日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集

Meta Engine を用いた Federated Linda の実験

赤嶺 一樹　河野 真治

本研究室では、分散型タプルスペースの実験用に Federated Linda を提案し、実装してきた。従来の Federated Linda は各ノードの間に配置された Protocol Engine によって互いに連携するが、プロセスが異なるため無駄な通信が存在した。そこで Federated Linda と同一プロセス上で動作する Meta Engine を提案し、実装してきた。本研究では、クラスター上で Meta Engine を用いた実験用トポロジーを構築し、MetaEngine の使用例を示す。

1　はじめに

twitter をはじめとする大人数参加型 Web サービスや MMORPG などの大人数参加型リアルタイムネットワークゲームが、これまで以上に大規模なものとなっていくためには、分散ネットワークプログラムの発展が不可欠である。

しかし、分散ネットワークプログラムにおけるスケーラビリティーの確保は難しい。ここで言うスケーラビリティーとは、サービスの大きさが増えたときにサーバーなどのリソースを追加することのみでサービスの質をリニアに維持できることを指す。

すなわち理想的なモデルは、複数のサーバーを接続することで負荷を分散し、クライアントの数に従ってサービスが自然にスケールするものでなくてはならない。

例えば、現在の分散技術を助けている Key-Value Store などでは、すべてのデータのレプリケーションを複数台用意するという手法がとられている。しかしながら、大人数が参加するサービスにおいて、全員が個人のとあるデータを持っている必要性はない。自分と関連のある人物の情報さえ取得できるようになっていればよいのである。つまり、全データのレプリケーションを用意する必要はなく、サーバーがクライアントの必要な情報とは何かを把握し、情報を取捨しながら伝搬していく必要がある。

本研究室では、分散プログラムモデルとして、タプルスペースを制御する Linda を採用してきた。さらに、複数台の Linda サーバーを接続したモデルである分散型タプルスペース Federated Linda を提案し、実装してきた。本研究では、Linda の API の見直しと、 Federated Linda を用いたアプリケーションの実装を行い、現在のシステムの問題点を洗い出すことにする。

2　ゲームの例題

2.1　水族館ゲーム

本研究では、ネットワークゲームを例題として用いることにした。そのゲームは、複数のクライアントのディスプレイを並べて使用する。各プレイヤーは 1 匹ずつ魚のオブジェクトが与えられ、それを自由に操作することが出来る。また、魚は画面の端まで移動すると、自分の画面上からは消え、隣のプレイヤーの画面の端から魚が出てくる。(図 1)

Experiment of Federated Linda with Meta Engine

Kazuki Akamine, 琉球大学理工学研究科情報工学専攻, Information Engineering Course, Faculty of Engineering Graduate School of Engineering and Science, University of the Ryukyus.

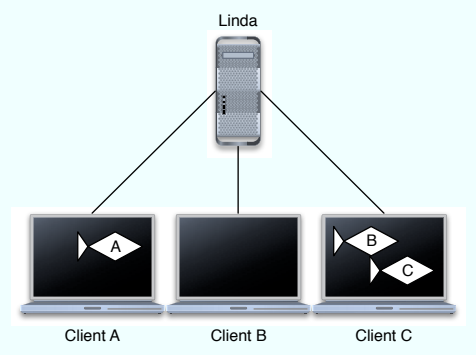

図 1　A,B,C の順に接続すると左から順に画面が接続される。 B の魚は C の画面まで移動している。

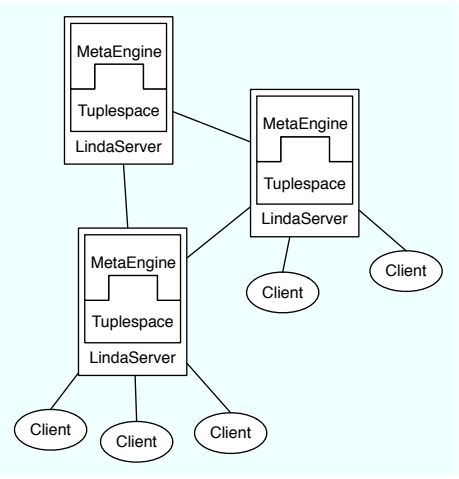

図 2　Federate Linda の接続モデル。組み込まれた Meta Engine がタプルスペースを操作し、外部のサーバーへデータを伝搬する

3　Federated Linda

3.1　Linda とは

Linda は、タプルスペースという ID で区画されたデータストアに、以下の API (表 1) を用いてデータを出し入れすることによって、外部との通信を行う分散プログラミングモデルである。

表 1　Linda API

in(id)	タプル空間から取り出す。 タプル空間にタプルは残らない。
rd(id)	タプル空間から取り出す。 タプル空間にタプルが残る。
out(id,data)	タプル空間にタプルを入れる。

