情報処理学会論文誌 Vol.58 No.3 754–761 (Mar. 2017)

# 動的エージェント負荷分散機構の開発

村田 悠也[1,a)] 山本 学[2] 寺野 隆雄[1]

受付日 2016年5月31日, 採録日 2016年12月1日

**概要**：本稿では，マルチエージェントシステムによる大規模データ集計時の性能劣化問題を取り扱う．マルチエージェントシステムは，エージェントプログラミングモデルにより抽象化された処理とデータセットを持つエージェントを多数配置したシステムである．エージェントプログラミングモデルにより設計されたエージェントは，データレコードとメッセージハンドラ，処理ロジックの 3 つの機能を持ち，メッセージパッシングに非同期かつ並列的な動作を行う．しかしながら，環境上に多数のエージェントを配置することから，特定のエージェントに負荷が偏ることでシステム全体の性能が劣化する．エージェントの負荷の偏りは，エージェントへ送られるメッセージが均等に分散されず，特定のエージェントに集中することで発生する．また，負荷はシステム実行中に入力されるデータやエージェント間でのリソース配分により絶えず変化するため事前予測が難しい．そこで，本研究ではシステム実行中にメッセージが集中するエージェントを増やし，負荷を分散させる動的なエージェント負荷分散手法を提案する．この手法により，各エージェントが自身の負荷を測定し，動的に負荷分散することが可能となる．提案手法を毎秒 50 万件のデータを集計するデータ集計システムに適用したところ，高負荷時にエージェントが増え，動的に負荷分散している様子が確認できた．また，本手法を適用したシステムと適用していないシステムで比較すると，データ偏り時に 1.6 倍の性能改善効果が得られた．

**キーワード**：エージェントプログラミングモデル，マルチエージェントシステム，ストリーム処理，分散システム

# Development of the Dynamically Agent Load Balancer for a Multi Agent System

Yuya Murata[1,a)] Gaku Yamamoto[2] Takao Terano[1]

Received: May 31, 2016, Accepted: December 1, 2016

***Abstract:*** This paper discusses a large scale data aggregation problem implemented multi-agent system techniques. The multi-agent system is an abstract programming model with processes and data, To implement such a system, we must use so many software agents assigned to multiprocessors. Each agent, which has functions of data records, message handlers, and processing logics, works message-passing with asynchronous and parallel manners. The system usually shows poor performances, because the processing resources tend to only small number of specific agents, Furthermore, such processes are dynamically according to the environmental conditions, thus, it is difficult to correctly estimate the performance. To cope with the problem, this paper proposes a novel performance balancing method, which dynamically increases and/or decreases the number of agents against the changes of required computational resources. The proposed method enables us to balance the performance loads to automatically measure the dynamic performance by agents themselves. We apply the proposed method to a data aggregation application system, which process five hundred thousand per second. The experimental results show the 1.6 times processing improvement compared with a conventional method.

***Keywords:*** agent programming model, multi agent system, stream processing, distributed system

## 1. はじめに

大規模データの集計処理を行うクラスタリングコンピュータフレームワークに MapReduce [1] を利用した Hadoop や Dryad [2] がある．これらのフレームワークは，企業や Web，

[1] 東京工業大学大学院
Tokyo Institute of Techonology, Yokohama, Kanagawa 226–8502, Japan
[2] 日本アイ・ビー・エム株式会社東京ソフトウェア開発研究所
IBM Japan, Tokyo Software Development Laboratory, Chuo, Tokyo 103–8510, Japan
[a)] murata@trn.dis.titech.ac.jp

EC データを蓄積した大規模データストアに対して高速な集計・分析処理を実行する．その一方で，スマートフォンアプリケーションや IoT システムのように，データが継続的に発生しかつ発生頻度が高いシステムには適用できない．

たとえば，スマートフォンの位置情報共有アプリケーションはセンサが稼働している間，つねにデータが発生する．このアプリケーションは，サービスの拡張や利用者の増加により，加速度的に発生データが増える [3]．もし，アプリケーションの利用者数が 1 万人いるとき，発生するデータ数はユーザ数 × ユーザ数の 1 億件のデータが毎秒発生することになる．さらに，ユーザが 1 人増えるごとに発生するデータは毎秒 2 万件増えるため，データ蓄積による処理はできない．そのため，データを蓄積させずに処理するリアルタイム性と高いデータアクセス性能が必要とされる．従来，このようなデータはストリームデータと呼ばれストリーム処理 [4] により高速に処理されてきたが，ストリーム処理は集計処理用のデータストアがないため適用が難しい．

近年，リアルタイムでのデータ集計処理を実現するフレームワークとして Resilient Distributed Datasets（RDDs）モデル [5] を用いた Spark [6] やエージェントプログラミングモデル（APM）を用いた AgentFramework [7], [8] が開発されている．また，APM の類似モデルにアクタモデルを用いた Orleans [9] があるが利用用途が異なるため議論しない．

