生した計算機の 2 種類に近づける．障害については，管理者が後々に対応する必要がある．

4.2 意味論の活用

3.5 節で述べたように，リスト操作関数の方がオブジェクトに比べて意味論（semantics）を活用し，障害をうまく扱える可能性がある．このように，できるだけ高水準な記述を利用し，操作の意味を活用した方がうまく障害に対処できる．ただし，一方でどこまで記述の意味に踏み込むかという問題が発生する．

例えば，図 4 では CPU 負荷の高い計算機を抽出しているため，障害の発生した計算機を結果に含めた方が正しいかもしれない．この判断は dfilter 関数の述語の意味に依存するため，計算機に解釈させるのは難しい．よって，スクリプトの記述者が必要な情報を

ⓒ 2012 Information Processing Society of Japan

```
// オブジェクトに障害が発生した場合
FailureObliviousObject(interface kumoi.shell.vm.HotVM,migrateTo,
            List(interface kumoi.shell.pm.HotPhysicalMachine, class kumoi.shell.aaa.AAA),
            List(ibm1, Anonymous),org.libvirt.LibvirtException: POST operation failed:
            xend_post: error from xen daemon: (xend.err '/usr/lib/xen/bin/xc_save 16 8 0 0 1 failed'))
// リスト操作関数に障害が発生した場合
FailureObliviousList('map,List(<function1>),List((centos6-7,java.util.concurrent.TimeoutException),
            (centos6-6,java.util.concurrent.TimeoutException),
            (centos6-5,java.util.concurrent.TimeoutException)))
```

図 7 Failure-Oblivious イベント

与えるか，現在の実装のようにどちらかに統一する必要がある．

また，オブジェクトの実装は関数に比べてよりアドホックな対処となる．これはメソッドの戻り値で何を返すべきか，メソッドの意味に依存し，判断が難しいからである．しかしながら，下記のようなアノテーションを付加することによって，多少改善できる．

```
@foblivious("100.0") def cpuRatio(): Double
```

この手法は各メソッドに対して行わなければならず，またスクリプトの文脈にも依存する．例えば，スクリプトで負荷に余裕のある計算機を抽出する場合は正しく動作するが，反対に負荷の高い危険な計算機を抽出する場合には 0 の方が望ましい．ただし，これもスクリプトの中で `max` 関数や `min` 関数が使用されていれば，関数側で対処した方がよりうまく対処できる．

4.3 副作用による影響

Failure-Oblivious Computing の導入によって，副作用が潜在的に発生する．これが起こりうるのは，障害オブジェクトの結果をもとに，正常なオブジェクトに操作を行う場合や統計情報を計算する場合などである．しかしながら，その影響は多くの場合，小規模に留まると考えられる．よくある例として，物理計算機間で VM 負荷が完全に均一に分散されないなどがある．これは，確かに正常な場合と異なるが，後の実行によって修正されるため，それほど深刻ではない．また，VM やデータが削除される重大な副作用も考えられるが，障害の発生したオブジェクトの結果によって，これらが生じるようにスクリプトを記述するのは極めてまれである．また，オブジェクトも `0` や `false` など Fail-Safe 側の値をできるだけ返却しているため，上記の実行パスを通ることは考えにくい．

4.4 導入によるオーバーヘッド

Failure-oblivious Computing のオリジナルの提案では，メモリ境界の動的なチェックによるオーバヘッドが問題となった．しかしながら，本研究は通常の `try-catch` による例外処理を使用するために，障害が起こらない場合のオーバヘッドは通常と変わらない．また，障害が発生した場合，リフレクション機能などを多用するため，オーバヘッドは発生するが，VM の起動や移送などの操作に時間がかかるため，ほとんどの場合，隠蔽される．

4.5 セキュリティ

元々の提案では，Failure-Oblivious Computing 導入の利点として，可用性とともにセキュリティの向上が指摘されている．しかしながら，本研究でセキュリティを向上させることは難しい．何故なら，本研究ではクラウド利用者のリクエスト処理とは別に，管理者がデータセンター内部で使用するスクリプトを対象としているからである．悪意を持った利用者がこれらのスクリプトの動作に影響を与えるのは非常に困難である．しかしながら，本方式を利用者のリクエスト処理などのアプリケーション・ロジックに適用すれば，セキュリティは向上すると考えられる．

4.6 傍観者効果

オリジナルの提案では，傍観者効果（bystander effect）が指摘されている．傍観者効果とは，障害が隠蔽されることで，開発者が障害回復処理の実装に真剣に取り組まないという問題である．

本提案でも，傍観者効果は確かに起こりうる．しかしながら，その影響はより小さいと考えられる．まず，管理者が記述するスクリプトは一度しか使用されないものも多く，これらに対して障害回復処理を実装させるのは現実的ではない．また，2 章で述べたように近年のデータセンターは非常に複雑であり，全ての障害を予測し，対策するのは現実的に難しい．そのため，現実的な方策として Failure-Oblivious Computing を正常な障害対応と併用する．

