連邦型タプルスペースを使ったコンパクトルーティングの実験

渕 田 良 彦 † 河 野 真 治 ††

連邦型タプルスペース (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て分散システムを構築する分散タプルスペースである。Federated Linda においてタプルスペース同士の接続を担当するプロセスとして Protocol Engine があり、それはプロトコル及びルーティングアルゴリズムを規定する。本稿では、動的ルーティング (Compact Routing) を行う Protocol Engine の実装を元に Fedareted Linda における分散プログラミングについて論じる。

Experiment on compact routing in federal type tupple space

Yoshihiko FUCHITA † and Shinji KONO ††

Federated Linda is decentralized tupple space that unites two or more tupple space, and spreads the tupple, and constructs the distributed system between those. In Federated Linda, Protocol Engine take in charge of the connection of the tupple space, and provides for the protocol and the routing algorithm. In this text, distrebute programming in Federated Linda is discussed by implementation of Protocol Engine performing dynamic routing.

1. はじめに

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で並列的に実行されるプロセス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。また、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。

それに対して我々は、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として大域 ID を持たない連邦型タプルスペース [2][3](以下 Federated Linda と記す) を提案してきた。

Federated Linda においてタプルスペース同士の接続を担当するプロセスとして Protocol Engine があり、それは現在の実装では非同期の Linda API を C 言語モージュルで提供し、Perl, Ruby, Python スクリプトによって記述される。

本稿では、コンパクトルーティング (Cowen,[4]) を行う Protocol Engine の実装を Python を用いて行った。コンパクトルーティングとは、ネットワーク上での接点間のパケットルーティングで、最短経路を保証しない代わりに各節点のルーティングテーブルのサイズを小さくする近似アルゴリズムの一種である。コンパクトルーティングを実装する上での利点は、ルーティングテーブルの収束が速く、スケーラビリティがあるといった点である。

また、今回の実装を通して感じたテスト駆動開発での分散プログラミングの問題点とそれを解消する手段についても報告する。

† 琉球大学理工学研究科
Graduate School of Engineering and Science, University of the Ryukyus
†† 琉球大学工学部情報工学科
Information Engineering, University of the Ryukyus

2. タプル空間を用いる分散プログラミングモデル Linda

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外部との通信を行う分散プログラミングモデルである。(図 1)

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、空間に接続するクライアントの構成の変更も容易である。

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。

表 1　Linda API

	id に対応する tuple
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
read(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

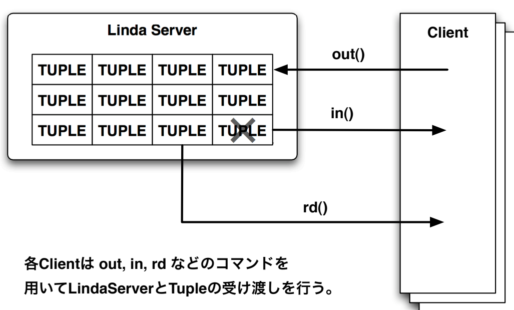

図 1　LindaServer

3. Federated Linda の提案

Federated Linda は、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、多数のタプル空間が接続により分散アプリケーションを実現する (図 2 参照)。smtp/nntp デーモンが行単位でプロトコルを作るのと似た形で、タプル空間への in/out でプロトコルを作ることになる。

通信モデルはタプルの出し入れによる、リレー転送のようになる。インターネットのパケット転送のように、タプルスペースからタプルスペースへとタプルを転送していく。

クライアントのアクセス数が増えても、タプルスペース等の数を増やし、処理を分散することにより、スケーラビリティを保つ。

4. Federated Linda の分散プログラミング

分散プログラムには “Local Access to Protocol”, “Protocol Engine”, “Link Configuration” の 3 つの要素がある。Federated Linda はこの 3 つの要素に基づいてプログラミングモデルを提供する。

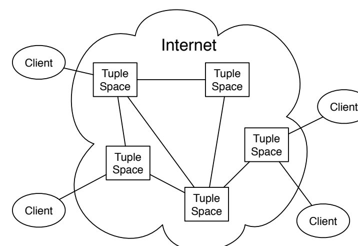

図 2　タプルスペースの相互接続

4.1 Local Access to Protocol

プロトコルへのアクセスは Linda の API を用いる。つまり、タプルスペースへの “in”, “read”, “out” などである。これらのコマンドは単純で、理解しやすいものである。タプルの出し入れというモデルで通信を行うことができる。また、通信は非同期に行う。この API を用いるプログラムはポーリングベースのプログラミングスタイルを取る。

