動的ルーティングによりタプル配信を行なう分散タプルスペース Federated Linda

安村 恭一 †, 河野 真治 ††
Kyoichi YASUMURA, Shinji KONO
琉球大学理工学研究科 †,
琉球大学工学部情報工学科 ††
Graduate School of Engineering and Science, University of the Ryukyus†,
Information Engineering, University of the Ryukyus.††

Federated Linda とは、複数のタプルスペースを結合し、その間をタプルを伝播させ、分散システムを構築する分散タプルスペースである。“in”, “read”, “out” などの単純な命令群や分散アルゴリズムを内臓したプログラミングモデルなどにより、玩具的に分散システムを構築することを目指している。本稿では、Federated Linda について説明し、動的ルーティングを用いてのタプル配信について述べる。

1 自然に分散プログラミングが書けるようなプログラミングモデル

分散プログラミングは比較的ネットワーク的に遠いコンピュータを取り扱うプログラミングであり、実際に書くことも習得することも難しい。単純に通信するだけでは、一ヶ所に通信が集中してしまうことが起きやすい。

逐次プログラムでも、もちろん、ある程度難しい。しかし、その難しさは、例えば、Perl や BASIC などのインタプリタ、あるいは、While 文 や For 文などの構造型、オブジェクト指向言語などにより、少なからず緩和される。アセンブラや C などのポインタを直接扱うような言語よりも、これらの高度な言語の方がはるかにプログラミングしやすく、習得も早い。それは、制御構造文やスタック、あるいはオブジェクトなどが自然な逐次プログラミングを書くようにしているからと考えられる。

分散プログラミングに対して、そのような自然に分散プログラミングが書けるようなプログラミングモデルを提供できないだろうか? 本論文では、Linda などのタプルスペースの拡張を用いて、自然に分散プログラミングが書けるようなプログラミングモデルを提案する。

2 分散プログラムのどこが難しいのか

単純に離れたホスト間で通信して動作するだけならば、プログラム自体は難しくない。ただ、逐次プログラムに通信プリミティブを導入すれば良いだけである。分散プログラムが難しいのは、それをスケーラブルにすること、つまり、規模を大きくしたときにもちゃんと動作するようにすることが難しいからである。

実際、Internet 上でも、2 点間で通信する、あるいは、比較的少数のアクセスを想定した集中サーバ構成が通常であり、きちんとスケールする分散アプリケーションは珍しい。例えば、DNS (Domain Name System) は、そのようなスケールする分散アプリケーションの一つである。DNS は、数十万人を対象とした強力な少数の集中サーバなどと異なり、はるかに大きな規模 (数億人) を対象のサービスを、より非力な、より多数のホストによって実現している。一方で、IRC (Internet Relay Chat) などは、単純な木構造を持つメッセージ放送システムであるが、サーバ管理などを怠ると全体のパフォーマンスが極端に下がってしまう。Net News なども配送効率は非常に高いが、運営コストは大きい。

このような状況では、Internet 規模で動作する分散プログラムは難しく、集中サーバ構成を取る方が容易だとも言える。しかし、DNS のような成功した例もあり、分散アプリケーションをちゃんと作ることができれば、全体的なコストとパフォーマンス、そして安全性もますと考えられる。

分散プログラムが Internet 規模で動作するためには、以下のような機能を実装する必要がある。

1. ホスト数が増えてもアクセスの集中がないようにする手法

	id に対応する tuple
in(id,tuple)	タプル空間に入れる
out(id)	タプル空間から取り出す

表 1: Linda API

2. サービスの増加に対応した動的な接続変更

3. 通信の切断への対処

3 タプル空間による分散プログラム Linda

本論文でははタプルスペースを用いた手法を使うので、Linda[1] についての考察を行う。Linda は、タプルという id で番号づけられたデータの塊を以下の API (表 1) で、共有されたタプル空間に出し入れすることにより分散プログラムを行う。(図 1)

