

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

杉木 章義^{1,a)} 奥畑 聡仁² 加藤 和彦¹

受付日 2012年3月28日, 採録日 2012年6月15日

概要: 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. 我々が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.

キーワード: 並列分散システム, 障害対策, 可用性, 運用継続性, クラウドコンピューティング

Enhancing Cloud Availability Through Failure-Oblivious Computing

AKIYOSHI SUGIKI^{1,a)} AKIHITO OKUHATA² KAZUHIKO KATO¹

Received: March 28, 2012, Accepted: June 15, 2012

Abstract: 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.

Keywords: parallel and distributed systems, failure handling, availability, operational continuity, cloud computing

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, データセンターをより適切に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, その一部でたえず障害 (failures) 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

¹ 筑波大学システム情報系情報工學域
Faculty of Engineering, Information, and Systems, University of Tsukuba, Tsukuba, Ibaraki 305-8573, Japan

² 筑波大学大学院システム情報工學研究科コンピュータサイエンス専攻

Department of Computer Science, University of Tsukuba, Tsukuba, Ibaraki 305-8573, Japan

^{a)} sugiki@cc.tsukuba.ac.jp

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、その名のとおりに、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—障害の発生をすべて検出し、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。データセンタでは前述のとおり、一部で障害がたえず発生しており、そのすべてを正常な状態で運用することは難しい。また、クラウドはさまざまなコンポーネントの組合せで実現されているため、これらすべてが正しく障害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機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、我々が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やクラウド提供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とどまることを確認した。

2. 動機

本章では、クラウド固有の障害対策の難しさについて述べ、Failure-Oblivious Computing 導入の必要性を示す。

2.1 障害対応の難しさ

本研究は、さまざまなクラウドの中で、特に IaaS 型のパブリッククラウドに着目して研究を進める。IaaS は、物理計算機、VM、ストレージ、ネットワークなどのさまざま

な資源が複雑に関係し、障害が問題となりやすいと考えられるからである。

IaaS 型のクラウドにおける障害対応の難しさについては、次のように要約することができる。

- スケーラビリティ：近年のデータセンタ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、すべ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。障害はまれにしか発生しないのではなく、たえず発生しているという視点の転換が必要である。
- 機敏性 (agility)：従来の障害対策手法では、事前に発生しうる障害を想定し、各々の障害に対して対策機構を実現していく。ただし、クラウドではサービスの改変や機能拡張が頻繁であることから、この障害対策もきわめて短期間で実現する必要がある。クラウドでは、従来のシステム以上にこのスピードに対する要求が高まっており、環境の変化に対応できる機敏性が求められている。
- オープンシステム：クラウドでは、ハードウェア、ソフトウェアともに、さまざまなコンポーネントを組み合わせて構築されている。クラウドでは、コスト低減の追求から従来の情報システム以上に汎用品 (commodities) が用いられる傾向があり、品質がさまざまなコンポーネントを組み合わせながら、必要とされる信頼性や可用性を実現していく必要がある。

以上から、クラウドでは従来の情報システム以上に動的で不確かな基盤の上で障害対策手法を実現し、可用性や信頼性と機能性のバランスを保ち続けていく必要がある。

クラウドにおけるスケーラビリティ、機敏性の必要性については広く知られているが、ここでは、特にオープンシステムのコンポーネントが問題となる 2 つの例について、特に取り上げて示す。どちらも広く用いられている VM 操作のライブラリ Libvirt の例である。

図 1 では、Libvirt で Unknown Failure が発生している。Unknown Failure は Libvirt において、比較的頻繁に遭遇する障害である。この場合、Libvirt が十分な情報を提供しないため、障害が発生した事実は分かるものの、上位のコンポーネント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、クラウド基盤ソフトウェアの実行パス上のさまざまなコンポーネントのいずれかが十分な情報を提供しなければ、その障害に応じて処理を切り替える

```
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 Unknown Failure の例

Fig. 1 Occurrence of unknown failure.

ことは難しい．特に，VM やそのハイパーバイザは非常に複雑なソフトウェアであることから，このような障害は頻繁に発生しうる．

図 2 は，タイミングに関係した障害の例である．Libvirt では Xen の VM を起動した後，ただ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．このような制限は Libvirt 側で対処してくれず，ライブラリの利用者側で適切に対処する必要がある．この例では，Kumoi の課金処理がバックグラウンドで動作しており，課金処理の VM の統計情報の取得が VM の起動直後のタイミングと重なったために障害が発生している．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ればなるほど発生する可能性がある．

著者らの 50 台のクラスターでも障害は予想以上に発生し，50 台で VM をいっせいに起動する場合においても，数台程度 VM が起動しないことは頻繁に見られる．実際のクラウドでは，台数は数千台から数万台となるため，さらに問題となる．このような状況では，全体のサービス継続性を優先し，すべての障害に対処するよりも，少々の障害を気にしない工夫が必要だろうと考えられる．そこで，本研究では Failure-Oblivious Computing の導入を行う．

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ただ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，影響の少ない値を生成 (manufacture) し，読

```
Xen: [DEBUG] start centos5
kumoi.impl.ps.ProcessActor@1bcd6f6: 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

Fig. 2 Occurrence of timing failure.