4.2 Protocol Engine

プロトコルを規定する Protocol Engine は、分散アルゴリズムを内包し、他のプロトコルへのアクセスもタプルスペース経由で行う。通信はタプルをタプルスペースからタプルスペースへと、バケツリレーの様に転送する。クライアントとはタプルスペースを介した通信を行うので、クライアントからはプロトコルの細かい挙動は見えない。しかし、クライアントプログラムのプロトコルへの依存を低く抑えることが可能である。

4.3 Link Configuration

タプルスペースや Protocol Engine の接続を規定する。接続の状況を XML として表し、各ノードがそれにしたがって論理ネットワークを構築する。論理ネットワークを構築するために、接続などを扱うモジュールを提供する。

5. Federate

Federate とは「連合の、連邦型」という意味がある。これは、タプルスペースを介して複数のプロトコルが緩く接続することにより、異なるシステム間のデータのやり取りが可能となり、連合型の分散システムを構築することを表している (図 3 参照)。この連合型の分散システムにより、より自由度の高いシステムを構築することが期待できる。例えば、ルーティングプロトコルで指定した先でブロードキャストを行う、などのようなことができるようになる。

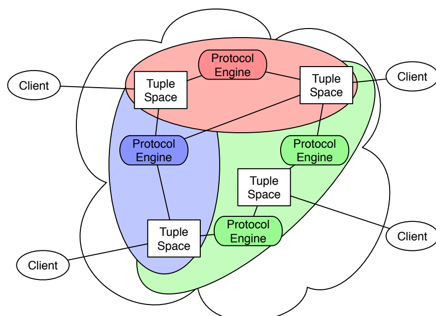

図 3　連邦型の分散システム

6. 実装の段階分け

Federated Linda の実装には次の 3 つの段階がある。

type 1　オリジナルの非同期型 Linda

type 2　複数の LindaServer をエージェントが橋渡しする

type 3　タプル空間自体が直接接続される

この段階分けは実装の容易さで決められている。type1 はすでに実装されている非同期型 Linda である。type2,type3 からはその前のタイプのプログラミングの経験よりその詳細な仕様を決定する。以下でその説明を行う。

6.1 type 1

type 1 は通常の非同期型 Linda プログラムである (図 4 参照)。タプルスペースを持つ Linda サーバに、Client プログラムがタプルを出し入れすることにより、分散プログラムを実現している。

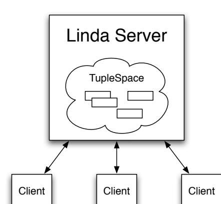

図 4　type 1

6.2 type 2

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、Protocol Engine と呼ばれるエージェントで接続される。Protocol Engine から Linda へのアクセスは通常の Linda API で行われる。(図 5 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 Protocol 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、Protocol Engine は Python 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。

Protocol 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、Protocol Engine のプログラムによって決まる。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。

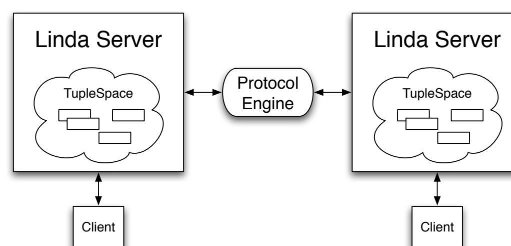

図 5　type 2

6.3 type 3

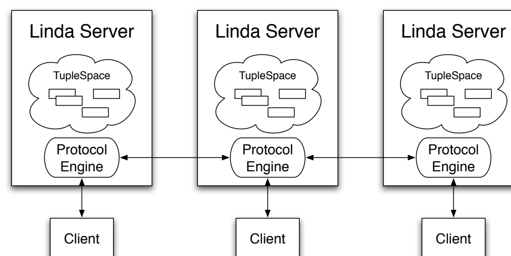

図 6　type 3

type 3 では、Protocol Engine は Linda と一体化して抽象化される。(図 6 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。特に Protocol Engine

は、必要となる分散プロトコルが十分に有限なら API として提供し、API として提供できないぐらい多様性を持つならプログラムのロード機構が必要になる。よって、type 2 で様々な Protocol Engine を実装する必要がある。