4.7 他のミドルウェアへの適用

Failure-Oblivious Computing の概念は，Eucalyp-

ⓒ 2012 Information Processing Society of Japan

```
def deploy(pms: List[HotPhysicalMachine]) = {
  pms.map { p =>
    val v = p.vmm.createVM
    v.name = "centos6-" + p.name
    v.memory = 256 * 1024 * 1024
    v.add(FileDiskAuto("/work/images/centos6-" +
            p.name + ".img"))
    v.add(NatAuto)
    p.vmm.add(v)
  }
}
```

図 8 VM の起動スクリプト

```
def shutdown(pms: List[HotPhysicalMachine]) {
  pms.dmap(_.vms.map(_.shutdown))
}
```

図 9 VM の終了スクリプト

tus や OpenNebula などの他のクラウド基盤ソフトウェアにも適用可能である．しかしながら，Kumoi は内部構造がオブジェクトと関数で整理されているため，Failure-Oblivious Computing 導入による挙動をより予測しやすいと考えられる．また，導入に必要なコストも小さい．RMI の呼び出し部分や並列分散関数のわずかな実装のみを修正すればよいため，簡単である．実際に，導入に要した行数は全体の 31,240 行のうち，445 行であった．

5. 実　　験

Failure-Oblivious Computing 導入によるスクリプト実行の継続性の向上や副作用による影響を確認するため，VM のライフサイクル管理に関するスクリプトを対象に実験を行った．

5.1 実験環境

実験はクラスタ計算機のうち 7 台で行った．各ノードはデュアル Xeon 3.60 GHz CPU，2 GB メモリ，32GB の SCSI ディスクで構成され，全て 1000BASE-T で接続されている．ソフトウェアは CentOS 5.7（Linux 2.6.18），Xen 3.0.3，Sun JDK 1.6.0，Libvirt 0.8.2，Scala 2.9.1final を使用した．

5.2 実験結果

実験に使用したスクリプトを**図 8** および**図 9** に示す．なお，VM の集約に関しては図 5 のスクリプトを使用した．これらの実行結果を**図 10** に示す．

(a) VM の起動： 意図的に壊れた VM 起動イメージを配置し，スクリプトの実行を行った．その結果，正常なものについては全て起動し，壊れた起動イメージの VM については起動しなかった．この起動しなかった VM については，管理者がメッセージを受け取り，後々対応する．

(b) VM の集約： 意図的に障害を注入し，動作を確認した．この場合，物理計算機および VM に障害が発生する 2 つの場合があり得る．図 5 の 15 行目の直前で障害を注入した VM（`centos6-ibm5`）を正常な物理計算機に集約する場合は，Failure-Oblivious 化により `v.cpuRatio` の値が常に 0.0 となり，どの計算機も移送先として該当する．ただし，実際の VM 移送を行う `migrateTo()` メソッドは何も行わないため，問題とならなかった．一方で，21 行目の直前で障害を注入した物理計算機（`ibm1`）に正常な VM を移動する場合は，その計算機の `h.cpuAvailable` の値は Failure-Oblivious 化により常に 0.0 のため，移送の対象とならなかった．

(c) VM の終了： `shutdown` メソッドは VM の終了を待たないため，元々Failure-Oblivious と言える．そのため，障害のタイミングによりメッセージが送信される場合がある以外は，導入前後で動作は変わらなかった．

6. 関 連 研 究

本章では，Failure-Oblivious Computing のデータセンター環境への適用に関する関連研究について述べる．Google は Sawzall の論文[9] の中で Failure-Oblivious Computing との関連について述べている．また，MapReduce の論文[10] でも，一部の不正データなどのさまざまな障害の対処方法について述べている．これらの話題は，大量データ処理を対象としており，本研究が対象とする IaaS とは異なる．また，上記はデータベース分野の外れ値対応と関連している．

田浦らの GXP[11] は，当初グリッドやクラスタを対象として設計された並列分散シェルである．GXP では，コマンドの実行範囲を直前のコマンドが成功した計算機，現在選択している計算機，GXP に参加している全計算機の集合の 3 つ組の中から選択してコマンドの実行範囲を制御することができる．GXP が提供するこれらのマスクを適切に利用してスクリプトを記述すれば，本システムよりも高い可用性と信頼性を実現することができる．ただし，これはスクリプトの記述者にとって多少の煩雑さをともなう．本システムはスクリプト記述の見通しを優先し，その制約の中で可能な限りの高い可用性を実現する．

7. ま と め

本論文では，Failure-Oblivious Computing の概念をクラウド基盤ソフトウェアに導入し，システム全体の継続性，すなわち可用性を向上させることを提案し