RDDs は，抽象化された分散コレクションをメモリ上で操作することによりストリームデータをリアルタイムに集計処理する．RDDs を用いた Spark は，データの更新処理に Hadoop と同様のデータベースを利用しているため，システム性能がデータベースのアクセス性能に依存する．

AgentFramework は，エージェントと呼ばれる特殊なソフトウェアエンティティを生成し，非同期かつ並列的なデータ処理を実現するフレームワークである．このエージェントは，非同期性や反応性といった特性を持つため並列かつリアルタイムな処理が実行可能である．また，エージェントは自身が管理するデータへのアクセス性能が高く，Hadoop や Spark で用いられるデータベースと比較し高速である [7]．さらに，MapReduce の Map や Filter，Reducer といった複数の機能をエージェント上で実現可能である．これにより，大規模データ処理フレームワークで行われる集計処理をエージェントによりリアルタイムに実行可能である．また，環境上に多数のエージェントを分散させることで並列分散処理が可能である．ただし，エージェントはユーザや地域など集計するデータに対して関連付けられているため，エージェントが複数存在すると特定のエージェントへ通信が偏る可能性がある．たとえば，都道府県ごとにデータを集計するエージェントを設計した場合，人口の多い東京都と関連付けられたエージェントは通信が集中する．先行研究では，特定エージェントへの通信の集中については示唆しているものの，その解決方法については示していない [8]．

本稿では，リアルタイムでのデータ集計を行うデータ処理基盤の開発を目指し，エージェントを多数配置した環境での特定エージェントへ通信が集中した場合の性能劣化問題の検証と，その解決方法を提案する．2 章で APM の概要を説明し，3 章で簡易なセンサアプリケーションを例にエージェントでの実装を行い，実行時の問題を分析する．4 章で分析した問題の解決方法を提案し，5 章で提案手法適用による性能改善効果を示す．

## 2. エージェントプログラミングモデル

この章ではエージェントプログラミングモデル（Agent Programming Model，APM）の定義を行い，大規模分散システムへの適用方法を示す．適用例としてスマートフォンアプリケーションから発生するログデータを分析するセンサログマイニングシステムに APM を適用する．

### 2.1 エージェントの定義

APM はデータセットとそれにアクセスする処理セットを抽象化し，メッセージパッシングにより処理を行うプログラミングモデルである．APM により生成されるソフトウェアエンティティは，複数の機能が記述でき，反応性や非同期性といった特性を持つことからエージェントと呼ばれる [7]．なお，自律性を有しないため厳密な意味でのエージェントとは異なる．APM におけるエージェントでは処理や管理するデータを 3 つの機能により抽象化する（**図 1**）．開発者が 3 つの機能を持ったエージェントを実装することにより，システム利用者はエージェント内の処理やデータを知らなくても，エージェントにメッセージを送ることで欲しい結果が得られる．

a). メッセージハンドラ
受け取ったメッセージのメッセージタイプから対応する処理ロジックを呼び出し，メッセージのデータを渡す．メッセージハンドラは，データの整合性を保つため 1 度に 1 つのメッセージを実行する．

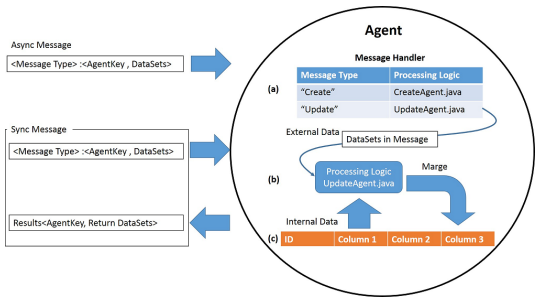


図 1 エージェントプログラミングモデル
**Fig. 1** Agent programming model.

b). 処理ロジック
メッセージハンドラに結び付けられた処理で，メッセージに対して対応する処理が結び付けられている．処理ロジックでは，エージェントが保持するデータの呼び出しや，更新などデータに対する処理や他エージェントやシステム内のリソースへアクセスするためのメッセージを送信できる．

c). データレコード
エージェントが保持するデータ形式．レコードキーによりデータの参照が可能．レコードはレコード自身を内包（レコードセット）することがあり，レコードキーで内包されたレコードセットを呼び出す．

### 2.2 APM でのマルチエージェントシステム

複数のエージェントが，並列的または協調的に動作するシステムをマルチエージェントシステム（Multi Agent System，MAS）と呼ぶ．APM に基づいて設計されたエージェントは，分散環境上に多数配置することで高いスケーラビリティ性能を持ち，高速な並列分散処理を実現する [7]．さらに，エージェント間で相互通信を行うことで協調的な動作を実現する [10]．

### 2.3 センサログマイニングシステム

センサログマイニングシステムの概要について述べる．センサログマイニングシステムは，センサから発生したログデータをユーザ年齢ごとに分類し，集計するシステムである（**図 2**）．このように，入力されるストリームデータが分類可能でかつ独立に集計できるとき APM を適用できる [11]．センサログマイニングシステムの利用例として，リアルタイムでのターゲティング広告が考えられる．たとえば 10 代のアプリケーション利用者に対して広告を出す場合，このシステムを用いることで 10 代が一番接続している瞬間に広告を出すことができる．