7. type2 によるルーティングアルゴリズムの実装

type2 による Federated Linda の実装には、Protocol Engine によるタプル空間のルーティングアルゴリズムが必要になる。我々は既に、Distance Vector 型ルーティングプロトコルを行う type2 の Federated Linda の実装を持っているが [3]、この方式は実装が容易という長所がある半面、ルーティング情報の伝播が遅いほか、ネットワークの規模が大きくなるとさまざまな運用上の問題がでてくるという欠点がある。そこで、今回新たにルーティングテーブルの収束速度やネットワークのスケーラビリティに対して優位なルーティングアルゴリズムである。コンパクトルーティング (Cowen,[4]) を行う Protocol Engine を実装した。

8. 実装の詳細

コンパクトルーティングを実装した Protocol Engine が持つルーティングテーブルはタプル空間に格納された XML ファイルから生成される。ルーティングテーブルとタプル空間でやり取りされる XML データの関係を図 7 に示す。

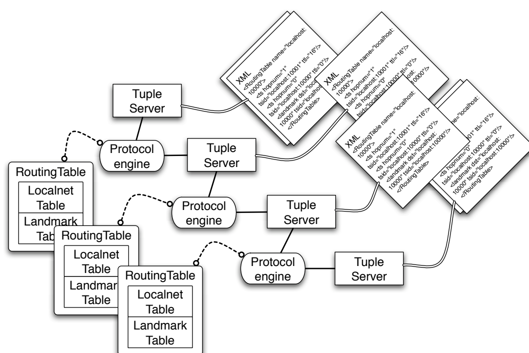

図 7 ルーティングテーブルとタプル空間でやり取りされる XML データの関係

また、Local Access Protocol, Link Configuration に関しては Distance Vector 型ルーティングを行う実装 [3] と同様のものを利用する。このような実装が可能なのは Federated Linda が分散プログラムの 3 つの要素を明示的に分けて記述している為であり、具体的なこれら 3 つの実装は表 2 のようになっている。

表 2 実装の詳細

Local Access to Protocol	Linda API, C 言語モジュール
Protocol Engine	Python スクリプト
Link Configuration	Omni Graffle ファイル, XML 生成 Python スクリプト

8.1 コンパクトルーティングのアルゴリズム

コンパクトルーティングでは、各ノードがある一定の範囲の近傍ノードに対しては完全なルーティング情報を持ち、遠くのノードに対しては Landmark と呼ばれるいくつかの選ばれたノードに対してのルーティング情報のみを持つようにすることでルーティングテーブルのサイズと更新量を抑える事が出来る。以下にコンパクトルーティングによるルーティングアルゴリズムの全体構造を示す。

アルゴリズムの全体構造

[準備]
・XML によりタプル空間同士のネットワークと Protocol Engine の Link Configuration を行う。

[ルーティングテーブル構築]
・全ノードは自身に近いノード、つまりローカルネットワークのルーティングテーブルを構築する。
・全 Landmark を求め、全ノードは全 Landmark に対するルーティング情報を求める。

[ルーティング]
・宛先は Landmark のアドレスと転送先アドレスの 2 つを持つ
・投入時のローカルネットワークに宛先があるときはローカルネットワークのルーティングを行う。
・それ以外の時、最寄りの Landmark を経由し宛先 Landmark へ転送する。

8.2 Protocol Engine によるルーティングテーブルの構築

Protocol Engine は Python スクリプトで記述され、コンパクトルーティングを行う為のルーティングテーブルを生成する。ルーティングに必要なテーブルはローカルエリアネットワークと Landmark に関するルーティングテーブルとなる。

8.2.1 ローカルエリアネットワークの構築

コンパクトルーティングを実装する場合、ローカルエリアネットワークを構築しなければならない。そのネットワークは、Land Mark を中心とするネットワー

クである (図 8)。

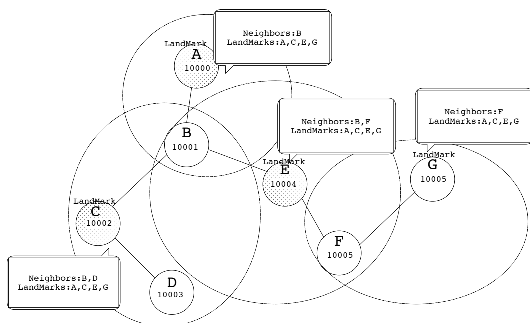

図 8 ローカルエリアのホップ数が 1 の場合

8.2.2 LandMark 情報の更新

全ノードは全ての Landmark に対するルーティング情報を持つよう更新される。仮にローカルエリアが Land Mark から 1 ホップの範囲だとすると、図 9 のように Land Mark 情報が更新される。