3.2　Federated Linda とは

Federated Linda は Linda サーバーを複数台、相互に接続することによって、分散プログラミングを実現する。各サーバーは、接続した Linda サーバー内のタプルスペースへデータの in/out を行うことによって、データを伝搬する。

3.3　Meta Engine とは

Federated Linda は、サーバー間に設置された、Protocol Engine と呼ばれるプログラムによって、タプルスペースの操作や、他サーバーへのタプルの伝搬などを行っており、タプルスペースとは別のプロセスとして、サーバー上に存在していた。しかし、別のプロセスであるため、タプルスペースへのアクセスには同一サーバー上であっても、ソケット通信を用いていた。

そこで、本研究室では、 Meta Engine と呼ばれるプログラムを提案し実装してきた。 Meta Engine は、 タプルスペースと同一プロセス上に組み込まれた Protocol Engine である。(図 2) すなわち、タプルスペースと同じメモリ空間にあるため、ソケット通信を用いることなく直接 Linda の API を使用して、タプルスペースにアクセスすることが出来る。

4　Linda API の見直し

4.1　update() API の追加

現状の Linda API では、タプル内のデータを更新するためには in() を実行してデータを取り出して削除したあとに、out() を実行してデータを書き込む必要があった。(図 3) 今回考えている水族館の例題でも、操作があったときにはタプルスペースの座標情報を最新のデータに更新する必要がある。

そこで、今回、新しく update() API を追加するこ

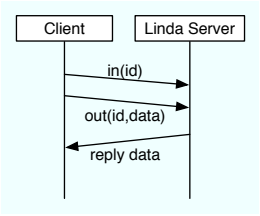

図 3　in/out を用いたデータの更新

とにした。update() を実行すると、現在存在するタプルは削除され、新しいデータで上書きされる。(図 4)

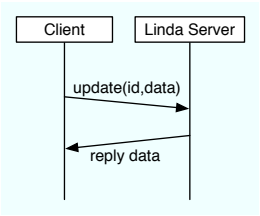

図 4　update を用いたデータの更新

update() の引数は、 out() と同じく update(id,data) のように、 id と data を渡す。

4.2　wait_rd() の問題点

Linda には、rd API の他に、wait_rd という API が存在する。wait_rd はただちに reply を返す rd とは違って、タプルの情報が更新されるまで、サーバー側で reply が保留される。この API を用いることで、必要な座標情報が更新されたときのみに通信を行うことができる。しかし、 wait_rd を受け付けたときのタプルデータは返されないため、最新の情報を得ることが出来ないというバグが、ゲームのクライアントを作成するときに発生した。

これを回避するために、 wait_rd の結果が次のループで返されなかったときは、rd を用いて現在のタプルの情報を取得するようにした。しかしこれは、2 重で rd の API を使っており、場合によっては無駄な通信を伴なう。また、クライアントの受信側で、 rd と wait_rd のどちらが最新のデータか判別する無駄な条件分岐が発生するため、場当たり的な対応でしかない。

これを解決するためには、タプルのバージョンをサーバー側で記録するようにするか、タイムスタンプを用いてタプルを管理するといった方法が挙げられる。

Linda にこのような API を実装する予定はないが、Linda を踏まえて次に本研究室で設計する分散データベースでは、参考にして設計しなおす予定である。

5　Meta Engine を用いたサーバーの設計

5.1　ツリー型トポロジーを用いた負荷分散

今回の例題には、ツリー型トポロジー Federated Linda を用いることにした。(図 5)

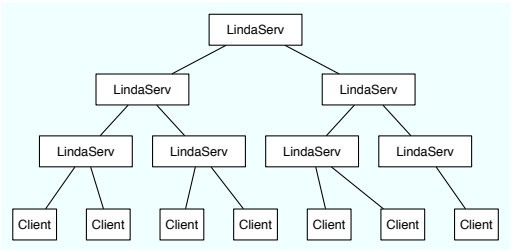

図 5　ツリー型トポロジーで接続された Federated Linda。ツリーの末端にクライアントを接続する。

ツリー型トポロジーの各ノードは、親と 2 つの子への接続を持つ。また、末端のノードはクライアントによる接続も受け付ける。

クライアントから座標情報を受け取った末端のノードは、その座標情報を他のクライアントが必要としているかを判断する。もしも必要ならば、その Linda サーバーのみで通信は完結しているため、 Federated Linda 上でデータの伝搬は行われない。

もし必要としない場合は、親のノードにタプルデータの伝搬を行う。そのとき親のノードは、その子から送られてきた座標情報を元に、対になっている子がデータを必要としているかを判断する。もし必要ならば、対になっている子に対してタプルデータを伝搬する。そうでなければ、さらにその親に対してタプルデータの伝搬を行う。

