エージェントプログラミングモデルのトレーディングシステムへの適用からの考察

Discussion on applying an agent programming model to a trading system

山本　学 [1]*

[1] 日本アイ・ビー・エム（株）東京ソフトウェア開発研究所
[1] Tokyo Software Development Laboratory, IBM Japan

Abstract: Agent programming models such as distributed multiagent models and mobile agent models have been proposed. We have proposed an agent programming model for business applications since 2000 and applied to auction systems, a financial trading system, and others applications. Through those applications we obtained the knowledge that the programming model is efficient on developing applications. Especially, it is very efficient for scale out applications. In this paper, we discuss about the efficiency by referring a trading system developed on top of the model.

1　はじめに

エージェントプログラミングモデルとは、プログラム開発のためのモデルである。ここでのエージェントは、アプリケーションの観点からわかりやすい役割を持ち、自律的・反応的に処理を行うソフトウェア・エンティティである。このようなプログラミングモデルとしてのエージェントには幾つかのタイプがある。分散エージェントは、分散システムを幾つかのサブシステムの集合体と考え、それぞれのサブシステムをエージェントで表すモデルである。サブシステム間の連携は、メッセージングで行われる。移動エージェントは、データとロジックを持ち、メッセージを受けて処理を行い、必要に応じて実行環境を移動するモデルである。

エージェントプログラミングモデルの一つとして、トランザクションを実行する業務システムへの適用を想定したものがある。これは、筆者が2000年以降に開発し、実システムに適用してきたものである [1][5]。このプログラミングモデルでは、エージェントは業務的にわかりやすい役割を持ち、メッセージを受けて自分のデータにアクセスしながら処理を実行するものである。必要に応じて、他のエージェントにメッセージを送ることもある。筆者は、この考え方に高速処理のためのインメモリデータストア技術とスケールアウトを融合させ、非常に高速なトランザクション処理システムを実現するためのフレームワーク「IBM Agent Framework for Datagrid XTP edition」を開発し、金融商品トレーディングシステムに適用した。インターネットの利用者が利用する金融商品トレーディングシステムでは、数万人を超える利用者の口座を管理し、レート更新にあわせて必要な処理を実施しなければならない。例えば、レート更新に合わせて口座の余力を計算し、余力がない口座に対しては損切り処理である。これを行うには、各口座の余力を計算して判定する必要があり、これを全口座に対して実施しなければならない。レート更新は数百ミリ秒間隔で行われるため、この間に数百万レコードの参照を行う必要がある。開発されたシステムはこの性能要件を満たすことができた。本稿では、このような非常に高い処理速度を要する金融系トレーディングシステムにおいてエージェントプログラミングモデルがどのように貢献したかに関して述べる。

2　エージェントプログラミングモデル

2.1　概要

このプログラミングモデルでは、例えば利用者のような業務的に意味のあるものに対してエージェントを定義する。エージェントは実行環境で生成され、明示的に削除されるまでそのサーバに存在し続ける。一般的には、多数のエージェントがサーバで生成される。エージェントはメッセージを受けて処理を行う反応的エー

*連絡先： 日本アイ・ビー・エム（株）
東京ソフトウェア開発研究所
〒 135-8511 東京都江東区豊洲 5 丁目 6 番 52 号
NBF 豊洲キャナルフロンティアビル
E-mail: yamamoto@jp.ibm.com

ジェントである。エージェント実行環境には、エージェントが利用するデータが存在する。これは、特定のエージェントが持つデータではなく、複数のエージェントが利用するものである。本稿では「共有データ」と呼ぶ。図 1 にエージェントプログラミングモデルの概要を示す。

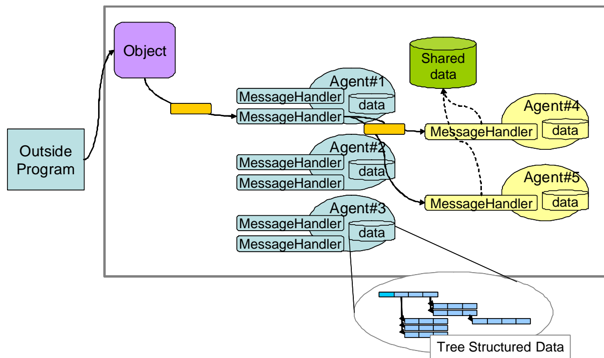

図 1: エージェントプログラミングモデル.