み込みの結果とする．

3. クラウド基盤ソフトウェア Kumoi

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．本章では，適用先となる基盤ソフトウェアの概要や基盤ソフトウェア自身の障害対策モデルについて説明する．

3.1 Kumoi の概要

我々が開発しているクラウド基盤ソフトウェア Kumoi [6], [7] の概要を図 3 に示す．Kumoi では，ミドルウェアの機能をカーネルとシェルに分割し，カーネルはデータセンタ内の各計算機で，シェルは管理者の計算機などさまざまな場所で動作するように設計されている．カーネルは必要最小限な機能だけにとどめ，クラウドに必要なその他の多くの機能をスクリプトとして実現するように設計されている．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．

カーネルでは Java Remote Method Invocation (RMI) と Scala の Actor がマイクロカーネルとして最下層に配置されている．そのうえで，物理計算機や VM などの各資源が分散オブジェクトとして抽象化されている (資源マップオブジェクト)．一方で，dmap() や dfilter() などの並列スケルトンや，メンバシップ機能などの並列分散サービスが提供されている．Kumoi では，前者は Java RMI を使用し，後者は Actor を使用して実装されている．よって，すべての通信がマイクロカーネル中の RMI か Actor を経由するようになっており，この通信をフックすれば，障害対策やセキュリティ対策など，さまざまな拡張が簡単に実装できるようになっている．今回の Failure-Oblivious Computing の導入も，この機能を利用して実装されている．

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記述でクラウドの構築や操作を行っていく．図 4 は対話的なスクリプト記述の例である．このスクリプトでは，CPU の使用率が 90% を超え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資源

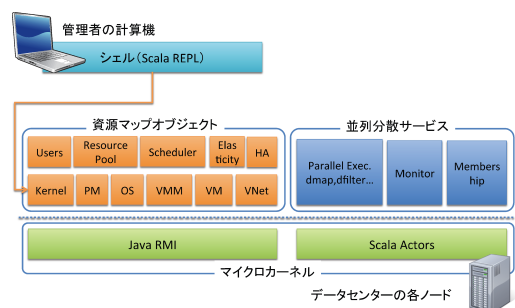


図 3 システムの概要

Fig. 3 System overview.

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

図 4 対話的なスクリプト記述の例

Fig. 4 Example of interactive scripting.

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

図 5 VM 集約スクリプトの例

Fig. 5 Example of VM compaction script.

マップオブジェクトのリストであり、これらをはじめとするさまざまなデータセンタ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 の集約配置である。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()` 関数の中で計算し、`migrateTo()` で実際に移送を行う。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 Kumoi 自身の障害対策

クラウド基盤ソフトウェア Kumoi では、障害対策の多くの部分をスクリプトに委譲しているが、Kumoi 自身も障害を考慮して設計されている。下記で個別の設計方針について説明する。

- 状態の最小化 (stateless) : Kumoi のカーネルでは、状態の一貫性管理や障害対策を考慮して、状態を極力持たないように設計されている。そのため、スクリプ

トから資源オブジェクトに操作を行うと、その内容をただちに実際の資源に反映させる。また、資源オブジェクトの状態をスクリプトから取得しようとする、実際の資源から状態を直接取得し、返却する。この方式はシステムの堅牢性を高めており、万が一、Kumoi のカーネルに障害が発生しても、Kumoi カーネルのみを再起動すれば、動作中の VM などを含めた現在の計算機の状態から、資源マップオブジェクトを正しく復元することができる。

- メンバシップ機構 : Kumoi では、現在参加している物理計算機のリストを Gossip プロトコルを使用し、自動的に管理している。このリストはシェル環境に `pms` として提供され、計算機の参加や離脱の度に自動的に変更される。現在、Kumoi が提供しているその他の VM などの資源はこの物理計算機オブジェクトからたどることができるため、物理計算機のリストを管理することは非常に重要である。

このほかの主要な機能である並列スケルトン部分の障害対策については、Failure-Oblivious 化の実装とも関連するため、後の 4.5 節で説明する。

3.3 従来の障害対策手法の実現

Kumoi では、データセンタ上の資源を資源マップオブジェクトとして抽象化し、スクリプト可能としているため、さまざまな障害対策手法を組み込むことができる。再試行や他の資源での代替などの従来の障害対策手法も、その多くが下記の方法の組合せで実現できる。

- 問合せによる方式 (図 6 (a)) : 各々の資源マップオブジェクトが提供するメソッドを使用して、状態や情報の取得を行い、フィードバックとなる操作を行う。
- 例外捕捉による方式 (図 6 (b)) : 資源マップオブジェクトの操作時に障害が発生した場合には、例外を捕捉し、その内容に応じて障害回復処理を行う。
- イベント通知による方式 (図 6 (c)) : あらかじめ Actor を資源マップオブジェクトに登録しておき、障害発生などのイベント通知を受け取り、回復処理を行う。

4. Failure-Oblivious Computing の導入

Kumoi では、クラウドに必要な多くの機能をスクリプトとして実現するため、スクリプト実行の継続性は非常に重要な問題である。本研究では、データセンタ運用の中核に Failure-Oblivious Computing を導入することを試みる。一方で、クラウド利用者のリクエスト処理を行うアプリケーション・ロジックは、元々、ある程度の Failure-Oblivious Computing が実現されているといえる。たとえば、ウェブ上のサービスはブラウザからの少々の不正な要求にも耐えられるよう設計されている。本研究では、より難易度の高いデータセンタ運用への導入に挑む。

```

1: val cond = (p: HotPhysicalMachine) =>
2:   p.cpuRatio > 0.9 || p.freeMemory < LeastMem
3: pms.filter(cond).map(alertByMail)

```

(a) 問合せによる方式

```

1: try {
2:   val v = local.vmm.createVM
3:   v.name = ...
4:   local.vmm.add(v)
5: } catch {
6:   case pe: PhysicalMachineException => ...
7:   case vmme: VMException => ...
8:   case vme: VMException => ...
9: }

```

(b) 例外捕捉による方式

```

1: val mon = actor { receive { case msg => ... } }
2: vm.watch(mon)

```

(c) イベント通知による方式

図 6 従来の障害対策手法

Fig. 6 Traditional methods for failure handling.