リアルタイム処理を必要としないログマイニングシステムは，MapReduce により実装可能である．MapReduce での動作をプロセスフローとして記述したものが図 2 となる．ただし，センサログを用いるセンサログマイニングシステムでは，利用例のようにリアルタイム性を有するため MapReduce による実現はできない．

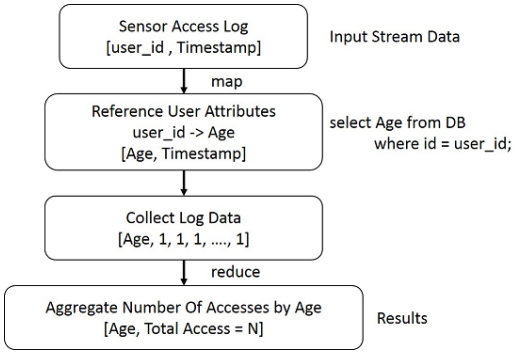


**図 2** センサログマイニングシステムでのログデータ処理
**Fig. 2** Example: sensor log mining.

## 3. システムの構成と実装

2 章で説明したセンサログマイニングシステムを MAS により実装する．MAS を実行するテスト環境は，ユーザデータを生成するテストデータ生成器とエージェントを配置するエージェント実行環境の 2 つに分けられている（**図 3**）．なおテスト環境は，AMD-FX8100 8Core 2.7 GHz の CPU，12 GB メモリ，CentOS 5.4 64 bit を搭載した計算機上で構築し，MAS は AgentFramework を用いて Java（Ver.IBM JDK1.8）で実装した．

### 3.1 MAS の実装

AgentFramework では，エージェントが管理するデータやメッセージハンドラは XML 形式のエージェント定義ファイルにより定義される．定義されたエージェントは計算機上に複数配置され，送られてきたメッセージに従い動作する．しかしながら，定義されたとおりに実装されたエージェントはメッセージ処理が非効率的である．そのため，エージェント実行環境上にウィンドウ制御とメッセージキューの 2 つの機能を組み込みメッセージ処理を効率化する．ウィンドウ制御は，ストリーム処理で行われているフロー制御の一種で，発生したデータをウィンドウと呼ばれる単位にまとめ，複数データを一括で処理する．メッセージキューは，送られてきたメッセージを一時的に保持することで，エージェントの待ち時間を減らし処理を効率化する．エージェントにメッセージキューを組み込むことで，メッセージのオーバヘッドによる性能劣化を改善することが確認されている [10]．この 2 つの機能を組み込むとメッセージ処理は次の順番で実行される．

1). ウィンドウ制御
送られてくるデータをウィンドウと呼ばれるメモリ領域に蓄積する．ウィンドウはエージェントごとに用意され，メモリがいっぱいになると，ウィンドウをメッセージに変換しエージェントに送信する．

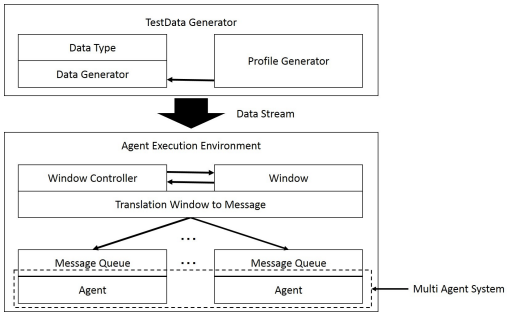


**図 3** MAS の実験システム構成
**Fig. 3** Experimental system for MAS.

2). メッセージキュー
エージェント宛に送られてきたメッセージを保持する．エージェントは，メッセージキューからメッセージを取り出し処理を実行する．指定されたキュー長以上のメッセージが受信されると，メッセージ送信元であるウィンドウ制御部に時間を置いて再送するようにリクエストを送る．

3). エージェント
メッセージキューから受信したメッセージを処理する．エージェントは，メッセージハンドラにより受信メッセージに対応する処理ロジックを呼び出しメッセージ処理を行う．メッセージが送られてこない間エージェントは待機する．

### 3.2 テストデータ生成器の実装

MAS の処理性能を測定するためのテストデータを生成する．テストデータは，毎秒数万～数十万件のストリームデータを発生させる．テスト条件に合わせて発生させるデータ形式を変更する．本稿では，センサログマイニングシステムのテストを行うため，ユーザ ID とアクセス時間からなるテストデータを生成する．