エージェントが活動するサーバは複数存在し、クラスタ化される。エージェントは生成されたサーバで活動し続ける。エージェントのサーバへの配置は、エージェントが持つ識別子のハッシュ値とサーバ数から得られるサーバの番号で決定される。

2.2 エージェント

エージェントは下記の特徴を持つソフトウェア・エンティティである。

1. 自分自身のデータを持つ
2. 他エージェントのデータに直接アクセスしない
3. 唯一の識別子を持つ
4. 役割を持ち、そのために活動する
5. メッセージを受けて、処理を開始し、短い時間で処理を完了する
6. 必要に応じて共有データを利用する
7. 他のエージェントに非同期のメッセージを送る
8. 同時並行して複数のメッセージ処理を行わない

エージェントは、言語処理系のオブジェクトである必要はない [2]。

2.3 エージェントのデータ

エージェントは自分のデータを保持する。メッセージ処理を行うメソッド (メッセージハンドラメソッド) が呼び出された時には、自分のデータにだけアクセスする。エージェントのデータの種類は数十から 100 種類近くある。ひとつのトランザクションでは、一部のデータ項目の更新・追加・削除が行われる。エージェントのデータはひとつのデータ項目をルートとするツリーで構成される。図 1 に示すように、ひとつのエージェントは必ずツリーのルートとなるデータ項目を持ち、他のデータ項目は、ルートの下位につながるデータ項目となる。

エージェントのデータは永続化される必要がある。また、後述するように、エージェントのデータはメモリ上に保持されるため、冗長化が必要である。そのために、エージェントのデータ項目はリレーショナルデータベース（RDB）のレコードとして表現され、RDB にも保持される。

2.4 メッセージモデル

ここで言うメッセージとは、実行環境内でエージェントを呼び出すためのものであり、外部プログラムからのリモート呼び出しのものではない。このメッセージモデルには図 2 の六種類がある。

同期型ユニキャスト エージェントの識別子を指定し、一つのエージェントにのみメッセージを送り、結果を待つ

非同期型ユニキャスト エージェントの識別子を指定し、一つのエージェントにのみメッセージを送るが、結果は待たない

同期型マルチキャスト 複数のエージェント識別子を指定して、一つのメッセージを複数のエージェントに送り、それぞれのエージェントの処理結果を待つ。処理結果は各エージェントの戻り値のコレクションである。

非同期型マルチキャスト 複数のエージェント識別子を指定して、一つのメッセージを複数のエージェントに送る。結果は待たない。

同期型ブロードキャスト 同じサーバプロセスにいる全エージェントに一つのメッセージを送り、それぞれのエージェントの処理結果を待つ。処理結果は各エージェントの戻り値のコレクションである。

非同期型ブロードキャスト 同じサーバプロセスにいる全エージェントに一つのメッセージを送る。結果は待たない。

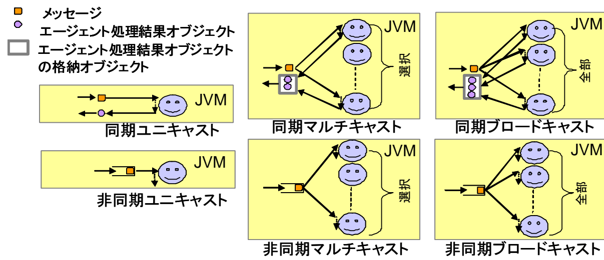

図 2: メッセージングモデル.

