脅威トレースの並列分散化

松元拓也[1]　三牧麻美[2]　神武克海[2]　小出洋[3]

概要：大規模なネットワークでの脅威トレースの実行時間を短縮するため，脅威トレースをAkka を用いた分散環境に適用する．脅威トレースは，システムの脆弱性を狙った個人や企業への標的型攻撃に資することを目的としている．脅威トレースはネットワークや計算機をモデル化し，マルウェアの動きをシミュレートすることで，脆弱性の発見や対策に活用することが可能である．現在の脅威トレースは単一の計算機上で実行されているが，実際の大規模なネットワークをシミュレートするには処理能力がまだ及ばない．脅威トレースは Scala の Actor モデルを使用して実装されているが，Akka と呼ばれるフレームワークの Actor モデルを使用すると分散環境への適用において耐障害性に優れた構築が可能であるなど数々の利点が得られる．そこで、脅威トレースを Akka を用いて並列分散化した．今後の課題として動的スケジューリングがあり，各計算機の負荷を調査し，負荷に応じたスケジューリングを検討する必要がある．

キーワード：Akka，Scala，脅威トレース，標的型攻撃，セキュリティ，並列分散処理

An Implementation of Parallel and Distributed Attacks Tracer

Takuya Matsumoto[1]　Asami Mimaki[2]　Katsumi Kotake[2]　Hiroshi Koide[3]

Abstract: We apply our attacks tracer to a parallel and distributed environment by using Akka to reduce the execution time of tracing attacks in large networks.The attacks tracer will contribute to take effective measures to counter targeted attacks or APT (Advanced Persistent Threat) on information systems of enterprises. In the attacks tracer, it makes models of networks and computers to simulate behaviors of malwares. The attacks traceser discovers vulnerabilities of information systems and we utilizes them to measure to counter attacks. Current implementation of our attacks tracer takes long time execution time to trace behaviours of malwares on practical networks, because it is executed on a single computer. We have implemented the attacks tracer based on actor model. We also present a new implementation using a new framework based on actor models, Akka, because Akka has a lot of fine characteristics, like applicable abilities to parallel and distribution environments. We will present and implement dynamic scheduling methods which investigate load works of each processors and assign tasks to processors considering the load works.

Keywords: Akka, Scala, attacks tracer, APT(Advanced Persistent Threat), Security, Parallel and Distributed Processing

1. はじめに

近年，個人や企業において標的型攻撃による内部情報の不正取得，漏洩が問題となっている．標的型攻撃に対抗するために様々な研究が行われており，対策が公開されている [3]．主に対策としてマルウェアをシステム内に侵入させないために行う「入口対策」や，内部に侵入した際にその侵入拡大を防止する「内部対策」，システム内の情報をシステム外へ持ちださせないために行う「出口対策」などが行われている．

対策を検討する際の手助けとして，脅威（攻撃者やマルウェア）がどのように内部情報を取得していくかについてのシミュレーションを自動化する事 (以下これを「脅威トレース」と呼ぶ) が行われている．脅威トレースはシステム内部を網羅的に探索でき，脆弱性発見の手助けとなるので内部対策や出口対策に有効である．

しかし，現在の脅威トレースは分散環境上では実行されておらず，今後の大規模なネットワークのシミュレート時間の増大への対応が難しい．本研究の目的は，Akka という優れた分散性，並列性があるフレームワークを用いて脅威トレースを分散環境に適用し，大規模なネットワークのシミュレート時間を短縮させることである．

メッセージパッシングによるメソッド呼び出しを行うように設計し，実装を行うことで分散環境に適用することができた．今後，動的スケジューリング手法も適用して脅威トレースの効率的な実行について考察する予定である．

2. 背景

2.1 APT 攻撃

近年，重要情報の不正取得を目的にしたサイバー攻撃が行われている．特定の個人や企業を狙った攻撃は標的型攻撃と呼ばれ，その攻撃手法は複数存在する．

よく行われている手法として，フィッシングメールと不正プログラム（マルウェア）を利用した手法がある．攻撃者は企業の業務連絡メールを盗み，それを元にマルウェアを仕込んだメールを社員に送信することで社員は何の疑いもなくそのメールを開いてしまい，マルウェアに感染させることができてしまう．大勢いる社員のうち１人でもメール開いてしまうとその企業のネットワークはマルウェアに感染してしまう．この他にも公的機関を詐称したメールを送るなど，非常に危険で巧妙であることがわかる．このような攻撃手法は年々増加の一途をたどっており，APT 攻撃 (Advanced Persistent Threats) と呼ばれている [2]．