### 3.3 センサログマイニングシステムの実装

2 章のセンサログマイニングシステムを実装する．ここでは，システムの核となるログ分析を行うエージェントの実装のみ説明する．すでにシステムの要求仕様からエージェント設計を行う方法が存在する [10], [11]．この設計法によりエージェントの定義ファイルを記述する（**図 4**）．

```
<?xml version="1.0"?>
<agentdef package="rda" version="rda1.0">
  <entities>
  <!--Aggregate Agent Entity define-->
    <entity type="aggregate agent">
      <attribute name="AgentID" type="string" primarykey="true" maxlength="32"/>
      <attribute name="Data" type="long" />
      <attribute name="ConnectionCount" type="long" />
      <attribute name="Log" type="log" />
    </entity>

    <entity type="log">
      <attribute name="AgentID" type="string"
                 primarykey="true" relationto="AgentID" maxlength="32"/>
      <attribute name="AccessID" type="string" primarykey="true" maxlength="16"/>
      <attribute name="CurrentTime" type="long"/>
      <attribute name="LastAccessTime" type="timestamp" />
    </entity>
  </entities>

  <messages>
    <!--message define-->
    <!--Agent Messages -->
    <message type="initAgent" class="rda.agent.message.InitMessage"/>
      <message type="readAgent" />
      <message type="updateAgent" class="rda.agent.message.UpdateMessage"/>
      <message type="readLogAgent"/>
      <message type="deleteAgent" />
  </messages>

  <agents>
    <!--Message handler define-->
    <agent type="aggregate agent">
      <handler message="initAgent" class="rda.agent.handler.InitHandler"/>
      <handler message="readAgent" class="rda.agent.handler.ReadHandler"/>
      <handler message="updateAgent" class="rda.agent.handler.UpdateHandler"/>
      <handler message="readLogAgent" class="rda.agent.handler.ReadLogHandler"/>
      <handler message="deleteAgent" class="rda.agent.handler.DeleteHandler"/>
   </agent>
  </agents>
</agentdef>
```

**図 4** エージェント定義ファイル
**Fig. 4** Agent definition.

entities はエージェントが保持するデータの定義で，messages，agents はメッセージハンドラの定義となる．データの定義ではエージェントが保持するデータレコードが記述される．AgentID は各エージェントの集約条件となるため固有な値が設定される．このとき AgentID の値の決め方は，ユーザ ID から AgentID を算出できるようにする．たとえば，年齢が集約条件となるときユーザ ID の下 2 桁を年齢とすることで，年齢を ID とする集計エージェントにメッセージを送信できる．メッセージハンドラの定義では，エージェントの処理ロジックを記述した Java プログラムとメッセージを関連付けることで，メッセージ受信時にエージェントの処理を呼び出し実行する．エージェントのデータ集計時の処理ロジックは図 2 のプロセスフローとなる．

### 3.4 センサログマイニングシステムの性能評価

実装した MAS を 100 秒間稼働したときの性能評価を行った．なお，エージェント数はユーザを年代別に集計するため 10 体とし，各実験条件で 100 回試行しその平均を比較した．データ発生方法は次の 2 つを比較した（**図 5**, **図 6**）．(a) ユーザ属性（年齢）に偏りがないデータを毎秒 50 万件発生させる．(b) ユーザ属性（年齢）に偏りがあるデータを毎秒 50 万件発生させる．

実験の結果，100 秒間の総トランザクション数は，データ

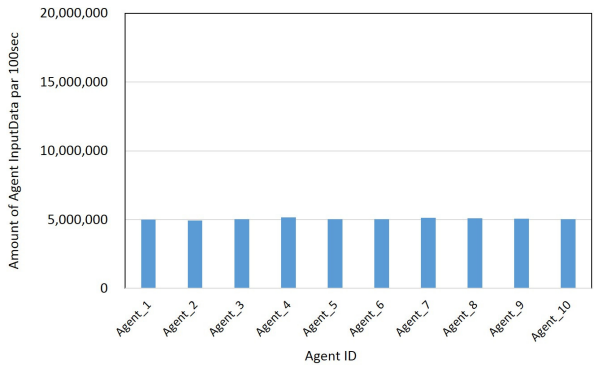


**図 5** (a) エージェントへの入力データ数（偏りなし）[100 sec])
**Fig. 5** (a) Amount of agent inputdata without bias par 100 sec.

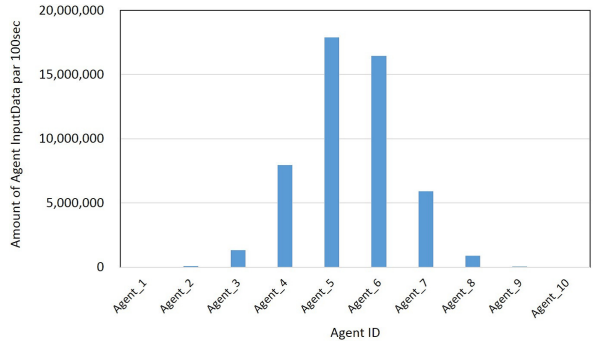


**図 6** (b) エージェントへの入力データ数（偏りあり）[100 sec])
**Fig. 6** (b) Amount of agent inputdata with bias par 100 sec.

の偏りのない (a) は 5,000 万件で偏りのある (b) は，3,000 万件であった．これは，データの偏りによりおよそ 40%の性能劣化が起きたということである．このことから，MAS への入力データの偏りがシステム全体のデータ処理性能に影響したことが分かる．