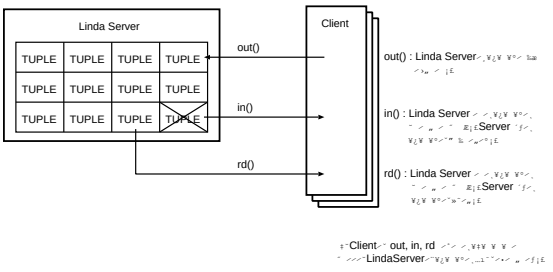

図 1: LindaServer

Linda の利点としては、まず、通信モデルが理解しやすいことがあげられる。一つは基本オペレーションが少ないからである。したがって実装も容易である。また、タプル空間は接続の切断にも強く ((3) に対応)、空間に接続するクライアントの構成の変更 ((2) に対応) も容易である。しかし、Linda が実用的なプログラムで用いられないのは、タプル空間を単一のサーバとして実装すると、サーバにアクセスが集中してしまうからである。タプル空間をスケールするように作るのは非常に難しい。キャッシュや複製を利用したものがいくつか提案されたが、実際の通信が Linda のモデルとは別なものになってしまうのは望ましくない。つまり、Linda は、素直に書くと集中型になってしまう分散プログラミングモデルである。

自然な分散プログラミングを目指すためには以下のようなことが要請されると考えている。

- 実際の通信のモデルが理解しやすい
- Basic のように会話的に開発できる
- WSDL[2] などを記述する面倒がない
- 一方で、WDSL などに自然に対応する
- Linda のように基本オペレーションが少ない
- 分散環境での運用が容易
- インターネット環境で自動的に相互接続する

自然に書いて、スケールする分散プログラムになることが目標である。そのためには、高度な分散アルゴリズムを使う必要がある。それは、アプリケーションのプログラミングからは、関数呼び出しの呼出先のデータ構造やアルゴリズムが隠蔽されるという意味で、隠蔽される必要がある。つまり、分散アルゴリズム自体を内蔵するようなプログラミングモデルが望ましい。

4 今までのツール

通信ライブラリを用意すれば分散プログラムは可能となる。しかし、それは、チューリングマシンで、すべての計算が可能というような意味でしかない。

多数のエージェントが巨大な共有データに自由にアクセスすると言うブラックボードモデルは、AI で良く使われている。これは、タプル空間のモデルと良く似たものであり、同じ欠点を持っている。我々は、Linda の API を非同期にすること [1] により、ビデオゲームのようなリアルタイプアプリケーションに対しても、Linda が有効であることを示して来た。分散共有メモリは、タプル空間よりも均一なメモリアクセスを提供するが、その裏では複雑なメモリのコンシステシー制御が動いていて、その通信のスケーラビリティを制御することは難しい。

分散オブジェクトは、通信の主体としてデータの集りを用いる。様々な提案された Object Request Broker(ORB)[4] は通信データの規格化や、通信の設定には有効である。しかし、任意のオブジェクトが任意のオブジェクトに自由に通信できるので、実際に、どうやって分散プログラムを作るのかと言うことに関しては助けにならない。Sun の Jini [5] なども分散アルゴリズムそのものには無関心である。JXTA [6] はサーバの配置しか解決してくれない。

このような「プロ向け」のツールではなく、より教育的、あるいは、分散アルゴリズム、分散アプリ

ケーションを玩具のように作っていけるツールがないのだろうか?

5 分散プログラムの要素

分散プログラムには三つの要素がある。

```
Distributed Program =
    Protocol Engine
    Local access to protocol
    Physical position , Link configuration
```

これらの三つを分離してサポートできれば、プログラム開発時、あるいはテスト時に、その一部に集中することができる。プロトコルのプログラムと、アプリケーションのプログラムの分離が重要であると考えられる。

Local access は直接に通信にアクセスする API である。これは、本質的に非同期である。終了を待つ read/write や、同じく終了を待つ Linda の in/out では機能的に足りない。例えば、データを待っているプロセスを途中で止めることができなくなってしまう。これを [マルチスレッド+同期機構] あるいは、[割込みや例外処理] で対処することもできるが、プログラミングモデルは極めて複雑になってしまう。一方で、コンピュータの CPU のハードウェアモデルは単純で [状態遷移機械] つまり、入力に対して反応して状態を変えるだけである。Local access API は、より単純なものが望ましい。

分散プログラムは、物理的に離れたホストと、それをつなぐ物理的、論理的なネットワーク接続を含む。これの管理は手動では極めて繁雑である。PC クラスタのような状況では MPI シェルのような形で管理するのが簡単だが、Internet 環境ではそうはいかない。同じ分散アルゴリズムでも配置によって異なる振舞をする。また、同じ物理構成、論理構成でも異なるアプリケーションが走ることがある。同じ物理構成でも、論理的には異なるネットワークを構成することもある。例えば、スパニングツリーなどはそのようなものである。