このような攻撃への対策も研究されており，入口対策，内部対策，出口対策などに分けられている．

入口対策　入口対策とは情報システム内に攻撃者 (マルウェア) を侵入させないように行う対策であり，主にファイアウォールやアンチウイルスソフトを設置することである．

内部対策　内部対策とは攻撃者が内部に侵入した際，システムの調査などを行いづらくする事や，管理者が侵入を早期に発見するために行う対策であり，ハニーポットを使用した対策があげられる．

出口対策　出口対策とはシステム内部の情報を外部に送信するのを阻害し，流出させないために行う対策であり，各プロキシやゲートウェイで通信を監視し外部に送信させないようにする方法がある．

しかし，前述した例のように APT 攻撃では入口対策を突破される恐れがあり，その際に内部対策，出口対策によって脅威に対処し情報流出を防ぐという考え方が大切である．入口対策だけではなく内部対策，出口対策も適切に行うことで，近年の高度な標的型攻撃に対する耐性を向上させることが可能である [3]．

2.2 脅威トレース

「脅威トレース」とは広義において，仮定した情報ネットワーク内でマルウェアがどのように情

[1] 九州工業大学情報工学部知能情報工学科
Department of Artificial Intelligence, Kyushu Institute of Technology
[2] 九州工業大学大学院情報工学府情報創成工学専攻
Department of Creative Informatics, Kyushu Institute of Technology
[3] 九州工業大学大学院情報工学研究院情報創成研究系
Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology

報を探索して見つけ出し，その情報を外部に送信するかをシミュレートし，そこで得た結果を利用してシステムの脆弱性などを考察する手助けを行うことである．

狭義では，脅威トレースで仮定するネットワークをモデル化し，トレーサの中でマルウェアの挙動を自動的かつ網羅的にシミュレートすることで，マルウェアがネットワークに与える影響などを調査するアプリケーションである．このアプリケーションを利用することでマルウェアの挙動を各ノード間で網羅的に探索し，様々なパターンの解析が可能である．

その特徴から未知のマルウェアの挙動の調査や，新たなシステムの脆弱性の発見・報告に役立てることが可能であり，内部対策，出口対策において有効であるとされている [4].

現在の脅威トレースは研究段階であるが，今後実際のネットワークをシミュレートし，脅威への対策に資するものと考えている．

3. 本研究があつかう問題

現在の脅威トレースは 1 台のコンピュータ上でしか実行できない．今後，実際の大規模なネットワークに適用される際，トレースにかかる時間はノード数の増加により，更に増大していく．網羅的にトレースを行うことにより，ノード数の増加は探索に大きな影響を与える．更に今後，脅威トレースに新しい機能が実装されることも考えると 1 ノードあたりのトレース時間も増大し，その合計時間も更に増大する．

そこで，脅威トレースを分散環境に適用することで，各々の計算機に負荷を分散させ大規模なネットワークのシミュレートに対応出来るようにする．

4. 脅威トレースの分散化

4.1 Akka の利用

脅威トレースを分散環境に適用するにあたり，Akka というフレームワークを使用する．Akka とは，Scala／Java で非同期処理を行うためのフレームワークで，Scala の開発元である Typesafe 社が中心となって開発している [5]．Akka を使用する利点として以下のものがある [6].

並列処理のためのシンプルで高レベルの抽象化
メッセージパッシングを行う Actor モデルを使用することで並列処理が自動的に行われ，個々の Actor がメッセージでデータのやり取りを行う．

自己修復などによる優れた耐障害性
Actor の処理中にエラーが起きても再実行し，停止してしまってもその情報は親 Actor に通知され，Actor を再起動するなどの修復が可能である．

優れた負荷分散性
設定ファイルによって Actor の生成先やメッセージ処理方法などが指定可能である．

ノンブロッキング処理
送信側はメッセージを送信するだけなので，返事を待つ間に他の処理を行うことが可能である．受け取ったメッセージはメールボックスに入り，そこからメッセージを取り出して処理を行うので，互いに処理を中断することがない．

現状の脅威トレースは Scala の Actor モデルを使用して設計されており，Akka にも Actor モデルが利用されている．また Akka を利用すると，設定ファイルに分散環境でのデプロイ先を記述することで任意の Actor モデルを分散環境上で実行させることが可能である．そこで，Scala の Actor モデルを Akka の Actor モデルに変更し，適切な設定ファイルを作成することで分散化を行うように実装する．