2.5　メッセージ処理

あるエージェントにメッセージが送られると、エージェント実行環境は、宛先のエージェントのメッセージハンドラメソッドを呼び出す。一つのエージェント実行環境のプロセスでは、限られた数のスレッドを順番に使う。そのため、メッセージは一旦エージェント実行環境のキューに入れられる。ひとつのエージェントは同時並行して複数のメッセージ処理を行わない。

メッセージが同期型の場合、そのメソッドの戻り値が処理結果として、呼び出し元に返される。同期型マルチキャストと同期型ブロードキャストの場合、例外を示すオブジェクトが戻り値を格納するコレクションに格納される。メッセージが非同期の場合、メッセージハンドラメソッドの戻り値は無視される。

エージェントが他のエージェントにメッセージを送る場合は、非同期型のみの利用である。

2.6　共有データ管理

エージェントプログラミングモデルでは、エージェントが保持するデータは排他的である。しかしながら、ほとんどのアプリケーションで、複数のエージェントが共有するデータが存在する。エージェントプログラミングモデルを適用するアプリケーションは、高い性能を要求するものが多い。共有データへのアクセスを他のサーバプロセスへのリモートアクセスとすると、それにかかる通信のオーバーヘッドが大きくなり、性能劣化となる。そのため、共有データは、全ての実行環境に同じデータを配置する方法が良い。しかしながら、この方法はデータ一貫性の問題引き起こす。これに関しては後述する。

3　高速処理技術の融合

3.1　概要

トランザクション処理システムの高速化のボトルネックとなる点は、データアクセスである。従来までの金融系トレーディングシステムでは、図 3 に示すように、すべてのデータをデータベースに格納し、トランザクション処理サーバからそこにアクセスするというものであった。これにはデータベースサーバのボトルネックとトランザクション処理サーバからデータベースサーバへの通信の 2 つが性能上の課題点であった。これに対し、第二世代のアーキテクチャとして、図 4 に示すようなデータをメインメモリに置くデータグリッドサーバ群をトランザクション処理サーバとデータベースサーバの間に配置するというものが登場した。これは、データベースサーバのボトルネックの解消とデータアクセスの高速化で従来よりも高速なシステムを実現できた。しかしながら、インターネットで多くの利用者が利用する金融系トレーディングシステムでは、より高い性能が要求され、十分な性能を実現するには至っていない。これは主にトランザクション処理サーバとデータグリッドサーバ間の通信コストのためである。

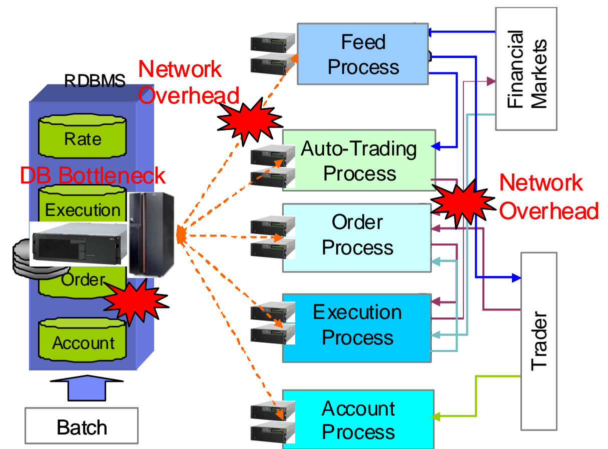

図 3: DB 中心のアーキテクチャ.

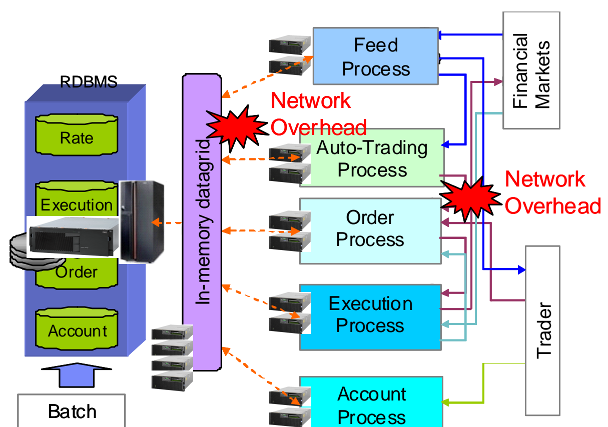

図 4: 第二世代のアーキテクチャ.