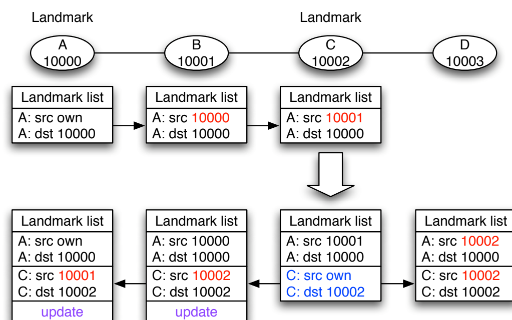

図 9 ローカルエリアのホップ数が 1 の場合の Land Mark 情報の流れ

8.3 Protocol Engine の継承による実装

コンパクトルーティングを行う Protocol Engine は Python スクリプトで記述される。Python はとても汎用的で上位レベルな、オブジェクト指向のプログラミング言語であり、それによって Distace Vector 型ルーティング [3] を行う Protocol Engine の Python スクリプトを、コンパクトルーティングを実装するいくつかの点において Python コードの継承や流用を行うことが可能となる。例として、ルーティングテーブルの情報について挙げると、Distance Vector 型ルーティングの実装で用いたルーティングテーブルを、クラスの継承を用いる事でコンパクトルーティングにおけるローカルエリアネットワークへのルーティングテーブルとして利用可能である。

DV 型ルーティング実装 Routing.py

```
# 他のタプルスペースの情報を表すクラス
class TSInfo:
    def __init__(self, tsid, hopnum, nexthop):
        self.tsid = tsid
        self.hopnum = hopnum
        self.nexthop = nexthop

# TSInfo のリストでルーティングテーブルを表すクラス
class RoutingTable:
    def __init__(self, name):
        self.tslist = {}
        self.name = name
```

C ルーティング実装 CompactRouting.py

```
# Landmark 情報を表すクラス
class Landmark_TSInfo:
    def __init__(self, landmark_tsid, landmark_dst):
        self.landmark_tsid = landmark_tsid
        self.landmark_dst = landmark_dst

# クラス RoutingTable を継承し、Landmark のリストを持つクラス
class CompactRoutingTable(RoutingTable):
    def __init__(self, name):
        RoutingTable.__init__(self, name)
        self.landmark_tslist = {}
```

上記のように定義した場合、クラス CompactRoutingTable は Distance Vector 型ルーティングで定義された、クラス RoutingTable を継承して利用できる。

また、Protocol Engine のメインループにおいても、変更が必要なメソッドのみをオーバーライドして定義することで、Link Configuration などの動作が以前の実装と同様でも良い部分のコーディングを省略することができる。以下にそのメインループのソースコードを示す。

Protocol Engine のメインループ

```
while(True):
   # Link Configuration
   rep = linkConfigReply.reply()
   if(rep):
       linkConfigReply = self.linda.In(TUPLE_ID_LINKCONFIG)
       # Link Configuration main (use Routing class method)
       self.LinkConfig(self.tsid, rep)

   # Routing Protocol
   rep = routingReply.reply()
   if(rep):
       routingReply = self.linda.In(TUPLE_ID_ROUTING)
       cmd , data = self.unpack(rep)
       # connect to other tuplespace
       if (cmd == ROUTING_COMMAND_CONNECT):
           self.RoutingConnect(data)
       # disconnect other tuplespace
       elif (cmd == ROUTING_COMMAND_DISCONNECT):
           self.RoutingDisconnect(data)
       # transfer tuple
       elif (cmd == ROUTING_COMMAND_TRANSFER):
           self.RoutingTransfer(data)
       # update own routing table
       elif (cmd == ROUTING_COMMAND_UPDATE_TABLE):
           self.RoutingTableUpdate(data)
       else:
     self.flinda.sync()
#end while
```

このように、Python による Protocol Engine の実装は、仮に新しい実装が必要になった場合でも新たに機能を拡張する差分だけをプログラミングすればよいので、ソフトウェア開発の効率化が行えると言える。

8.4　ルーティングテーブルの構築、更新処理

ルーティングテーブルの構築、及び更新処理は RutingTableUpdate メソッドで行われ、それは Distance Vector 型ルーティングにおける実装での同名メソッドに対してオーバーライドして定義される。以下にそのソースコードを示す。