4.2 メッセージパッシングによる処理の実行

脅威トレースの処理は，Actor モデルのメッセージパッシングを利用して実現する．前節でも述べた様に，現在の脅威トレースは Scala の Actor モデルを利用している．これを利用することで，ネットワーク (各ノード) やアプリケーションなどがそれぞれ Actor として動作する．

その中で，各々の Actor の設定を行う際はそれぞれが持っているメソッドを直接呼びだす実装になっている．フィールドの参照，更新についても同様に直接呼び出している (図 1).

純粋な Actor モデルであればメッセージパッシング利用し受け取ったメッセージのみ処理すれば

```
val echoactor = new EchoActor()
echoactor.setName("echo")

echoactor ! "Hello!"
```

図 1 Scala Actor の利用.
Fig.1: Using Actor of Scala.

良いので，他との依存関係を持たず独立性があると言える．独立性があれば同時に実行することが可能なので分散化，並列化しやすい．しかし，脅威トレースで使用されている Actor は他の Actor の結果を必要とすることから依存関係が拭いきれず，純粋な Actor モデルではない．

そして，Akka の Actor モデルはメッセージパッシングしかできず，Actor の各メソッド呼び出しやフィールドへのアクセスができない．Actor へのアクセスは ActorRef と呼ばれるオブジェクトからメッセージを介して行う (図 2).

脅威トレースで Actor に対して直接参照している

```
val sys = ActorSystem.create()
val ref = sys.actorOf(Props[
    EchoActor],"echo")
//ref.setName("echo") //impossible

ref ! "Hello!"
```

図 2 Akka Actor の利用.
Fig.2: Using Actor of Akka.

部分を，メッセージパッシングを利用してメソッド呼び出しやフィールド参照，更新を行うように変更する．

具体的には，メソッド呼び出しやフィールド参照，更新に関してメッセージを定義しておき，メソッドなどを呼ぶ代わりにこのメッセージを送信する．Actor 側では受信したメッセージを元に場合分けを行い，適切な処理を行う．これにより同等の処理を，メッセージパッシングで実現することが可能である．

図 3 に示したように，従来では Actor の各メソッド，各フィールドについて直接呼び出しを行って

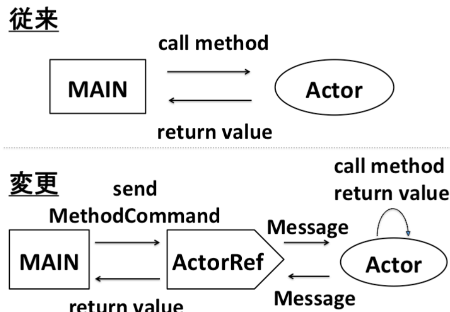

図 3 メッセージパッシングによる処理の実行.
Fig.3: Execution of the process by message passing.

いたが，変更後では ActorRef を介して Actor にメッセージを送信する．そしてメッセージに応じて処理を行い結果をメッセージで返信する．

4.3 スケジューリングを考慮した設計

適切な分散処理を行うために各分散コンピュータ上に親となるコントローラを作成する．このコントローラ自身も Actor モデルを使用する．そして，コントローラは以下の機能を実装している．

ノードの管理
このコントローラを親として子ノードの作成や起動，終了が行える．

負荷の監視
このコントローラ下に作成されている子ノードの処理状態を管理し把握する．

次に全ての分散環境のコントローラを持つスケジューラクラスを作成する．このスケジューラは以下の機能を実装している．

各分散環境の状態チェック
定期的にコントローラの負荷状態をチェックし，必要に応じて処理を切り替える．

マイグレーションの実施
負荷に応じてコントローラ下にあるノードを他のコントローラ下に移動する．

以上のような設計を行うことで，スケジューリングを考える際の情報を得ることができる．

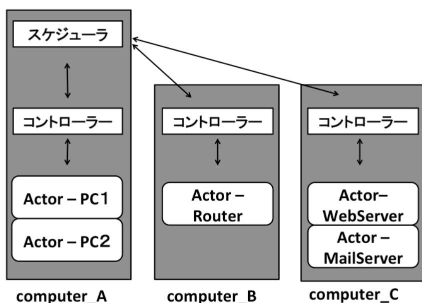

図 4 スケジューリングの考慮.
Fig.4: Consideration of scheduling.