4.1 導入への期待

Failure-Oblivious Computing の導入によって、スクリプトの記述者は下記を期待することができる。

- スクリプト記述の見通し：これまでに、さまざまな言語上の工夫が行われているが、障害対策部分の記述はスクリプト全体の見通しを悪くする。Failure-Oblivious Computing などの手法によって障害が自動的に対処されるのであれば、スクリプト記述者は本質的な部分のみを記述すればよいことになる。
- 従来の障害対策の補完：従来の障害対策手法に対して、補完的に Failure-Oblivious Computing を用いる。障害対策の漏れや、事前の想定を超えた障害があった場合に、スクリプトの実行が本提案によって継続される。
- 段階的な改善：当初は、Failure-Oblivious Computing に頼り、スクリプトの本質的な部分のみを記述する。その後、段階的に従来の障害対策手法に切り替えていくアプローチである。これは、クラウドの機敏性を重視した選択であるといえる。

4.2 障害の仮定

Failure-Oblivious Computing はすべての障害に対処できる万能な手法ではないため、本研究が対象とする障害に下記のような仮定をおく。

- 本研究では、スクリプト実行中に発生した障害を対象とする。これには、シェル上の対話的な操作や自動バッチ処理の双方が含まれる。それぞれの例は 3.1 節

の図 4 および図 5 に対応している。

- スクリプトは障害が発生しない限り、正しく動作すると仮定する。よって、スクリプト自身のバグによるソフトウェア障害は対象としない。ただし、障害対応については十分記述されていないとしても、Failure-Oblivious Computing により動作を継続するものとする。
- クラウド基盤ソフトウェア自身は正しく動作すると仮定する。3.2 節で述べたように、Kumoi 自身も障害を考慮して設計されている。また、万が一、基盤ソフトウェアに障害が発生する場合にも、Failure-Oblivious Computing 以外の対処方法で実現する必要がある。
- 本研究では、データセンタを構成するコンポーネントの障害を対象とする。これらの障害は、スクリプトの実行時に資源マップオブジェクトの例外として検出される。これらのコンポーネント障害には、ハードウェア障害および Kumoi 以外の外部のソフトウェア障害を含む。これらの例外は、資源ごとに障害検出器 (failure detectors) が検出し、送出する。障害検出器が例外を送出しなければ、そもそも障害に気づかないため、Failure-Oblivious 的な動作になるといえるが、その結果は管理されていないため、本手法の実行結果以上に予想できないものとなる。
- 本研究では、スクリプトの実行開始から実行完了までに発生した障害を対象とする。3.2 節で述べたように、物理計算機の障害は自動的にメンバシップ機構によって一定期間経過後、取り除かれる。また、その他のコンポーネントも物理計算機のオブジェクトから動的にたどることになるため、この際に障害があれば排除されている。ここで問題となるのは、資源マップオブジェクトの参照を取得してから、スクリプトの完了までに発生した障害である。

4.3 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 4.4 節で説明し、関数については 4.5 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 4.6 節で説明する。

4.4 オブジェクトの Failure-Oblivious 化

分散オブジェクトの Failure-Oblivious 化では、オリジナルの論文 [5] に近い方法を採用する。これはアドホックに見えるが、最も Failure-Oblivious Computing らしい手法である。

分散オブジェクトでは、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 7 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、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。たとえば、`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式的に生成する必要がある。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うるが、これらはすべ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生成を試みる。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 (gracefully) 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式的に生成する。このプロキシはすべ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舞いを行う。

なお、戻り値があるにもかかわらず、書き込み操作をともしない場合も考えられる。ただし、書き込み操作の場合は何も行わないこととしているため、通常の読み込み操作の場合と同じように対処する。

```

1: try {
2:   invoke(proxy, method, args) // RMI invoke
3: } catch {
4:   case e: Exception =>
5:     method.getReturnType match {
6:       // write op.
7:       case c if c == classOf[Unit] =>
8:         // read ops.
9:       case c if c == classOf[Int] => 0
10:      case c if c == classOf[Boolean] => false
11:      case c if c == classOf[String] => ""
12:      case c if c == classOf[List[_]] => List()
13:      case c if c == classOf[Option[_]] => None
14:      ...
15:    }
16: }

```

図 7 メソッド呼び出しの疑似コード

Fig. 7 Pseudo code for remote method invocations.

4.5 関数の Failure-Oblivious 化

本研究では、リスト操作を行う並列スケルトンも 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。

図 8 に実装の概要を示す。本実装も通常のスケルトンと同様にマスタ側の物理計算機でリストを分割し、複数のス

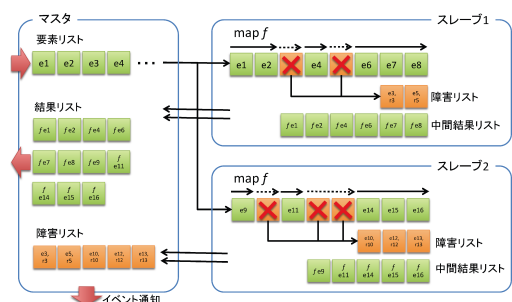


図 8 Failure-Oblivious 化された map() リスト操作関数

Fig. 8 Failure-Oblivious version of map() list operation.

表 1 Failure-Oblivious 対応のリスト操作関数
Table 1 Failure-Oblivious list operations.

関数	定義と副作用
fmap(f)	リストの各要素に対して関数 f を適用し、正常に適用できた要素の結果のみを含むリストを返却する。結果のリストの長さが入力のリストのものと異なることがある。また、要素の位置も異なることがある。
ffilter(p)	述語 p が真となるか正常に判定でき、かつ真となる要素のみを抽出する。障害が発生した要素は、結果のリストに含まれない。
freduce(f)	関数 f を使用し、正常に計算できるリストの各要素から 1 つの値に集約する。障害が発生した要素は、集約の結果に含まれない。
fcount(p)	述語 p が正常に判定でき、かつ真となる要素の数をカウントする。障害が発生した要素は、カウントの結果に含まれない。
fforeach(f)	関数 f を正常に適用できるリストのすべての要素に適用する。障害が発生した要素については、関数 f の適用が完了していない。
fexists(p)	述語 p が正常に判定でき、かつ真となる要素が存在する場合に全体の結果が真となる。障害が発生した要素については、偽となる。正常かつ述語 p を満たす要素があれば、真となる。
fforall(p)	すべての要素のについて述語 p が正常に判定でき、かつ真となる場合に全体の結果が真となる。障害が発生した要素が存在する場合には、全体の結果が偽となる。