### 3.5 MAS の問題

エージェントへ送信されるメッセージの偏りが MAS の性能に影響する．分散処理では特定ノードへのアクセス集中により性能が劣化することが知られている．アクセス集中の問題は，集中するアクセスを均等に分散するロードバランサを設置することで解決できる．ロードバランサは，特定ノードへ通信が集中するとき，その通信を空いている他のノードに変更することで負荷を分散させる．これは，どのノードにアクセスしても同様の結果が返ってくるという前提条件が存在しているからである．

MAS の場合は特定エージェントにメッセージが集中することで性能が劣化する．しかしながら，分散処理の解決方法を MAS に適用する場合，ロードバランサの前提条件が問題となる．MAS ではエージェントがメッセージを受信する際，各エージェントの集約条件に従ってメッセージを受け取るかどうかを決める．もし，エージェントに集中するメッセージを負荷の低い別のエージェントに送った場合，集約条件が異なるためメッセージを受信することはできない．つまり，20 代のユーザデータを集約するという集約条件を持ったエージェントは 20 代のユーザデータが含まれたメッセージしか受け取れないということである．集約条件を無視してメッセージを分散させた場合，分散されたメッセージがどのエージェントで処理されたか知る方法がないため，すべてのエージェント間でデータの同期をとらなければならない．

## 4. 動的エージェント負荷分散

多数のメッセージが特定のエージェントに集中し，処理待ちが発生している状態をエージェント負荷と呼ぶ．エージェントはデータ整合性を保つため 1 度に 1 メッセージしか処理ができない．MAS はエージェントを増やして並列度を上げることで性能を上げるが，特定のエージェントにメッセージが集中するとエージェント内で逐次的に処理されるため性能が劣化する．この問題は，メッセージが集中するエージェントを複製しメッセージを分散することで解決できる．複製されたエージェントは，同一の集約条件を持つため分散されたメッセージを受信できる．これにより，特定エージェントへ集中したメッセージを分散し並列にメッセージ処理を実行することで処理性能の劣化を抑える [12], [13].

ただし，メッセージ集中の問題はシステム実行中に発生する問題である．そこで，先に述べたエージェントの複製による負荷分散をシステム実行中に実施する必要がある．動的エージェント負荷分散は，負荷が集中するエージェントをシステム実行中に動的に負荷分散する手法である．これにより，実行中のメッセージ集中によるボトルネックを，エージェントの負荷分散を行い並列度を上げることで性能を向上させる．動的エージェント負荷分散は 3 つの機能により実現される．この章では 3 つの機能について説明する．

### 4.1 エージェント負荷検出

エージェント宛のメッセージは，各エージェントのメッセージキューに挿入されてからエージェントに送られる．エージェントが忙しいとき，受信メッセージはキューに蓄積され続ける．つまりメッセージの処理待ちが発生している状態である．これを利用しエージェントは自身に備え付けられたメッセージキューから負荷を検出する．負荷の判断には，エージェントが持つメッセージキューのキュー長が設定された長さ以上のときエージェント負荷とする．

### 4.2 エージェントクローンの作成

負荷検出後，エージェントは自身の ID からクローン ID を生成する．その後，生成したクローン ID をもとにオリジナルのエージェントの複製を作成する．ここでは，オリジナルエージェントの複製をエージェントクローンと呼ぶ．エージェントクローンは，オリジナルのエージェントと同じ集約条件を持ち，同様の作成プロセスにより生成される．エージェントクローンは生成完了後に完了メッセージをオリジナルのエージェントに送信する．この段階では，メッセージの宛て先リストに作成したエージェントクローンが登録されていないためメッセージが届くことはない．

### 4.3 メッセージ宛て先リストの更新

メッセージ宛て先リストとは，**表 1** のように，初期に生成されたエージェント ID をキーとし，その ID と生成されたクローン ID のリスト（宛て先リスト）がペアとなったデータである．初期エージェント（オリジナルエージェント）の ID からリストを参照することで，生成したクローンを含めた宛て先リストを取得する．メッセージ宛て先リストはウィンドウ制御部で管理されている．

ユーザが Agent_2 にメッセージを送るとき，AgentID の Agent_2 を利用し，表 1 の宛て先リストを取得する．次にリストから，Agent_2 とそのクローンの Agent_2-1 の 2 つの ID を取り出す（表 1 カッコの中）．最後にユーザは，自身の ID のハッシュ値を計算し，2 つの AgentID のうちどちらか 1 つを取得することでエージェントにメッセージを送ることができる．このようにユーザ ID のハッシュ値を使うことでオリジナルとクローン間で均等にメッセージを分散できる．

エージェントクローンは，生成時ではメッセージ宛て先

表 1 エージェントのメッセージ宛て先リスト
**Table 1** Destination list of messages.

| Original AgentID | Destination List |
|---|---|
| AgentID = Agent_1 | Agent_1 ->(Agent_1) |
| AgentID = Agent_2 | Agent_2 ->(Agent_2, Agent_2-1 (Clone_1)) |
| AgentID = Agent_3 | Agent_3 ->(Agent_3, Agent_3-1 (Clone_1), Agent_3-2 (Clone_2)) |