5. 実装

5.1 現状のメソッド呼び出しによるメッセージ送信の実装

メッセージパッシングによる処理を，現状の脅威トレースのような手続き的なメソッド呼び出しで実装する．設計指針に基づき実装を行う中で，ノードに関する全てのメソッド呼び出しやフィールド参照を全てメッセージパッシングに置き換える必要が生じた．このまま順に全てを変更すると，コード量も増え可読性の低下を招く (図 5).

```
val net = Network(...)
net ! MethodCommand("addNode",pc1)
```

図 5 メッセージパッシング.
Fig.5: Message Passing.

そのため，現状のソースコードのような手続き的なメソッド呼び出しを維持するように，メッセージパッシングを普通のメソッド呼び出しやフィールド参照の様に呼び出す変更を行いたい．そこで，Scala の Method Missing を利用してメッセージパッシングの処理を自動的に行うように変更した．

Method Missing とはメソッドやフィールド変数が見つからないとき，代わりに呼ばれる関数のことで，Scala では scala.language.dynamics をインポートし，Dynamic をミックスインすることで利用が可能である (図 6).

ノードへの `ActorRef` をフィールドに持つクラスを作成し，そのクラスに Method Missing を実装する．その内部ではメソッド名，フィールド名に応じたメッセージを作成しノードに送信する．これにより現状の脅威トレースのソースコードを維持した設計が可能である (図 7).

```
class MethodMissing extends Dynamic{
    def applyDynamic(methodName:String)
        (args:Any*){
        //Method Call
    }
    def updateDynamic(key: String)(
        value: Any) = {
        //set Feild
    }
    def selectDynamic(key: String): Any
         = {
        //get Feild
    }
}
```

図 6 Method Missing の例.
Fig.6: Examples of Method Missing.

```
val net = new NetWorkRef(...)
net.addNode(pc1)
```

図 7 Method Missing による手続き的な関数呼び出し.
Fig.7: Procedural function call by the Method Missing.

5.2 メッセージの動的処理

Actor が受け取ったメッセージを動的に処理する様に実装を行う．メッセージパッシングによる送信処理の方は先ほど解決したが，一方でメッセージを受信したノード側ではそのメッセージに応じて処理を変えなくてはならない．これについても同様に，全てのメソッド呼び出しやフィールド参照のコマンドについての処理を順に追加しなくてはならない．そこで，Scala の Reflection を使用してメッセージの受信処理を自動的に行えるように変更を行った．Reflection とはメソッド名やフィールド名から動的にそのメソッドを実行したり，フィールドの値を読み書きできる機能である．

コマンドにメソッド名 (フィールド名) や引数などの情報が含まれる様に変更し，その情報から動的に処理を行えれば比較的少量の実装で済み，可読性を下げない．

具体的には，メッセージとしてメソッド呼び出しコマンドとフィールド操作コマンドの 2 つを用意する．メソッド呼び出しコマンドを受信した場合はコマンドからメソッド名と引数を受け取り，その名前と引数の型情報を利用してランタイムミラーから実行するメソッドを選択する．そして，受信したノードのインスタンスからインスタンスミラーを作成しメソッドを実行する (図 8).

```
//[ this.methodname(argument) ]

//Get RuntimeMirror
val rm = runtimeMirror(getClass.
    getClassLoader)

//Get Method
val t = rm.classSymbol(getClass).toType
val m = t.member(newTermName(methodname
    )).asMethod

//Get InstanceMirror
val im = runtimemirror.reflect(this)

//Execute Method
val mm = im.reflectMethod(m)
mm(argument)
```

図 8 Reflection によるメソッド呼び出し．
Fig.8: The method calls by Reflection.

フィールド操作コマンドを受け取った場合でも同様に，コマンドからフィールド名や引数を受け取る．この時引数がないときはフィールドの参照，引数があるときはフィールドの更新と考える．フィールド操作の場合，ランタイムミラーにはフィールドのゲッターメソッドやセッターメソッドがあるので，引数に応じて適切なメソッドを選択する．後は先ほどと同様に実行していく．

これらの設計を行うことで受信したメッセージに応じて動的に処理を行うことが可能である．

6. 実験

6.1 実験方法

分散環境に適用した脅威トレースでシミュレーションを行い正しく分散化されているかを確認する．使用したネットワークモデルを図 9 に示す．

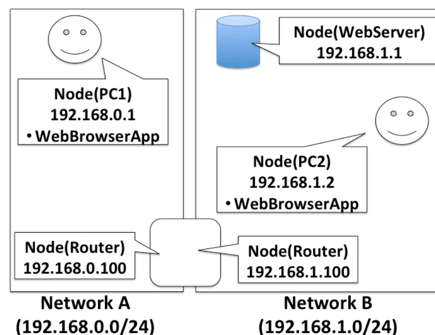

図 9 使用したネットワークモデル．
Fig.9: Network model which we use.