レーブとなる計算機で並列に計算を行う。Kumoi のカーネルが動作している計算機はいずれもマスタおよびスレーブになることができ、シェルが接続している計算機が通常マスタとなる。スレーブについてはスケジューラによって自動的に選択される。また、計算対象が分散オブジェクトのリストの場合は、実際のオブジェクトがある計算機上で計算を行うようにスケジュールする。

並列スケルトンの Failure-Oblivious 化は、スレーブ側で障害が発生した要素を取り除き、正常な要素のみで計算するようにして実装している。この際、障害が発生した要素とその例外情報も 4.6 節で述べる機構によってイベント通知を行うため、正常な要素の計算結果とは別にリストで管理されている。すべてのスレーブの計算が完了すると、マスタ側で結果を結合し、スクリプトへの値の返却や障害のイベント通知を行う。

本実装では、基本的にリストの要素ごとに障害を管理している。また、計算中のスレーブ自身に障害が発生した場合や通信障害は、マスタがタイムアウトで検出し、委譲した部分リスト全体を障害とする。この場合も、タイムアウト発生イベント通知が行われるため、管理者が後ほど状況を確認することもできる。また、スレーブの障害対策の別の方法として、同じ部分リストを他のスレーブで再計算することも可能であるが、本システムでは分散オブジェクトのリストを計算対象とすることが多く、この場合、実オブジェクトも失われるため、現在はこのようにしている。さらに、マスタに障害が発生した場合は、シェル環境に結果をそもそも返却できないため、現在は考慮していない。

現在、Failure-Oblivious 化されているのは、表 1 の 7 つのリスト操作関数である。Failure-Oblivious 化されていることを示すため、**fmap()** や **ffilter()** などの **f** から始まる関数名としている。現在実装しているのは 7 つであるが、実装の労力上の問題からこの数となっており、その他

のデータ構造に対する操作関数も同じような手順で実装していくことができる。

これらの関数はすべて、正常に計算できた要素のみを結果に含むように計算を行う点で、元々のリスト操作関数の定義と異なっている。本研究では、無理に対処するよりも、スクリプト記述者が意識して使用することを期待しているが、主に 2 つの場合で問題となる可能性がある。

- 関数を単独で使用する場合：たとえば、**fmap()** 関数は、障害が発生した要素があると、入力のリストの長さと結果のリストの長さが異なる。また、結果の要素の位置が入力のものと異なることがあるため、これらに依存したスクリプトを記述すると問題となることがある。
- 複数の関数を組み合わせる場合：たとえば、**l.freduce(_+_) / l.length** のように平均を計算した場合に、**freduce()** のみで障害が発生し、**length** が正常に計算できる要素があれば、結果にズレが生じる可能性がある。このように、一方の関数で正常に計算でき、他方の関数で障害となる要素があると計算結果に誤りが生じる可能性がある。

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。Failure-Oblivious 版の関数を使用し、**ffilter()** の一部の要素で障害が発生したとすれば、それらを述語にマッチしなかったと見なし、結果のリストから取り除いて、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、**fmap()** 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返却する。

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る場合に調整が必要であることが分かる。この例の場合、**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、**ffilter()** 関数

```
FailureObliviousObject(interface kumoi.shell.vm.HotVM,migrateTo,
List(interface kumoi.shell.pm.HotPhysicalMachine,
class kumoi.shell.aaa.AAA),
List(ibmi, Anonymous),org.libvirt.LibvirtException:
POST operation failed: xend_post: error from xen daemon:
(xend.err '/usr/lib/xen/bin/xl_save 16 8 0 0 1 failed'))
```