6 Federated Linda の提案

Federated Linda は、簡単で、複数のタプル空間を相互に接続することにより分散プログラムを実現する。一つのタプル空間には少数の接続があることが期待されており、多数のタプル空間の接続により分散アプリケーションを実現する。smtp/nntp デーモン

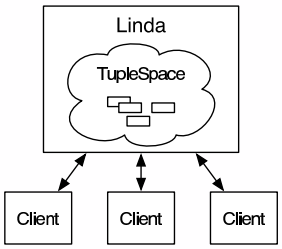

図 2: type 1

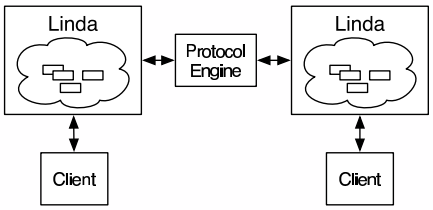

図 3: type 2

が行単位でプロトコルを作るのと同じ感じで、タプル空間への in/out でプロトコルを作ることになる。

Federated Linda には 3 つ段階がある。

type 1 original Linda

type 2 複数の Linda をエージェントが橋渡しする

type 3 Tuple 空間自体が直接接続される

type 1 は通常の Linda プログラムである。(図 2 参照)。type 2 は、複数の Linda に同時にアクセスすることができる。type 2 は type 1 の実装から容易に作ることができる。

type 2 のタプル空間 (Linda) は、Engine と呼ばれるエージェントで接続される。Engine から Linda へのアクセスは通常の Linda API で行われる。(図 3 参照)。エージェントは状態を持たないように作ることが望ましい。そのようにすれば、状態は Linda 上に維持される。Linda との接続が切れても、状態が Linda に維持されていれば、状態を持たない Engine を接続することにより自動的に再接続される。現在の実装では、Engine は Perl 上の非同期タプル通信であり、シングルスレッドで動作する。

Engine は、タプル空間のタプルを見張り、タプルの状態の変化により、接続されたタプル空間へアクセスする。必要なら計算処理を行う。どのような処理を行うかは、Engine のプログラムによって決まる。

FederatedLinda->open($hostname, $port)	タプルスペースへ接続する
FederatedLinda->sync()	タプルの送受信を行なう
Linda->in($tuple_id)	タプルスペース上の指定された ID のタプルの受け取り要求をする
Linda->out($tuple_id)	タプルスペース上の指定された ID のタプルの書き込み要求をする
Reply->reply()	受け取ったタプルのデータを取りだす

表 2: Federated Linda API

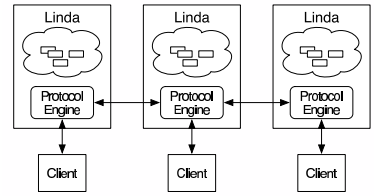

図 4: type 3

type 2 では、これらは前もって決まっており変更することは想定していない。type 3 では、なんらかのプログラムのロード機構が必要となると考えられる。

type 3 では、Engine は Linda と一体化して抽象化される。(図 4 参照)。Linda 間は、Inter-Linda プロトコルで接続される。その API は、現在は未定であるが、タプル空間にアクセスする時に、そのプロトコルを指定するような手法を用いる予定である。この段階で、ネットワーク資源の管理、例えば、複数ユーザや複数のアプリケーションの分離を実装する。これらの詳細は、type 2 でのプログラミング経験に基づいて決定する予定である。

Linda と Engine の接続は、ネットワークで接続される。type 2 では、それらは手動で設定される。これらの設定の自動化は、やはり type 3 で行われる。接続のトポロジーは例えば図 5, 図 6 のような Tree や Mesh が考えられる。

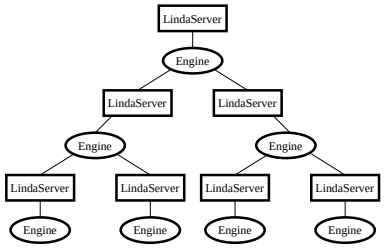

図 5: Tree 型トポロジ

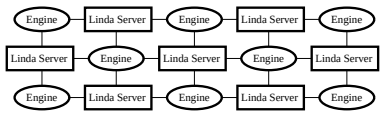

図 6: Mesh 型トポロジ

7 Federated Linda の実装と例題

Linda サーバの実装では、タプルの ID は、16bit の整数値を用いている。タプルのデータは任意の文字列である。Linda server は C で記述されており、クライアントは C または Perl で記述する。Perl の API は表 2 の通りである。Linda サーバ自体はメモリ上のキューである。