ネットワークモデルを構築する際に生成される Actor を以下のように分散する．

コンピュータ 1

Node（PC1），WebBrowserApp（PC1），Network（A），Node（Router）

コンピュータ 2

Node（PC2），WebBrowserApp（PC2），Network（B），Node（WebServer），WebServerApp

Actor を生成する際に以上の分散環境となるように，各コンピュータのコントローラを介して生成を行う．

正しく分散化されているかどうかは，それぞれのコンピュータに出力されるログで確認を行った．メッセージパッシングを行う際にメッセージの送信前と受信後にログを出力する．それぞれのログから，メッセージがもう一方のコンピュータに送信されている事を確認する．

また，ブラウザの設定などに呼び出される，「setApplication」もログを出力するように設定した．これによりメッセージパッシングによって対象のメソッドが正しく呼び出されたかについても確認できる．

実験は個々の Actor（Node や Network など）を各分散環境で作成した後，以下の操作を行う．

- Network に Node を追加（`addNode`）
- Node(Router) にルーティング情報を設定 (`addRoute`)
- Node(PC1,PC2,WebServer) にアプリケーションを設定 (`setApplication`)

そして，実行した後に画面に出力されるログを確認する．

6.2 実験環境

今回実験に使用するコンピュータ 1（表 1），コンピュータ 2（表 2）の詳細を以下に示す．

表 1 コンピュータ 1 の詳細.
Table1: Details of Computer 1.

項目	性能
OS	OS X 10.9.5（13F34）
プロセッサ	1.8 GHz Intel Core i7
メモリ	4 GB 1333 MHz DDR3

表 2 コンピュータ 2 の詳細.
Table2: Details of Computer 2.

項目	性能
OS	ubuntu 14.04 LTS
プロセッサ	3.70GHz Intel Xeon(R) E5-1660v2
メモリ	31.4 GB

6.3 実験結果

以下にコンピュータ 1 での出力（図 10）とコンピュータ 2 での出力（図 11）を示す．

まずは，コンピュータ 1 の出力の 1 行目に注目すると，networkA にメッセージを送信して，次の 2 行目ではそのメッセージを受信していることがわかる．この時，networkA はコンピュータ 1 上に配置している Actor であるので，そのログはコンピュータ 1 に出力されている．同じコンピュータ内での処理が 1-6 行目にわたって行われている．

しかし，8 行目からの出力に注目するとメッセー

```
1  ..NetworkRef@595cfc3b <-- MethodCommand
       (addNode, ..)
2  Network(networkA) rec MethodCommand(
       addNode, ..)
3  ..NodeRef@34eb30cd <-- MethodCommand(
       getAddress, ..)
4  Node(Router) rec MethodCommand(
       getAddress, ..)
5  ..NodeRef@34eb30cd <-- MethodCommand(
       makeConnection, ..)
6  Node(Router) rec MethodCommand(
       makeConnection, ..)
7  (omission)
8  ..NetworkRef@7854cbbe <-- MethodCommand
       (addNode, ..)
9  ..NodeRef@7552bf24 <-- MethodCommand(
       addRoute, ..)
10 ..NetworkRef@7854cbbe <-- MethodCommand
       (addNode, ..)
11 ..NodeRef@20bce0d5 <-- MethodCommand(
       addRoute, ..)
12 ..NodeRef@20bce0d5 <-- MethodCommand(
       setApplication, ..)
13 ..NodeRef@6321c65e <-- MethodCommand(
       setApplication, ..)
14 (omission)
```

図 10 コンピュータ 1 での出力 (一部省略).
Fig.10: Output of the computer 1 (Omitted).