(a) オブジェクト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)))
```

(b) リスト操作関数に障害が発生した場合

図 9 Failure-Oblivious イベント

Fig. 9 Failure-Oblivious events.

や `fmap()` 関数で正常な要素として計算してしまう。現在の Failure-Oblivious 化の実装では、障害が発生した場合に、オブジェクト側は模式的な値の生成を試み、関数側は結果から障害を取り除こうとする関係がある。後々のスクリプト実行に障害による影響をできるだけ伝搬させないという観点からは、関数側で対処させた方が有利であり、現在の実装では、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、通常どおり例外を発生させ、関数側で対処させるようにしている。

4.6 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 9 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。障害の発生はあらかじめ指定しておいたアクタにメッセージとして通知される。

これらのイベントはあくまでアドバイザリな通知であり、ただちに障害に対処するか、そのままにしておいて残りの計算機で実行を継続するか、管理者の判断に委ねられる。

5. 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。

5.1 導入の影響

5.1.1 導入の効果

本研究の方式の導入によって、クラウド管理における多数のスクリプトの実行継続性を改善することができる。一方で、本方式は VM や物理計算機などの障害に直接解決せ

ず、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。従来であれば、障害対策処理を記述しない場合、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、処理がまった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と障害が発生した計算機の 2 種類に近づける。障害が発生したコンポーネントについては、特に必要としなければそのままにしておくか、Kumoi が提供する資源マップオブジェクトからの情報の取得や操作機能を利用し、回復を試みる必要がある。

5.1.2 副作用

Failure-Oblivious Computing のオリジナルの論文 [5] でも深く議論されているように、副作用が潜在的に発生する。副作用が問題となるのは、障害が発生したオブジェクトの操作結果をもとに、正常なオブジェクトに対して何らかの操作を行う場合である。

クラウド環境において、VM やデータが削除されるなど重大な副作用も考えられなくはないが、下記の場合については副作用があまり問題とならず、Failure-Oblivious Computing が有効に機能すると考えられる。

- 複数の資源に対して並列に操作を行う場合：たとえば、資源プールから物理計算機を多数取得し、並列に VM を起動するなどの状況では、それぞれの資源の操作が独立しており、障害の影響が他の正常な結果に波及しにくい。このような状況では、副作用があまり問題とならず、むしろ全体のスクリプト実行継続の利点が大きくなる。
- 周期的に実行されるタスクの場合：物理計算機間での VM の負荷分散スケジューリングなどでは、スクリプトは 1 度しか実行されないのではなく、周期的に実行する必要がある。たとえば、副作用によって VM の負荷が一時的に不均一になっても、後のスクリプト実行によって修正されるのであれば、副作用はあまり深刻な問題とはならない。この周期実行による改善の考え方は、自己安定アルゴリズム [9] を参考にしている^{*1}。

Failure-Oblivious Computing は、すべての障害に対する万能な解決策ではなく、必要に応じて従来の障害対策手法やその他の障害対策手法と組み合わせる必要がある。Kumoi では、さまざまな障害対策手法を容易に組み込むことができることから、将来的に複数の障害対策手法を提供し、Failure-Oblivious Computing を選択的に利用できるようにする予定である。

5.2 導入コスト

3.1 節で説明したように、Kumoi の内部はオブジェクトと関数で非常に整理されているため、Failure-Oblivious

^{*1} ただし、自己安定性を満たすためには、いかなる状態からも正常な振舞いに戻ることを数学的に証明する必要がある。

Computing のような新しい概念を簡単に導入することができる。Kumoi 全体のコード行数 31,240 行のうち、Failure-Oblivious Computing 導入に要した変更は、わずか 445 行であった。このうち、オブジェクトの実装に関する変更は 168 行であり、関数の実装に係るものは 277 行であった。また、Failure-Oblivious Computing は必要に応じて、有効・無効を切り替えることができるように実装されている。

5.3 Failure-Oblivious Computing の改善

5.3.1 スクリプト記述ガイドライン

本研究では、スクリプト記述に一定のガイドラインを設けることで、できるだけ副作用の影響を抑え、導入の効果を高めることができる。主要なガイドラインとして好ましいものを以下に示す。なお、これらのガイドラインは、Failure-Oblivious Computing だけに限らず、障害対策全般に有効である。

- **状態の最小化**：3.2 節で説明した Kumoi 本体の設計と同様に、スクリプトも極力状態を保持しないように記述することが望ましい。たとえば、物理計算機のリストを、スクリプトの変数として長期間保持するのではなく、必要となるたびに `pms` から取得し、処理を行うなどである。また、周期的にスクリプトを実行する場合も、それらの間で状態を受け渡すのではなく、単一のスクリプト実行で処理ができるだけ完結していることが望ましい。
- **状態取得から利用までの期間の最小化**：資源マップオブジェクトの参照や情報を取得してから、利用するまでの期間はできるだけ短い方が望ましい。この期間が長くなるほど、途中で障害が発生する可能性は高まる。

5.3.2 意味論の活用

現在の実装では、オブジェクトに比べてリスト操作関数の方が、障害をうまく扱える。これは、オブジェクトは障害時にあらかじめ決められた値を生成するのに対して、リスト操作関数はそれぞれの関数の性質を利用して、対処しているからである。このように、意味論 (semantics) を活用した方が、障害をうまく扱える可能性があるが、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。

たとえば、図 4 では CPU 負荷の高い計算機を抽出しており、このあと、管理者に警告のメールを送信するなどの対応が考えられる。よって、障害の発生した計算機も抽出の結果に含めた方が都合がよい可能性がある。ただし、この判断は `ffilter()` 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。

また、オブジェクトでは、下記のようなアノテーションを付加することによって、あらかじめ決められた値しか生成しないという問題を多少改善できる。

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

```
1: def balanceOne = doMaybe {
2:   p1 <- pms.find(_.cpuRatio > higherLimit)
3:   v1 <- chooseOne(p1.vms)
4:   p2 <- pms.find(_.cpuRatio < lowerLimit)
5:   v1.migrateTo(p2)
6: }
```

図 10 Maybe モナドを使用した場合の擬似コード

Fig. 10 Maybe Monad example (Pseudo code).

ただし、この手法はそれぞれのメソッドに対して行わなければならない、上記の `ffilter()` と同じ問題が発生する。負荷の高い計算機を抽出する場合には 0.0 の方が望ましいことがあり、やはりスクリプト記述の意味に依存する。

5.3.3 モナドとの比較

Haskell などの関数型言語では、Maybe モナド [10] が採用されている。Maybe モナドを使用すれば、障害が発生した場合の処理を簡潔かつ明示的に記述することができる。Kumoi に関しても、Failure-Oblivious Computing と並行して、Maybe モナドのような機構を導入することができる。

Maybe モナドを導入した場合の擬似コードを、図 10 に示す。この擬似コードでは、`do` 構文を利用しており、負荷の高い計算機と負荷の低い計算機の間で VM を移送することで負荷を調整している。Maybe モナドを使用した場合には、各行の記述が成功した場合にのみ、以降の計算を進めるため、Failure-Oblivious Computing と異なり、全体の継続よりも安全性を優先しているといえる。よって、Failure-Oblivious Computing の方が好ましい場面、Maybe モナドの方が好ましい場面の双方が存在する。また、Maybe モナドを導入するためには、すべての資源マップオブジェクトの結果を `Option` 型に変更する必要がある。

さらに、4.5 節の `fmap()` の実装は、正常な結果を Haskell の `Just` 値、障害が発生した要素を `Nothing` 値とした場合の Haskell の `mapMaybe` 関数の動作と同じであり、その点でも非常に深く関連している。

5.4 他のミドルウェアへの適用

Failure-Oblivious Computing の概念は、Eucalyptus 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、簡単である。

5.5 オリジナルの提案との関係

5.5.1 オーバヘッド

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。

5.5.2 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。なぜなら、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。

5.5.3 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、当面の問題の対処のために、1度しか使用しないものも多く、これらに対しても障害回復処理を記述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は非常に複雑であり、すべての障害を予測し、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。

6. 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、クラウドを提供するデータセンタのさまざまな管理に関するスクリプトを対象に実験を行った。

6.1 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, 32 GB の SCSI ディスクで構成され、すべて 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 0.6.3, 0.8.2, Scala 2.9.2 を