今回、type2 において分散システムを構築するため、基本的なプロトコルである、ルーティングプロトコルを実装した。ルーティングは RIP などのディスタンスベクタ型を参考にし、各タプルスペースごとに経路テーブル情報を用意した。タプルスペース自体はルーティングを行わず、代わりにルーティングなどを行うエージェントを perl で作成した。

ルーティングに用いるテーブルには

- タプルを送る宛先 (タプルスペースの ID)
- 宛先に送る為の次のタプルスペース
- ホップ数

の情報が含まれている。タプルスペースの ID は、“ホスト名:ポート番号” としている。これは、タプルスペースに一意な ID を割り振る目的でこの型式を取った。宛先に送る為の次のタプルスペースは、宛先のタプルスペースに送るために、隣り合ったどのタプルスペースにタプルを送るべきかを表わす。ホップ数は、宛先へ送り届けるために通るタプルスペースの数を表わす。

タプルを送る場合、宛先と送りたいタプルを指定する。ルーティングエージェントは、その指定された宛先を、自分が保持しているルーティングテーブルから引き、次にどのタプルスペースへタプルを渡せばいいかを判断する。テーブルを表すデータ構造は、perl の hash を用いて実装した。

ルーティングプロセスは、起動すると、まずルーティングテーブルの作成を行う。起動直後は、自分

が担当するタプルスペースへ接続し、その ID をルーティングテーブルへ登録する。次に、他のタプルスペースへの接続を行う。この場合も各タプルスペースの ID をテーブルへ登録する。このとき、次のホップ、ホップ数も一緒に登録される。hash へは宛先をキーとして、次のホップとホップ数が引けるように実装した。

```
register_tuplespace($tsid,$nexthop,$hopnum);
sub register_tuplespace {
  my ($ts_dst,$ts_nexthop,$ts_hopnum) = @_;
  if ($routing_table{$ts_dst}) {
    undef $routing_table{$ts_dst};
  }
  # タプルスペースの ID を key に、
  #次のホップ、ホップ数を value に
  $routing_table{$ts_dst} =
    {"nexthop" => $ts_nexthop,
     "hopnum"  => $ts_hopnum}
  return;
}
```

作成されたテーブルは、他のタプルスペースへも伝えられる。hash で実装されたテーブル情報は XML 形式に変換される。テーブルを更新したということを、テーブル情報とともに直接接続しているタプルスペースへ伝える。

更新情報を受け取ったタプルスペースでは、ルーティングプロセスがその情報を受け取り、自身のルーティングテーブルを更新する。更新は受け取ったテーブルと自身のテーブルの統合をする。統合は、受け取ったテーブルにおいて

1. 知らない宛先
2. 既知の宛先だが、ホップ数が小さい

場合のみ、自身のテーブルに統合する。ホップ数は登録するときに 1 増やしている。

```
update_routing_table($got_table, $nexthop)
sub update_routing_table {
  my ($ts_table, $ts_from) = @_;
  my $update = 0;
  foreach my $dst (keys(%$ts_table)) {
    if ($dst eq "name") {
      next;
    }
    # 既に登録されていて、
    #かつホップ数が大きいものは登録しない
    if ($routing_table{$dst}) {
      if ($routing_table{$dst}{"hopnum"} <=
            $$ts_table{$dst}{"hopnum"}+1) {
        next;
      }
    }
    register_tuplespace($dst, $ts_from,
           $$ts_table{$dst}{"hopnum"}+1);
    $update = 1;
  }
  return $update;
}
```

この更新作業を繰りかえし、各タプルスペースでルーティングテーブルが安定するようにする。

このように作成されたルーティングテーブルをもとに、各ルーティングエージェントはタプルスペースに投入されたタプルを宛先まで届ける。

8 type 2 から type 3 へ

type 2 は比較的容易に実装できたので、その上でいくつかの分散アプリケーションを記述することが次の目標である。例えば、

```
Routing Protocol
DNS
Multi-caset
Snapshot
Debugger
```