RoutingTableUpdate メソッド

```
def RoutingTableUpdate(self,data):
    print "update"
    srcname = self.rt.getdstname(data)

    # Landmark find
    if(self.rt.getNewLandmark()):
         self.rt.landmark_register(self.rt.name,
                         self.rt.name)

    if(self.rt.update_withLandmark(data, srcname)):
         # Send Update Info to Neighbors
         upedxml = self.rt.getxml_withLandmark()

         for n in self.neighbors.values():
             n.Out(TUPLE_ID_ROUTING, self.pack(upedxml,
                        ROUTING_COMMAND_UPDATE_TABLE))
```

RoutingTableUpdate メソッドにおいてはルーティング情報を表す XML を受け取り、それに記された送信元ノードの情報を取得する。また、自ノードが新しく Landmark になる条件を満たしていた場合は自身の Landmark テーブルに自ノードを追加した後、update_withLandmark メソッドでルーティングテーブルの更新処理を行う。それから更新された情報を元に新しい XML ファイルを生成し、近傍ノードへ対して送信する。

update_withLandmark メソッドにおいては、受け取った XML ファイルと送信元ノードの情報を引数として受け取り、XML ファイルを読み込んでルーティングテーブルの更新を行い、ローカルネットワークと Landmark のルーティング情報を登録する。

ローカルエリアネットワークの更新は受け取ったテーブルと自身のテーブルの統合を行う。統合は、受け取ったテーブルにおいて

(1)　ローカルエリアのホップ数上限に満たない (ttl による)

(2)　知らない宛先かまたは

(3)　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、TTL(Time To Live) を 1 減らす。

Landmark に関する情報の更新は、現在の Landmark テーブルに存在しない Landmark 情報がある場合に、その Landmark のアドレスと転送先アドレスとして、引数で与えられた送信元アドレスを登録する。

これらの一連の処理により、ルーティングテーブルの構築、及び更新処理が完了する。

update_withLandmark メソッド

```
def update_withLandmark(self, xmldoc, src_ts):
    rt = xml.dom.minidom.parseString(xmldoc).childNodes[0]
    tslist = rt.childNodes
    updateflag = False

    tmplist = []
    tmplandmarklist = []

    for t in tslist:
        # append tuplespace
        if t.nodeType == t.ELEMENT_NODE
        and t.localName == 'ts':
            tsattr  = t.attributes
            tsid    = tsattr['tsid'].nodeValue
            hopnum  = int( tsattr['hopnum'].nodeValue )
            ttl     = int( tsattr['ttl'].nodeValue )
            nexthop = src_ts

            tmplist.append(tsid)
            if (((not self.tslist.has_key(tsid))
                  or (self.tslist[tsid].hopnum > hopnum+1))
                  and (ttl-1 > 0)):
                    self.register(tsid, hopnum+1, ttl-1,
                                               nexthop)
                    updateflag = True

        #parse landmark tsid/dst for register
        elif t.nodeType == t.ELEMENT_NODE
        and t.localName == 'landmark':
            lkattr = t.attributes
            landmark_tsid = lkattr['tsid'].nodeValue
            landmark_dst  = lkattr['dst'].nodeValue

            tmplandmarklist.append(landmark_tsid)

            if (
            not self.landmark_tslist.has_key(landmark_tsid)):
                self.landmark_register(landmark_tsid, src_ts)
                updateflag = True

    # delete tuplespace
    for t in self.tslist.values():
        if (( not t.tsid in tmplist ) and (t.ttl-1 > 0)):
            updateflag = True
            if (t.nexthop == src_ts):
                del self.tslist[t.tsid]

    for lt in self.landmark_tslist.values():
        if ( not lt.landmark_tsid in tmplandmarklist ):
            updateflag = True

    return updateflag
```

9. ルーティングテーブル構築時間の測定

ここで、Distance Vector 型のルーティングテーブルを構築する方法 [3] と今回実装したコンパクトルーティングを行う方法のルーティングテーブルの構築時間をそれぞれ 10,20,30 のノードでトポロジは Line 型トポロジと Binary Tree 型トポロジを用いて測定した。またこの場合、コンパクトルーティングを行う際のローカルネットワークのホップカウントは 2 と固定して測定を行う。

9.1 測定の流れ

ルーティングテーブルについては、LinkConfiguration からの他タプルスペースへの接続要求を受け取ってから、ルーティングテーブルが最短経路を構築するまでの時間を各ノードで計り、その平均を出した。