```
1: def deploy(pms: List[HotPhysicalMachine]) =
2:   pms.fmap { p =>
3:     val v = p.vmm.createVM
4:     v.name = "centos6-" + p.name
5:     v.memory = 256 * 1024 * 1024
6:     v.add(FileDiskAuto("/nfs/images/centos6-" +
7:       p.name + ".img"))
8:     v.add(NatAuto)
9:     p.vmm.add(v)
10:  }
```

図 11 VM の起動スクリプト

Fig. 11 Script for VM deployment.

```
scala> deploy(pms)
reactions: FailureObliviousList('map,List(<function1>,
List((ibm5,java.rmi.UnexpectedException: unexpected
exception; nested exception is:
    org.libvirt.LibvirtException: Unknown failure)))

res1: List[kumoi.shell.vm.HotVM] = List(centos6-ibm1,
centos6-ibm2, centos6-ibm3, centos6-ibm4, centos6-ibm6,
centos6-ibm7)
```

(a) Failure-Oblivious Computing を有効にした場合

```
scala> deploy(pms)
java.rmi.UnexpectedException: unexpected exception;
nested exception is:
    org.libvirt.LibvirtException: Unknown failure
...

scala> pms.map(p => p.name -> p.vms)
res2: List[(String, List[kumoi.shell.vm.HotVM])] =
List((ibm1,List(centos6-ibm1)), (ibm2,List(centos6-ibm2)),
(ibm3,List(centos6-ibm3)), (ibm4,List(centos6-ibm4)),
(ibm5,List()), (ibm6,List()), (ibm7,List()))
```

(b) 無効にした場合

図 12 VM 起動スクリプトの実行結果

Fig. 12 VM deployment results.

使用した。

6.2 実験結果：VM ライフサイクル管理

(a) VM の起動：図 11 のスクリプトに対して、意図的に 2.1 節の Unknown Failure を生じさせる物理計算機 (ibm5) を 1 台用意し、実験を行った。Failure-Oblivious Computing を使用した場合、使用しなかった場合の双方のスクリプト実行の結果を、図 12 に示す。Failure-Oblivious Computing を使用した場合には、正常なものについてはすべて起動したため、起動した VM は 6 台であった。一方で、使用しなかった場合には、例外でスクリプトの実行が停止するため、ibm5 以降の VM は起動しなかった。よって、起動した VM は 4 台となった。

```
scala> compact(pms)
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)
res4: 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()))
```

(a) Failure-Oblivious Computing を有効にした場合

```
scala> compact(pms)
java.rmi.UnexpectedException: unexpected exception;
nested exception is:
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)
res5: List[(String, List[kumoi.shell.vm.HotVM])] =
List((ibm1,List(centos6-ibm1, centos6-ibm7,
centos6-ibm6)), (ibm2,List(centos6-ibm2)),
(ibm3, List(centos6-ibm3)), (ibm4,List(centos6-ibm4)),
(ibm5, List()), (ibm6,List()), (ibm7,List()))
```

(b) 無効にした場合

図 13 VM 集約スクリプトの実行結果 (1)

Fig. 13 Execution results of VM compaction (1).

(b) VM の集約: 3.1 節の図 5 のスクリプトに対して、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 つの場合がありうるため、双方について実験を行った。

- VM の障害の場合: この場合の実行結果を図 13 に示す。図 5 の 15 行目の直前で VM (centos6-ibm5) に障害を注入した場合は、Failure-Oblivious 化により `v.cpuRatio` の値がつねに 0.0 となり、どの計算機も移送先に該当する。ただし、図 5 の 6 行目の VM の移送を行う `migrateTo()` メソッドは実際には何も行わないため、問題とならなかった。よって、centos6-ibm5 を除くすべての VM が、物理計算機 ibm1 に集約されている。

一方で、Failure-Oblivious Computing を無効化した場合には、centos6-ibm5 を操作した段階でスクリプトの実行が停止するため、以降の 3 台の VM は元の物理計算機上で実行を続けた。

- 物理計算機の障害の場合: この場合の実行結果を図 14 に示す。20 行目の直前で障害を注入した物理計算機

```
scala> compact(pms)
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)
res7: 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()))
```

(a) Failure-Oblivious Computing を有効にした場合

```
scala> compact(pms)
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)
res6: List[(String, List[kumoi.shell.vm.HotVM])] =
List((ibm2,List(centos6-ibm2)), (ibm3,List(centos6-ibm3)),
(ibm4,List(centos6-ibm4)), (ibm5,List(centos6-ibm5)),
(ibm6,List(centos6-ibm6)), (ibm7,List(centos6-ibm7)))
```

(b) 無効にした場合

図 14 VM 集約スクリプトの実行結果 (2)

Fig. 14 Execution results of VM compaction (2).

```
1: def shutdown(pms: List[HotPhysicalMachine]) {
2:   pms.fmap(_.vms.map(_.shutdown))
3: }
```

図 15 VM の終了スクリプト

Fig. 15 Script for VM shutdown.

(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 化によりつねに 0.0 のため、移送の対象とならず、物理計算機 ibm2 に centos6-ibm1 を除くすべての VM が集約された。

一方で、Failure-Oblivious Computing を無効にした場合は、ibm1 を操作した段階でスクリプトの実行が停止するため、VM は元の物理計算機上で実行を続けた。

(c) VM の終了: 図 15 のスクリプトを実行した結果を図 16 に示す。この場合、`shutdown` メソッドは VM の終了を待たないため、元々 Failure-Oblivious 的であるといえる。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。

```
scala> shutdown(pms)
```

(a) Failure-Oblivious Computing を有効にした場合

```
scala> shutdown(pms)
```

(b) 無効にした場合

図 16 VM 終了スクリプトの実行結果

Fig. 16 Execution results of VM shutdown.