これらを解決する方法として下記の方法が考えられる。

1. 業務ロジックと同じプロセスのメモリ上にデータを配置する
2. スケールアウトを適用し、複数計算機で処理を行う

3.2　インメモリデータ処理

トランザクション処理システムの処理速度の典型的な課題は、データアクセスの性能である。特に、通信を介したデータアクセスは通信コストが大きく、本稿で対象とするような数百ミリ秒で数百万レコードにアクセスする処理を実現することは困難である。これを実現するには、トランザクションのロジックとデータを同じプロセスに配置し、通信なしでアクセスを可能にさせる方法が有効である。

このような手法は、既存のインメモリデータグリッド技術でも可能である。例えば、多くのキーバリューストア製品でも実現できる。しかしながら、例えば、100 ミリ秒で 10 万口座の損切り処理を行うことを仮定すると、既存のキーバリューストア製品での実現は容易ではない。1 口座 100 レコード保持すると考えると、1000 万レコードをわずか 100 ミリ秒で参照しなければならない。既存のキーバリューストア製品では、キャッシュが同一プロセスに存在していても 1 オブジェクトの取得に数マイクロ秒程度かかる。仮に 5 マイクロ秒としても単純計算で 500CPU コア必要になる。16CPU コアの計算機で 32 台である。本稿で紹介するエージェントプログラミングモデルに基づくインメモリデータ処理機構では、1 オブジェクトの取得にわずか 50〜200 ナノ秒程度である。200 ナノ秒と考えると、1000 万レコードを 100 ミリ秒で参照するために必要な CPU コア数は 20 である。16CPU コアの計算機で 2 台である。実際はビジネスロジックやトランザクション処理のオーバーヘッドもあるため、2 台ということはないが、それでも 5 台程度で可能であろう。

つまり、鍵はインメモリデータアクセスコストを非常に小さくすることであり、これは「エージェント」の概念に基づくデータ管理手法により実現できている。詳細は [3][4] に譲るが、簡単に述べると、「口座エージェント」がその口座の全レコードをツリー構造で保持し、ロックを各レコードではなくエージェントに対して行うこと、ツリーを辿りレコードを取得するコストを非常に小さく実装したことが鍵である。

これらの仕掛けは、エージェントプログラミングモデルを提供する実行環境の内部に埋め込むことができ、プログラム開発者がこれらの機構を認識する必要はない。

3.3　スケールアウト

スケールアウトの効果を最大限に発揮させるには、処理の過程でサーバ計算機間の通信をできるだけ排除させることである。そのためには、ひとつのトランザクションでアクセスするレコード群をひとつのプロセスに配置することである。これはエージェントプログラミングモデルでは、エージェントの活動空間を複数の区画に分割し、あるエージェントは必ずひとつの区画に配置することである。エージェントプログラミングモデルでは、トランザクションは基本的にはそのエージェントが持つデータにしかアクセスしない。そのため、エージェントのキー値でハッシュ分割することで容易にスケールアウトを実現できる。

エージェント同士のコミュニケーションが存在する場合はこれは難しいが、金融系トレーディングシステムではコミュニケーションは不要である。唯一考慮すべき点は、共有データである。例えば、レート情報は全エージェントが参照するデータである。これに関しては後述する。

3.4　新アーキテクチャ

図 5 にこの考え方に基づいたアーキテクチャを示す。この図では、「DataGrid」にある各サーバ計算機上にエージェントが分散配置される。各サーバ計算機には、エージェントのデータと業務ロジックが置かれる。業務ロジックはエージェントのメッセージハンドラメソッドとして実装される。エージェントのデータは、RDB にも保持されるが、RDB はエージェントのサーバプロセス毎に設けられる。これにより、ひとつの RDB がボトルネックとなることはない。また、「IBM Agent Framework for DataGrid XTP edition」では、この RDB にインメモリテーブル機能と通常のディスクベーステーブル機構を持つ RDB 製品を利用している。このため、要件に合わせてテーブル毎にいずれかのテーブルを利用することができる。

エージェントのデータは図中にある「Centralized DBMS」に非同期にコピーされる。これはエージェントのデータ全体に対しての処理、例えばバッチ処理など、を行うためのものであり、全エージェントのデータが保持される。

4　金融商品トレーディングシステム

筆者が適用した金融商品トレーディングはインターネットで個人投資家が利用するシステムで、利用者の注文を受けて金融機関に注文を出す。利用者は PC やスマートフォンから自分の口座にアクセスする。金融