このように、座標データを必要としているマシンの

みに座標データをコピーすることで、負荷分散を実現するモデルである。 このモデルを用いない場合、つまり Linda サーバーを単体で用いる場合は、各クライアントは、すべての接続しているクライアントの所持しているオブジェクトの座標情報を常に監視する必要があり、画面に表示されない (すなわち、必要のない) データも操作がされるたびに通信する必要があった。

5.2 Meta Engine の問題点

具体的に Meta Engine とは、サーバーが他サーバーとどのような接続を行い、どのタプルを監視し、どこに伝搬するかというのを記述するものである。

Federated Linda は Java を用いて実装しており、その上に実装されている Meta Engine ももちろん Java で記述している。処理は関数でシンプルに記述できるとよいが、 Java では関数のみを記述することはできないので、 interface を実装したクラスを作成することで処理を記述する必要がある。

また、タプルの伝搬を行うために Federated Linda サーバーの接続一つごとに専用のタプルを決め、そこを in によって監視するのだが、その記述を Callback 関数のように記述すると、コードの難読化が進んでしまう。

さらに、他サーバーと接続する処理を記述し、そのサーバーからのデータを受信するタプルと、相手への伝搬先のタプルを対応付けする処理を毎回記述する必要があるため、新しいトポロジーを構築しようとしたときにコードの記述量が増えてしまう。これを解決するためには、 Meta Engine とタプルスペースの間にそれらの伝搬のシステムを管理する機構を準備する必要がある。

6 評価

6.1 update API の検証

update() API を 4.1 で設計した通りに実装した。

また、 in/out を用いた場合と、 update を用いた場合で時間を比較することにした。

それぞれの API で N 回の更新を行う処理時間を 2 台のクラスタを用いて計測した。(図 6)

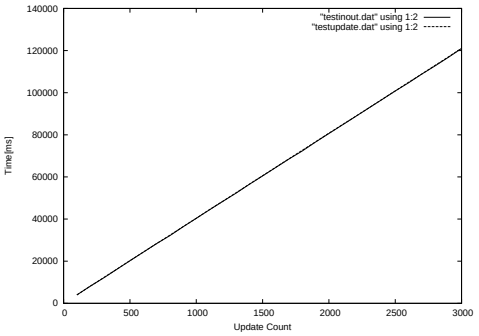

図 6 in/out による更新と update による更新時間の比較

この結果より、 update を用いてタプルを更新した場合と、 in/out を用いてタプルを更新した場合とで、特に差を見ることはできなかった。

しかし、クライアントのコードを書くときに、in/out を 2 つ書くよりも update を 1 つだけ書く方が、書きやすく、読みやすいという点は利点として挙げられる。

6.2 TCP No Delay の検証

TCP No Delay (短いパケットの送信を遅らせる動作の制御フラグ) の影響が大きいとの指摘があったので、 Federated Linda でリング型トポロジーを構築し、タプルの転送速度の差を検証した。今回、使用したクラスタの台数は 46 台である。

リングに流したデータには、 4096 Bytes のデータを用い、 100 周したうちの平均時間より、 1 周にかかる時間を算出した。(図 7)

この事より、TCP No Delay によって、パフォーマンスに大きな差を得られることはないことがわかった。

7 まとめと今後の課題

今回の研究では、 Federated Linda を用いたゲームの設計を行った。この設計や Linda をもちいたクライアントを実装している段階で、 4.2 や 5.2 のように改善できそうな点を複数見つけることができた。

まだ、ツリー型トポロジーの Federated Linda を用いたバージョンをきちんと実装できていないのでこれらの実装を通して、さらに Federated Linda の問

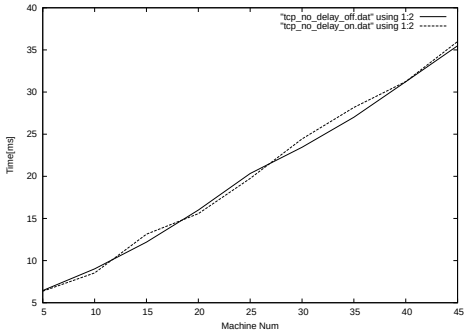

図 7　TCP No Delay の検証

題点を見つけていくことができる。

また、複数の Federated Linda サーバーをデバッグするのは難しいため、それらに加えて、スケーラブルなデバッグ方法なども考えていかなくてはならない。

参 考 文 献

[1] 渕田良彦: 分散プログラミングモデル Federated Linda と分散デバッグ開発

[2] 赤嶺悠太, 小野雅俊, 河野真治: 連邦型 Linda による分散アルゴリズムをデバッグするためのメタプロトコル