```
1: def schedule(v: ColdVM) = {
2:   val candid = pms.ffilter(_ .availableFor(v))
3:   val sorted = candid.sortWith(rank)
4:   val top = sorted.head
5:   top.vmm.add(v)
6: }
```

図 17 順位スケジューラによる VM の配置

Fig. 17 Example of VM rank scheduler.

6.3 実験結果：クラウドの提供

クラウドの定義 [11] を満たすため、最近、Kumoi に導入された資源プールや VM のインスタンス数の自動増減などの機能に対しても、Failure-Oblivious Computing を適用することができる。Kumoi では、これらの機能もスクリプトとして記述されており、本実験では実際に Kumoi で使用されているスクリプトを多少、簡略化したものの結果について示す。

(d) 資源プールのスケジューラ：図 17 に、Kumoi での VM の順位スケジューラの記述例を示す。順位スケジューラは、ある関数で物理計算機に順位を付け、先頭の計算機に VM を配置する。順位付け関数の実装次第でさまざまな場合に対応できるため、OpenNebula をはじめとするクラウド基盤ソフトウェアで採用されている。

図 17 のスクリプトの場合、障害が問題となるのは大きく分けて下記の 3 つである。

- (1) **VM に障害が発生する場合**：この場合、スケジュールの対象となる VM にそもそも障害が発生しているため、VM は実行されない。ただし、スクリプトの実行は Failure-Oblivious 化によって最後まで継続される。
- (2) **2 行目で物理計算機に障害が発生する場合**：この場合、障害が発生した計算機は `ffilter()` 関数で取り除かれるため、以降の処理に影響を与えない。
- (3) **3–5 行目で物理計算機に障害が発生する場合**：スクリプトの実行は最後まで継続するが、障害が発生した物理計算機が順位付けの最上位となった場合には、VM の起動に失敗する。

ただし、障害が発生したオブジェクトは各型の 0 を返却するため、順位付けで最上位とならない可能性も高い。図 18 にこの場合の 1 つの実行結果を示す。この例では、2 行目でメモリ空き容量によってフィルタを行い、3 行目で CPU の空き使用率 `cpuAvailable` をもとにソートを行っている。障害が発生した計算機については、`cpuAvailable` が 0.0 となるため、VM の起動先とはならず、問題が発生していない。

(1) と (3) の場合に対処するには、再試行が必要であると思われる。5.1.2 項で副作用について議論したように、Failure-Oblivious Computing が得意とするのは複数

```
scala> schedule(v)
reactions: FailureObliviousObject(interface kumoi.shell.
pm.HotPhysicalMachine,cpuAvailable,List(class kumoi.
shell.aaa.AAA),List(Anonymous),java.rmi.ConnectException:
Connection refused to host: 10.0.0.1;
nested exception is:
java.net.ConnectException: Connection refused)
... 6 times
res5: kumoi.shell.vm.HotVM = centos6
```

図 18 順位スケジューラの実行結果

Fig. 18 Execution result of VM rank scheduler.

の資源に対して並列操作を行う場合や、周期的な実行を行う場合である。2 行目で障害が発生せず、3 行目以降で障害が発生するのは、希ではあるが、この例は Failure-Oblivious Computing が比較的苦手な場合といえる。

(e) **VM の伸縮機能**：図 19 に Kumoi での VM インスタンス伸縮機構の実装例を示す。これは、Amazon EC2 の自動スケール機能を参考に実装されている。図 19 に登場する `pool` は物理計算機の資源プールである。なお、本来ならば、資源プールや、VM マスタ、Actor の障害についても考慮する必要があるが、これらは Failure-Oblivious Computing の範囲を超えるため、今後の課題とする。これらは、Zookeeper [12] や Paxos [13] などを使用し、基盤ソフトウェア自身が複製を行うことで実現する必要がある。

図 19 の場合、障害が問題となるのは大きく分けて下記の 2 つである。

- **6 行目で VM に障害が発生する場合**：この場合、`ffilter()` 関数によって障害が発生した VM が取り除かれるため、以降の処理に影響を与えない。
- **7–8 行目で VM に障害が発生する場合**：この場合、分子の障害が発生した VM の `cpuRatio` は 0.0 となるが、分母の VM の台数には含まれるため、平均 CPU 使用率が低めに計算される。VM のインスタンス数が正しい計算の際よりも少なくなる可能性があるが、後々の繰返し実行によって修正されることが期待される。

2 つ目の場合について、実行結果を図 20 に示す。障害を意図的に発生させているが、スクリプトの実行は最後まで継続された。

```

1: actor {
2:   val pref = masterVM.name + "-clone"
3:   var breachCount = 0
4:   loop {
5:     val vms =
6:       pool.vms.filter(_.name.startsWith(pref))
7:     val mean = vms.foldLeft(0.0)(_+_.cpuRatio)/
8:       vms.length.toDouble
9:     if (mean > upperThreshold) {
10:      breachCount += 1
11:      if (breachCount >= breachDuration &&
12:        vms.length < maxVMs) {
13:        pool.add(masterVM.clone)
14:        breachCount = 0
15:      }
16:    } else if (mean < lowerThreshold) {
17:      breachCount -= 1
18:      if (breachCount <= -breachDuration &&
19:        minVMs < vms.length) {
20:        pool.remove(vms.last)
21:        breachCount = 0
22:      }
23:    } else {
24:      if (vms.length < minVMs)
25:        pool.add(masterVM.clone)
26:      breachCount = 0
27:    }
28:    sleep(interval)
29:  }
30: }

```

図 19 VM インスタンス数の自動的な増減

Fig. 19 Script for elasticity of VMs.