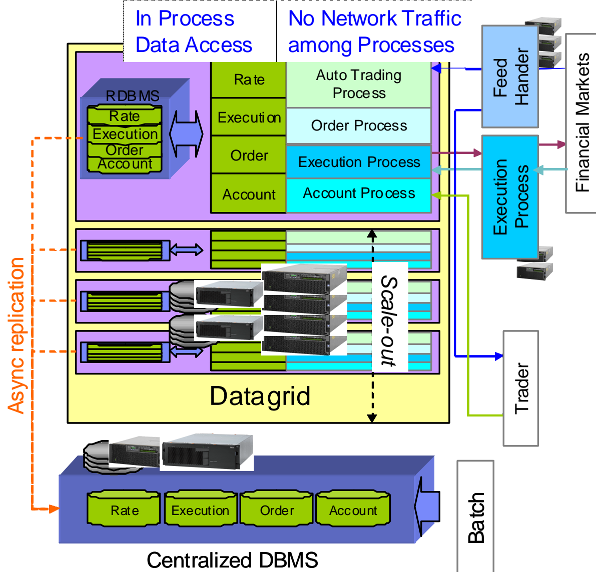

図 5: 新しいアーキテクチャ.

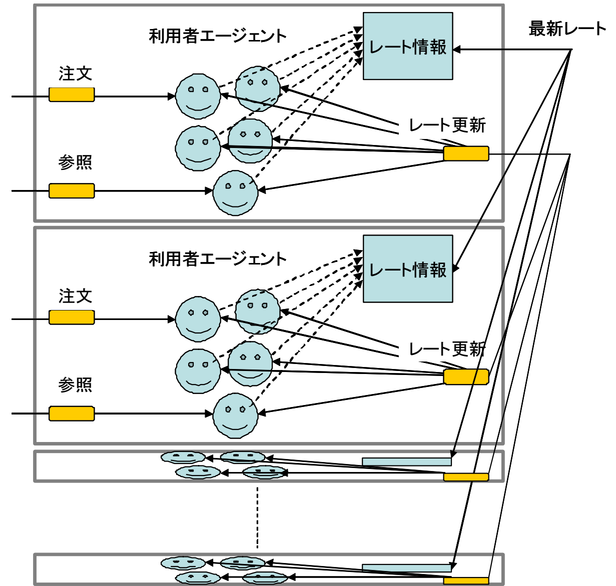

図 6: 金融系トレーディングシステム.

市場からの商品売買レートを毎秒数回の頻度で受信し、各口座の残高や注文状態などを参照して、損金が口座残高を超えていないかを確認し取引停止状態にする損切り確認処理や見込まれる利益がある閾値を超えた場合に自動的に注文を行う自動売買処理を行う。注文情報は外部の金融市場に送られ、結果はその金融市場から受けとり、口座情報が更新される。

図 6 にシステム概要を示す。このシステムでは、エージェントは利用者を代表するものであり、口座情報を保持する。口座情報は、名前などの属性情報、口座残高、注文情報、注文履歴情報など多岐に渡る。口座数も数十万となるため、数十万エージェントが生成される。商品売買レートは全区画にて共有データとして、商品番号をキーとしてレート情報を取得できる形式で保持される。商品売買レート更新はメッセージとして全利用者エージェントに配信される。これを受けたエージェントは自分の口座情報と共有データである商品売買レートにアクセスしながら、損切り確認処理や自動売買処理を行う。利用者からのアクセス要求もメッセージとしてエージェントに渡される。

このシステムでは処理速度が非常に重要となる。システムの処理の遅延が利益損失に直結する。例えば、金融市場からのレート受信後の損切り確認処理が次のレート配信までに終わらないと、レートが変化し、損害が大きくなることが起きる。したがって、毎秒数回更新されるレート更新に追従できなければならないが、実際に取引を行う口座が数万あり、一口座が数十から 100 近いレコードを持つため、数百万レコードの参照を 100 ミリ秒程度で完了させなければならない。そのため、数台の計算機を使用し、十以上の区画に分割した構成となっている。