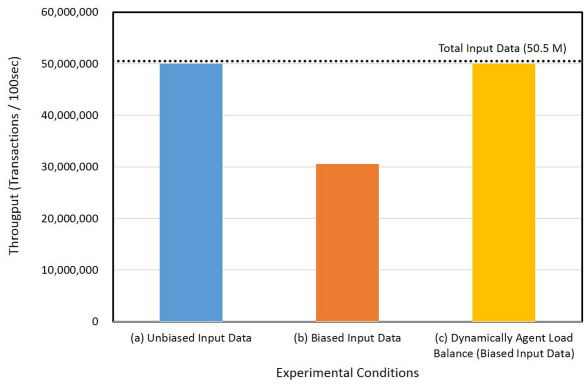


図 7 動的エージェント負荷分散機構のトランザクション比較
**Fig. 7** Evaluation performance of the dynamically agent load balancer.

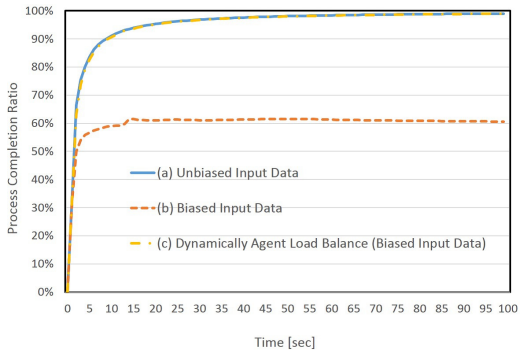


図 8 発生データの処理完了割合の推移
**Fig. 8** Finished transactions of the agents par sec.

リストに ID が存在しないためメッセージを受信することはできない．そこで，メッセージ宛て先リストの更新が必要となる．メッセージ宛て先リストの更新は，クローンを生成したオリジナルエージェントが行う．オリジナルエージェントは，クローンから生成完了のメッセージを受信すると，自身の ID をキーとして表 1 にあるメッセージ宛て先リストを取得する．取得されたメッセージ宛て先リストにクローン ID を追加することで更新完了となる．

## 5. 動的エージェント負荷分散の性能評価

3 章で実装したシステムに動的エージェント負荷分散機構を適用し性能を評価する．実験は 3 章で実装したセンサログマイニングシステムを単一環境上で実現し，次の条件で比較する．なお，実験結果はそれぞれ各 100 回実行しその平均とする．

(a) ユーザ属性（年齢）に偏りがないデータを毎秒 50 万件発生させる．
(b) ユーザ属性（年齢）に偏りがあるデータを毎秒 50 万件発生させる．
(c) 実験 (b) と同じデータ発生条件で動的なエージェント負荷分散機構を適用する．

また，エージェントのキュー長は最大 1,000 としている（エージェントが 1 秒以内に処理できるメッセージ数）．

実験の結果，実験 (a) はエージェントの負荷が均等に分散されているため，発生データの 99%を処理している（**図 7** (a)）．一方，実験 (b) は，特定のエージェントに負荷が集中していることで性能が劣化し，発生データの 60%しか処理できていない（図 7 (b)）．実験 (c) は実験 (b) に動的エージェント負荷分散機構を組み込むことで，処理性能が大幅に劣化せず，実験 (a) と同程度のスループット（−0.001%差）となった（図 7 (c)）．このことから，提案手法を適用することで，特定エージェントへのメッセージ集中による性能劣化が改善できることが確認された．

100 秒間の処理完了割合の時間的変化を確認する．処理完了割合とは，各時間の累積発生データ数のうち，実際にエージェントによって処理された割合であり，エージェントが処理したデータ数を各時間の累積発生データ数で割った値となる．**図 8** の結果から，データの偏りがない実験 (a)，動的エージェント負荷分散機構を組み込んだ実験 (c) は，ほぼ同じ結果となっている．一方．データの偏りがある実験 (b) はシステム稼働から 3 秒後に性能限界に達し，毎秒発生データの 60%しか処理されない．

次に，エージェント負荷分散機構が期待どおりに働いているか確認するため 100 回の実験のうち 1 回を取り出しシステム実行中のエージェント数の推移，負荷検出後のトランザクションの変化を確認する．

動的エージェント負荷分散による環境中のエージェント数の時間的推移について考察する．**図 9** を見ると，オリジナルエージェントから複製されたクローン数の時間変化を見ると，初期段階（5–15 秒）で急激にエージェント数を増やし，その後 80 秒までクローンの増加はない．このことから，動的エージェント負荷分散機構を組み込んだ実験 (c) では，初期にエージェント数を増やしていたことで並列性を損なわれることなく，順調に発生データの処理を完了していったことが予想される．また，図 9 のグラフの帯幅と，図 6 のデータの偏りを比較すると，発生データ数