```

...
reactions: FailureObliviousObject(interface kumoi.shell.
vm.HotVM,cpuRatio,List(class kumoi.shell.aaa.AAA),
List(Anonymous),java.rmi.ConnectException: Connection
refused to host: 10.0.0.1; nested exception is:
java.net.ConnectException: Connection refused)
...

```

図 20 VM インスタンス数増減の実行結果

Fig. 20 VM elasticity result.

(f) VM の課金処理：Kumoi 内部の VM の資源使用量の課金処理では、図 21 とほぼ同様のコードで記述されている。Failure-Oblivious 化を行った場合には、図 22 のようなイベントが通知されるが、課金処理の実行は継続される。本課金処理は独立した Actor として記述されているため、VM の起動直後や終了直後に障害が起こりうる。なお、起動直後などで使用量が取得できなかった VM については、次の課金処理のサイクルの実行時に課金されることになる。

```

1: actor {
2:   loop {
3:     vms.map { v =>
4:       genericAccount(v, v.info)
5:       for (e <- v.ethStats) netAccount(v, e)
6:       for (d <- v.diskStats) diskAccount(v, d)
7:     }
8:     sleep(interval)
9:   }
10: }

```

図 21 VM の課金処理

Fig. 21 Example script for VM accounting.

```

reactions: VirtualMachineAdded(centos3)
...
reactions: FailureObliviousObject(interface kumoi.shell.
vm.HotVM,diskStats,List(class kumoi.shell.aaa.AAA),
List(kumoi.impl.aaa.RootAAAImp1$@16cbcd92),
org.libvirt.LibvirtException:
internal error read_bd_stats: Failed to read any block
statistics)
...

```

図 22 VM 課金処理の実行結果

Fig. 22 Execution result of VM accounting.

表 2 スクリプト中の障害を考慮すべき箇所

Table 2 Possible failure points in various scripts.

	スクリプト	行数	潜在的な障害箇所
(a)	VM の起動	10	10
(b)	VM の集約	21	4
(c)	VM の終了	3	2
(d)	VM のスケジュール	6	3 以上
(e)	VM の伸縮	30	18
(f)	VM の課金	10	4

6.4 障害対策処理の軽減について

本研究では、障害の仮定を 4.2 節のようにおいたため、スクリプト中で資源マップオブジェクトの操作を行っている箇所を障害要因箇所として考える必要がある。表 2 にこれまでの各スクリプトにおいて、潜在的に障害が発生する箇所を示す。(e)については、availableFor(), rank() 内での資源マップオブジェクトのメソッド呼び出しも考慮しないとならないため、3 カ所以上としている。

本研究では、これらの箇所に対して特に障害対策の記述を行わず、全体のスクリプトの実行継続を実現している。一方で、従来の障害対策手法を採用する場合には、個別の箇所について影響を分析し、障害の対処方法を使い分け、対策を実施する必要がある。

7. 関連研究

本章では, Failure-Oblivious Computing のデータセンタ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 [14] 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 [15] 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.

Taura らの GXP [16] 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.

8. 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やクラウドの提供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, Failure-Oblivious Computing のさらなる評価や改良を進めるとともに, その他の障害対策手法の導入も進める. 特に, 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も検討する. 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. また, Kumoi では, さまざまな障害対策手法を組み込みやすくなっており, 将来的にはさまざまな手法を選択的に利用できるようにする予定である.

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] Nurmi, D. 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), available from <http://www.opennebula.org/>.
- [4] Rackspace Cloud Computing: OpenStack: The Open Source, Open Standards Cloud (2010), available from <http://openstack.org/>.
- [5] Rinard, M. et al.: Enhancing server availability and security through failure-oblivious computing, *USENIX OSDI'04*, p.21 (2004).
- [6] Sugiki, A. 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), available from <http://www.scala-lang.org/>.
- [9] Dolev, S.: *Self-Stabilization*, The MIT Press (2000).
- [10] O'Sullivan, B., Goerzen, J. and Stewart, D.: *Real World Haskell*, O'Reilly Media, Inc. (2008).
- [11] National Institute of Standards and Technology: NIST Definition of Cloud Computing (2009), available from <http://www.nist.gov/itl/cloud/>.
- [12] Hunt, P. et al.: ZooKeeper: Wait-free Coordination for Internet-scale Systems, *USENIX Annual Tech. Conf.'10*, p.11 (2010).
- [13] Lamport, L.: The Part-time Parliament, *ACM TOCS*, Vol.16, No.2, pp.133-169 (1998).
- [14] Pike, R. 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).
- [15] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *USENIX OSDI'04*, pp.137-150 (2004).
- [16] Taura, K.: GXP: An Interactive Shell for the Grid Environment, *IEEE IWIA'04*, pp.59-67 (2004).



杉木 章義 (正会員)

2002 年電気通信大学情報工学科卒業. 2004 年同大学大学院情報工学専攻博士前期課程修了. 2007 年同博士後期課程修了. 博士 (工学). 2007 年科学技術振興機構 CREST 研究員, 2009 年筑波大学システム情報工学研究科助教. 分散システム, オペレーティングシステムの研究に従事. 2009 年度山下記念賞受賞. 日本ソフトウェア科学会, ACM, IEEE-CS, USENIX 各会員.



奥畑 聡仁

2011 年筑波大学情報学群情報科学類卒業。現在、同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博士前期課程に在籍中。



加藤 和彦 (正会員)

1989 年東京大学理学部情報科学科助手，1993 年筑波大学電子・情報工学系講師，1996 年同助教授，2004 年筑波大学大学院システム情報工学研究科教授，現在に至る。1992 年博士（理学）（東京大学）。オペレーティングシステム，分散システム，仮想計算環境，セキュリティに興味を持つ。

1990 年情報処理学会学術奨励賞，1992 年同研究賞，2005 年同論文賞，2004 年日本ソフトウェア科学会論文賞，各受賞。