5 考察

金融商品トレーディングシステムでは、あらゆる処理の高速化が要求される。スケールアウト環境でのインメモリデータストアでは、ひとつのサーバプロセス内で処理を完結させることが鍵となる。本質的には、アプリケーションが持つスケールアウトのための性質が鍵であるが、アプリケーションの要件から何ら指針やガイドなくこの性質を引き出して設計することは簡単なことではない。エージェントプログラミングモデルは、インメモリデータストアとスケールアウトを利用したアプリケーションの基本形を提示しており、そのために設計者が適切な設計を行うことができる。金融商品トレーディングシステムの開発では、インメモリデータストア技術とスケールアウトを初めて使う設計者と開発者による構築であった。非常に高い性能をもつ複雑なアプリケーションを、高速処理技術の経験がない開発者により開発できたことである。これはエージェントプログラミングモデルによるところが大きいと言える。

エージェントプログラミングモデルの設計の鍵は何をエージェントとするかである。業務から容易に抽出されるものをエージェントとして定義する方法でよい。金融商品トレーディングシステムでは利用者エージェントが定義された。これは業務上の明確な役割がある。

エージェントの処理がひとつの区画で閉じるための設計を行うに当たり、次の点の考慮が必要である。

1. トランザクションがエージェントのメッセージハンドラメソッドの単位で閉じること
2. エージェント間コミュニケーションの扱い
3. エージェントの配置
4. スケールアウトによる共有データの扱い

金融商品トレーディングシステムでは 1、2 が問題になることはなかった。そのため、3 はエージェントの識別子でハッシュ分割させるという簡単な方法で問題なく対応できた。

4 に関しては、データの一貫性をどう考えるかである。これは「区画内での一貫性が維持されれば良い」という方針で設計することである。この判断には、業務的に問題がないかどうかの判断が必要となる。これに該当するものは、各種のマスターデータと呼ばれるデータ項目の ID から名称を引くためのテーブルや金融商品レート情報である。マスターデータは、ほとんど更新されることはなく、また、更新時においても全サーバプロセスで完全に同期している必要はないため、問題にはならない。金融商品レート情報に関してはもう少し慎重な判断が必要になるが、結論から言えば、業務的に考えると、必ずしも金融商品レート情報を全区画で同期更新する必要はない。このデータは更新頻度は高いが、トランザクションの投入タイミングと金融商品レート情報の更新は到着順に処理されるものであり、金融商品レート情報を全区画で同期更新させても、トランザクションの処理タイミングはシステム依存であるため、金融商品レート情報の更新タイミングが各区画で大きくずれないのであれば、区画毎に更新しても業務的に問題にはならない。このように、共有データの更新に関しては、業務上の影響を考えながら、区画毎の一貫性維持で問題ないように設計しなければならない。

6　まとめ

エージェントプログラミングモデルの利点は、わかりやすいモデルでアプリケーションの構造化を行える点である。特にスケールアウトやインメモリデータストア技術を必要とする高速なアプリケーションで効果が大きい。一般的には、スケールアウトやインメモリデータストア技術を適切に利用して、高速処理を実現することは、設計も含めて簡単なことではない。エージェントプログラミングモデルは、その基本形を提示しており、考慮点を明確に示すことで、高速でスケーラブルなアプリケーションの開発に貢献している。一方、このプログラミングモデルの適用にあたっては、アプリケーションをうまくモデル化できるかどうかを慎重に考える必要がある。この要は、トランザクションがエージェントのメッセージハンドラメソッドの単位で閉じること、エージェント間コミュニケーションの扱い、エージェントの配置、スケールアウトによる共有データの扱いである。この議論には、業務上の制限を緩められるかどうかの議論も必要となる。

参考文献

[1] G. Yamamoto, and H. Tai : Performance Evaluation of An Agent Server Capable of Hosting Large Numbers of Agents, *Proceedings of the fifth international conference on Autonomous agents*, pp。 363?369 (2001)

[2] 山本学：高い機能拡張性を持つ業務システム用のエージェントフレームワークの実現, 人工知能学会論文誌 26(1), 127-135, 2011 (2011)

[3] 山本学: エージェントプログラミングモデルの高速トランザクション処理システムに対する効果の考察, 合同エージェントワークショップ＆シンポジウム 2008 (2008)

[4] 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 2011 (2011)

[5] T. Koyanagi, Y. Kobayashi, S. Miyagi, and G. Yamamoto: Agent server for a location-aware personalized notification service, *Massively Multi-Agent Systems I*, Lecture Notes in Computer Science (2005)