ジの送信しか行われていないように見える．この送り先はコンピュータ 2 上に配置されている networkB になっており，実際にコンピュータ 2 上の出力の 1 行目には，そのログが出力されている．同様にコンピュータ 1 の出力の 9 行目のメッセージも，コンピュータ 2 上での出力の 8 行目で受信が確認できる．

また，コンピュータ 2 上の出力の 12 行目では「`setApplication`」というメッセージを受信した後，次の行で「`setApplication`」が実行されたことが確認できる．

6.4 考察

以上の実験の結果から，メッセージパッシングによりメソッドを呼び出すコマンドの送受信が行

```
1  Network(networkB) rec MethodCommand(
       addNode, ..)
2  ..NodeRef@36aae0 <-- MethodCommand(
       makeConnection, ..)
3  Network(networkB) rec MethodCommand(
       addNode, ..)
4  ..NodeRef@3e3e5507 <-- MethodCommand(
       getAddress, ..)
5  Node(PC2) rec MethodCommand(getAddress,
        ..)
6  ..NodeRef@3e3e5507 <-- MethodCommand(
       makeConnection, ..)
7  Node(PC2) rec MethodCommand(
       makeConnection, ..)
8  Node(PC2) rec MethodCommand(addRoute,
       ..)
9  Network(networkB) rec MethodCommand(
       addNode, ..)
10 ..NodeRef@4ca97517 <-- MethodCommand(
       getAddress,List())
11 (omission)
12 Node(Web Server) rec MethodCommand(
       setApplication, ..)
13 call setApplication:Node(Web Server),
       malwaresimulator.extend.
       WebServerAppRef@3b265746
14 ..WebServerAppRef@3b265746 <--
       MethodCommand(setNode, ..)
15 application(Web Server, node=null) rec
       MethodCommand(setNode,WrappedArray(
       malwaresimulator.extend.
       NodeRef@46e1a74d))
16 Node(PC2) rec MethodCommand(
       setApplication, ..)
17 call setApplication:Node(PC2),
       malwaresimulator.extend.
       WebBrowserAppRef@63e64dfe
18 (omission)
```

図 11 コンピュータ 2 での出力 (一部省略).
Fig.11: Output of the computer 2 (Omitted).

われている事を確認し，今回作成したコントローラによって Actor が分散環境に適用されていることを確認することができた．

また，Method Missing によるメッセージの送信，その受け取ったコマンドから Reflection が実行され，実際にメソッドが実行されていることも確認できた．

7. まとめ

本研究では脅威トレースで大規模なネットワークでのシミュレーションを行う際の実行時間を短縮させることを目的に，分散環境への適用を行った．分散化には Akka というフレームワークを使用し，メッセージパッシングによるメソッド呼び出しや，スケジューリングなどの実装を行い，現状のような手続き的なメソッド呼び出しによるメッセージ送信や，メッセージの動的処理も併せて行った．

最終的に一部問題もあったが，脅威トレースを分散環境に適用することができた．Actor の依存関係が解消できない部分や，分散環境間のメッセージパッシングがうまくいかない部分もあり，メッセージパッシングの処理の設計を各 Actor の依存関係から見直す必要もある．

しかし，本研究の Akka による分散環境への適用やメッセージパッシングによるメソッド呼び出しといった設計は，今後様々なプログラムを分散環境へ適用する際の基盤として期待できる．

謝辞 本研究は独立行政法人情報処理推進機構の支援および，JSPS 科研費 24500043 の助成を受けたものです．

参考文献

[1] 独立行政法人情報処理推薦機構 (2014) **「高度標的型攻撃」対策に向けたシステム設計ガイド** http://www.ipa.go.jp/files/000042039.pdf,pp.47-95 (参照 2014-11-26)

[2] 独立行政法人情報処理推薦機構 (2011) **標的型攻撃／新しいタイプの攻撃の実態と対策** https://www.ipa.go.jp/files/000024542.pdf,pp.9-21 (参照 2014-11-25)

[3] 独立行政法人情報処理推薦機構 (2014) **前掲「高度標的型攻撃」対策に向けたシステム設計ガイド** pp.39-42

[4] Kato,M.,Matsunami,T.,Kanaoka,A.,Koide,H.and Okamoto,E. **Tracing Attacks on Advanced Persistent Threat in Networkd Systems** In,Proc. SafeConfig 2012 5th Symposium on Configuration Analytics and Automation (October 2012)

[5] Typesafe Inc. **Akka** http://akka.io/ (参照 2014-11-26)
[6] Derek Wyatt.**Akka Concurrency** artima,pp.1-9 (April 2013)