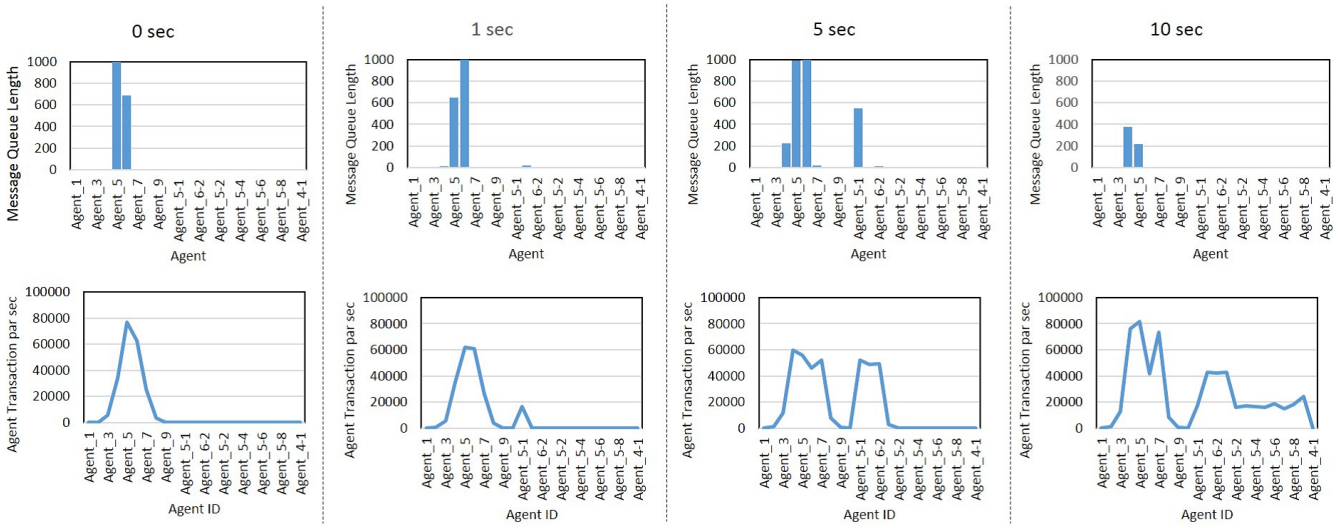


図 10 エージェント負荷（上）とトランザクション数（下）の時間変化
**Fig. 10** Temporal change in agent load and the transactions. Top is the queue length in the message queue. Bottom is the amount of the agent transactions par sec.

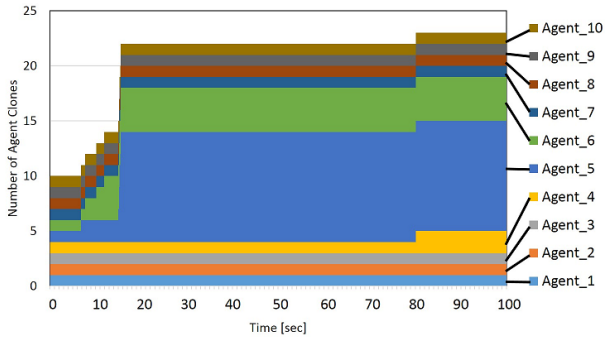


図 9 システム実行中のエージェント数の推移
**Fig. 9** Transition of the number of agents.

の多い Agent_5 と Agent_6 が集中して増えていることが分かる．

負荷検出後のメッセージキューのキュー長とトランザクション数の変化を確認する（図 10）．グラフの一番左が，はじめてエージェント負荷を検出したときを 0 秒とし，1 秒～10 秒までのキュー長（上）とトランザクション（下）の変化を見る．0 秒後に負荷を検出したシステムは動的にエージェントを生成し，作成したエージェントがデータを処理していることが分かる．1 秒後，5 秒後のトランザクション数を見ると 0 秒後のときトランザクションの山が 1 つだったのに対し，1 秒後はトランザクションの山が 2 つになっている．5 秒後では，1 秒後の小さい山が大きくなっていることが確認できる．その後，10 秒の段階でエージェント数をさらに増やし，データ処理を分散して実行している．

## 6. おわりに

APM により構築した MAS は，多数のエージェントが非同期かつ並列に動作することで高速なデータ処理を実現する．本稿では，大規模データの集計処理フレームワークが適用できない例としてセンサログマイニングシステムをあげ，エージェントで実装したときの MAS の性能を評価した．結果，MAS は発生データの偏りにより特定のエージェントにメッセージが集中するとシステム全体の性能が劣化することが確認された．このメッセージ集中の問題は，分散処理システムや Web システムでは通信，アクセスの集中問題として知られている．しかしながら，これらのシステムと同様の解決方法を MAS に適用することは難しい．その理由は，すべてのエージェントは固有の集約条件を持ち，集約条件に合致したメッセージしか受け取れないことにある．そこで，この問題を固有の集約条件を持つエージェントのクローンを作ることで解決した．エージェントクローンは，複製元のエージェントと同じ集約条件を持つため，そのエージェントに送られたメッセージを受け取ることができる．これにより，メッセージが集中するエージェント自身に自分の複製を作成させ，受け取るメッセージを分散させることで負荷を分散させる．一方，センサアプリケーションではセンサの稼働中にシステムを止めることはできない．システムを止めることなく複製による負荷分散を実現したものが，動的なエージェント負荷分散である．この機構を組み込むことで，データの偏りが存在しても性能の劣化を抑えることが可能となった．実験では，この機構を適用することでデータの偏りが存在しても，データの偏りがないときと比較して処理性能が変わらないという結果となった．これにより，設計段階で予測できないメッセージ集中による性能劣化をシステム利用者の負担なく，自動で解消することが可能となった．