などを実装することが可能であると思われる。

これらを実現する分散アルゴリズムはさまざまなものがあるが、アプリケーションプログラム程の多様性はないと期待される。もし、それが十分に有限、あるいは、限られた機能ですむのならば、Engine のプログラムは、API として固定することが出来る。

もし、そのような API を決めることが出来れば、分散プロトコルの抽象化が実現できる。そのアルゴリズムに対して、途中のタプルでの計算などの具象化のための API (関数の登録など) を type 3 の API として用意してやれば良い。type 3 がそのように設計されれば、Federated Linda のプログラミングは、以下のようになる

分散プロトコルを選択して具象化し、それを末端の UI より呼び出す

しかし、分散アルゴリズム自体が十分な多様性を持つ可能性もあり、また、独自な分散アルゴリズムを実装する必要もある。そのような場合は、Engine の API を固定することは出来ず、Engine 自体をプログラムする必要がある。その場合は、type 3 の API は、Engine 上で任意のプログラムを動かすための配布機構を持つ必要がある。

User Interface あるいは、一般のアプリケーション (例えばブラウザなど) との接続は、Engine の一種となる。type 3 では、Linda と Engine は一体化するので、この場合は、UI またはアプリケーションがタプル空間を持つことになる。

タプル空間に持続性を持たせることも可能であるが、その場合は、他のタプル空間との整合性が問題となる。

自動的な Engine と Linda の構成、また、構成の管理は現状では未定である。type 2 では起動スクリプトのような形で実現する。

9 他の分散フレームワークとの比較

分散フレームワークは、通信機構を使いやすくしたものであることが多い。オブジェクトをシリアライズしてネットワーク上で呼び出せるようにする、あるいは、XML SOAP [3] で実現するなどである。

Federated Linda では、タプル空間を使うことにより、分散アルゴリズムの記述と、末端での分散アルゴリズムへの接続を分離することができる。分散アルゴリズムは、Linda により比較的容易に記述することが出来る。

XML-SOAP などと異なり、Federated Linda はタプルのやり取りというモデルを持っている。RPC のように、通信よりも、相手側の処理を呼び出すと言う手法では、通信の様子 (待ちキューなど) を視覚的に認識することが難しい。in や out によってタプル空間にどのようなことが起きるかは明解で理解しやすい。

Engine は Perl で記述されるために容易かつ簡潔に記述することができる。

タプル空間への接続は、サーバ・クライアント的であり、切断や再接続が容易である。

タプルの ID という間接的な名前でアクセスするために、分散アプリケーション上の任意のオブジェクトに直接アクセスするということはできない。それは Engine 上にルーティング・プロトコルを実装して初めて可能になる。普通にタプル空間にアクセスしている場合には、アクセスの集中は起きない。アクセスの集中が起きるのは、Engine のアルゴリズムによってである。Eingine が実現する分散アルゴリズムがアクセス集中を避けるように構築されていれば、そのアルゴリズムを選択するだけで良い。

Linda は、特に、こちらで用いている非同期型の Linda API ではポーリング型のプログラミングになることが多い。通信は必ずタプル空間を経由するので、直接的な通信よりも低速である。これは、Linda の欠点をそのまま引き継いでいる。

しかし、例えば、N ノードのマルチキャストを考えると、通常の Linda では 2N のメッセージが一つのサーバに集中するが、Federated Linda で木構造を構築すると、6 log N のメッセージが分散して処理される。

10 まとめ

玩具的で教育的な分散プログラムのフレームワークとして、複数の Linda サーバを接続したモデルを提案し実装した。

他のシステムに比べての利点と欠点について考察を行った。ここで提案した type 3 の設計を明確にするために、type 2 でのプログラミング経験を積むことが重要であると思われる。

参考文献

[1] 河野 真治, 仲宗根 雅臣: 同期型タプル通信を用いたマルチユーザ Playstation ゲームシステム, 卒業論文, 1998

[2] WSDL(Web Services Description Language). http://www.w3.org/TR/wsdl20/

[3] Simple Object Access Protocol (SOAP). http://www.xml.org/

[4] CORBA. http://www.omg.org/

[5] Jini. http://www.jini.org/

[6] JXTA. http://www.jxta.org/

[7] XML SOAP. http://www.w3.org/TR/soap/