ⓒ 2012 Information Processing Society of Japan

```
scala> deploy(pms) // (a) VM の起動： VM の起動イメージ（centos6-ibm5 用）を破壊
reactions: FailureObliviousList('map,List(<function1>),List((ibm5,java.rmi.UnexpectedException: unexpected exception;
nested exception is:
org.libvirt.LibvirtException: POST operation failed: xend_post: error from xen daemon: (xend.err "Error creating domain: (2,
'Invalid kernel', 'xc_dom_parse_elf_kernel: ELF image has no shstrtab\\n')"))))
res2: List[kumoi.shell.vm.HotVM] = List(centos6-ibm1, centos6-ibm2, centos6-ibm3, centos6-ibm4, centos6-ibm6, centos6-ibm7)

scala> compact(pms) // (b) VM の集約： (1) VM（centos6-ibm5）に障害を注入
reactions: FailureObliviousObject(interface kumoi.shell.vm.HotVM,migrateTo,List(interface kumoi.shell.pm.HotPhysicalMachine,
class kumoi.shell.aaa.AAA),List(ibm1, Anonymous),org.libvirt.LibvirtException: POST operation failed: xend_post: error from xen
daemon: No such domain centos6-ibm5)
scala> pms.map(p => p.name -> p.vms)
res3: List[(String, List[kumoi.shell.vm.HotVM])] = List((ibm1,List(centos6-ibm1, centos6-ibm7, centos6-ibm6, centos6-ibm4,
centos6-ibm3, centos6-ibm2)), (ibm2,List()), (ibm3,List()), (ibm4,List()), (ibm5,List()), (ibm6,List()), (ibm7,List()))

scala> compact(pms) // (b) VM の集約： (2) 物理計算機（ibm1）に障害を注入
reactions: FailureObliviousObject(interface kumoi.shell.pm.HotPhysicalMachine,cpuAvailable,List(class kumoi.shell.aaa.AAA),
List(Anonymous), java.rmi.ConnectIOException: Exception creating connection to: 10.0.0.1; nested exception is:
java.net.NoRouteToHostException: No route to host) ...
scala> pms.map(p => p.name -> p.vms)
res4: List[(String, List[kumoi.shell.vm.HotVM])] = List((ibm2,List(centos6-ibm2, centos6-ibm7, centos6-ibm6, centos6-ibm5,
centos6-ibm4, centos6-ibm3)), (ibm3,List()), (ibm4,List()), (ibm5,List()), (ibm6,List()), (ibm7,List()))

scala> shutdown(pms) // (c) VM の終了
```

図 10 VM のライフサイクル管理スクリプトの実行結果

た．今回導入を行った Kumoi は，オブジェクト指向と関数型言語を融合した Scala のプログラミング・モデルを採用しており，Failure-Oblivious Computing 導入による副作用をある程度予測し，管理下に置きやすいと考えられる．実装では，オブジェクトと関数それぞれに対応する実装を行い，全体として Failure-Oblivious Computing 対応とした．実験では，仮想マシンのライフサイクル管理において標準的と思われるスクリプトを用意し，影響を調査した．その結果，スクリプトの実行が最後まで継続されることを確認した．

今後の予定として，さらなる評価や改良を進めるとともに，本アプローチとは反対にトランザクション機能を導入することを検討する．管理者の操作によっては，可用性よりも操作の不可分性（atomicity）を優先したい場合も考えられることから，この実装を進める．

謝　　辞

本研究の一部は，総務省 SCOPE「ディペンダブルな自律連合型クラウドコンピューティング基盤の研究開発」，科研費（2230006，22700023）の支援を受けている．

参 考 文 献

1) Barroso, L. A. and Hölzle, U.: *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, Morgan & Claypool (2009).
2) D. Nurmi *et al.*: The Eucalyptus Open-Source Cloud-Computing System, *IEEE/ACM CCGrid'09*, pp. 18–21 (2009).
3) OpenNebula Project Leads: OpenNebula: The Open Source Toolkit for Cloud Computing (2008). `http://www.opennebula.org/`.
4) Rackspace Cloud Computing: OpenStack: The Open Source, Open Standards Cloud (2010). `http://openstack.org/`.
5) M. Rinard *et al.*: Enhancing server availability and security through failure-oblivious computing, *USENIX OSDI'04*, pp. 21–21 (2004).
6) A. Sugiki *et al.*: Kumoi: A High-Level Scripting Environment for Collective Virtual Machines, *IEEE ICPADS 2010*, pp. 322–329 (2010).
7) Sugiki, A. and Kato, K.: An Extensible Cloud Platform Inspired by Operating Systems, *IEEE/ACM UCC 2011 (Short paper)*, pp. 306–311 (2011).
8) Odersky, M.: The Scala Programming Language (2003). `http://www.scala-lang.org/`.
9) R. Pike *et al.*: Interpreting the Data: Parallel Analysis with Sawzall, *Scientific Programming Journal, Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure*, Vol. 13, No. 4, pp. 227–298 (2006).
10) Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *USENIX OSDI'04*, pp. 137–150 (2004).
11) Taura, K.: GXP : An Interactive Shell for the Grid Environment, *IEEE IWIA'04*, pp. 59–67 (2004).

© 2012 Information Processing Society of Japan