今後の課題として，本手法で性能が劣化する可能性について考察し，その対策を行う必要がある．本実験は，100 秒間でのデータ発生パターンが変化しない理想環境下での実

験であった．エージェント負荷分散時のクローンの生成の手順から性能劣化要因は次の 2 つが考えられる．(a) データ発生間隔がエージェント複製時間より短いとき，複製コストがシステム性能に大きく影響してしまう．(b) 偏る集計項目が時間ごとに変化するとき，エージェントの過剰な生成が起きる．本手法適用時には，これらの問題が発生するか検証し，発生する場合その解決が必要とされる．

## 参考文献


[1] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *Comm. ACM*, Vol.51, No.1, pp.107–113 (2008).

[2] Isard, M., Budiu, M., Yu, Y., Birrell, A. and Fetterly, D.: Dryad: Distributed data-parallel programs from sequential building blocks, *ACM SIGOPS Operating Systems Review*, Vol.41, No.3, pp.59–72, ACM (2007).

[3] Ericsson AB: ERICSSON MOBILITY REPORT ON THE PULSE OF THE NETWORKED SOCIETY, Technical Report, Ericsson Mobility Report, available from ⟨http://www.ericsson.com/mobility-report⟩ (2016).

[4] Arasu, A., Babu, S. and Widom, J.: The CQL Continuous Query Language: Semantic Foundations and Query Execution, *The VLDB Journal*, Vol.15, No.2, pp.121–142 (2006).

[5] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S. and Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, *Proc. 9th USENIX Conference on Networked Systems Design and Implementation*, p.2, USENIX Association (2012).

[6] Spark: Spark Documentation, Apache Spark (online), available from ⟨http://spark.apache.org/docs/⟩ (accessed 2016-05-30).

[7] 山本　学：エージェントプログラミングモデルに基づくデータキャッシュの性能評価，合同エージェントワークショップ&シンポジウム 2010（JAWS2010）(2010).

[8] 山本　学：エージェントプログラミングモデルの業務システムへの適用からの考察，コンピュータソフトウェア，Vol.31, No.3, pp.109–119 (2014).

[9] Bernstein, P., Bykov, S., Geller, A., Kliot, G. and Thelin, J.: Orleans: Distributed virtual actors for programmability and scalability, Technical report, MSR Technical Report (MSR-TR-2014-41, 24), available from ⟨http://aka.ms/Ykyqft⟩ (2014).

[10] 村田悠也，山本　学，寺野隆雄：リアルタイムデータ集計アプリケーションにおける効果的なエージェント分散配置，合同エージェントワークショップ&シンポジウム 2014（JAWS2014）(2014).

[11] Yuya, M., Gaku, Y. and Takao, T.: Improving the Performance of Mobile Application Systems through Agent Deployment Strategies in a Distributed Environment, *5th World Congress on Social Simulation* (2014).

[12] Shehory, O., Sycara, K., Chalasani, P. and Jha, S.: Agent cloning: An approach to agent mobility and resource allocation, *IEEE Communications Magazine*, Vol.36, No.7, pp.58, 63–67 (1998).

[13] Ishida, T.: *Parallel, distributed and multiagent production systems*, Vol.878, Springer Heidelberg (1994).



**村田 悠也** （学生会員）

2013 年東京工業大学大学院修士課程修了．同年同大学院博士課程へ進学．電気学会，日本ソフトウェア科学会，人工知能学会，IEEE 各会員．



**山本 学**

1991 年東京工業大学大学院工学研究科制御工学専攻修士課程修了．同年 3 月日本アイ・ビー・エム（株）入社，東京基礎研究所配属．2008 年同社東京ソフトウェア開発研究所へ異動．2008 年京都大学大学院情報学研究科博士課程修了．博士（情報学）．2009 年 4 月～2014 年 3 月東京工業大学大学院総合理工学研究科連携教授．エージェントプログラミングモデル，エージェント基盤技術の開発に従事．第 8 回合同エージェントワークショップ&シンポジウム最優秀論文賞受賞．



**寺野 隆雄** （正会員）

1976 年東京大学工学部計数工学科卒業．78 年同大学大学院工学系研究科修士課程修了．1978～1990 年（財）電力中央研究所，1990～2004 年筑波大学講師・助教授・教授を経て 2004 年より東京工業大学大学院総合理工学研究科教授．1996 年イリノイ大学アーバナシャンピン校，スタンフォード大学客員研究員．工学博士．社会シミュレーション，データマイニング，サービスサイエンス等の研究に従事．