以下の順序で実験を行う

(1) Tuple サーバーと Protocol Engine を全ノードで起動しておく。

(2) LinkConfigure.py で XML を作成、初期ノードへ XML を投入する (測定開始)。

(3) 収束を確認して、結果を集計

Distance Vector 型ルーティングは最短パスを構築するので、全ノードで各ホップ数が最短になったら収束したと判断した。

以下に Line 型トポロジと Binary Tree 型トポロジのルーティングテーブルを構築したときの構築時間の結果を載せる (表 3),(図 10)。

表 3 Routing Table 構築にかかる時間

Compact Routing		
トポロジ	ノード数	かかった時間 (sec)
Line	10	1.19
Tree	10	2.40
Line	20	5.18
Tree	20	6.95
Line	30	7.56
Tree	30	26.26
Distance Vector Routing		
トポロジ	ノード数	かかった時間 (sec)
Line	10	2.66
Tree	10	4.54
Line	20	89.66
Tree	20	29.91
Line	30	8634.50
Tree	30	127.55

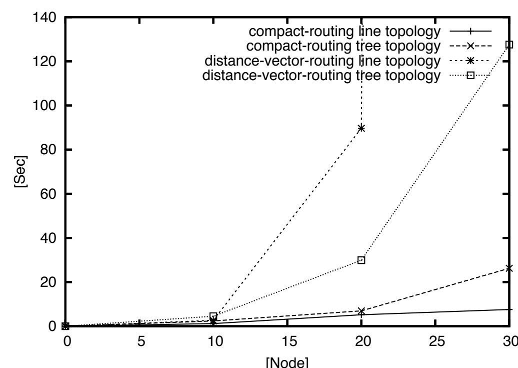

図 10 ノード数に対してルーティングテーブルを構築するのにかかる時間

上記の結果より、コンパクトルーティングでのルーティングテーブル構築時間は Distance Vector 型でのルーティングテーブル構築時間より高速に構築可能であり、ノード数に対しての増加比も Distance Vector 型よりもスケーラビリティに優れているという事がわかる。

10. テスト駆動開発における分散プログラミングの問題点

今回、Federated Linda に新たにコンパクトルーティングを行う Protocol Engine を実装するにあたって、図 11 に示すような多ターミナルでのテスト駆動開発によって開発を行ったが、実際に開発を行ってみて気がつく開発のやりにくさやデバッグ環境の不備があった。

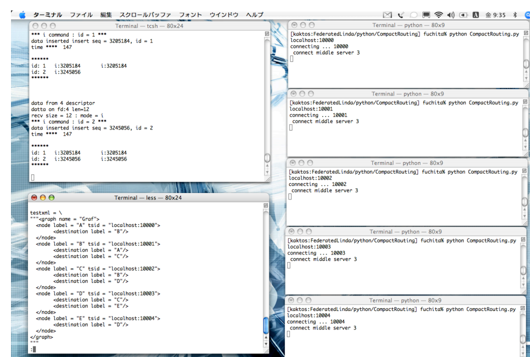

図 11 Federated Linda のテスト状況

特に問題となった点は一旦タプル空間に保持された値が、一度タプルサーバーを終了させないと初期化できないという点である。この問題点を改善する為にはタプルサーバーにタプル空間の初期化や処理データを

ダンプできるような機構が必要だと考える。

具体的な例として Federated Linda が分散プログラミング環境でありながら、その通信データは通信プロセスではなくタプル空間に保持されるという点を利用したデバッグインターフェースを提案する。タプル空間に対して、通信プロセスとは別の外部からのタプル参照を許可するインターフェースを定義してやれば、通信プロセス (Federated Linda と通信するクライアントプログラム) を改変する事無く、タプル空間のデバッグ操作を行うことができると考えられる。

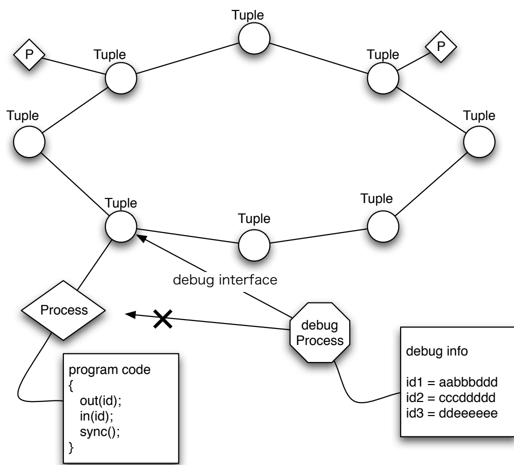

図 12 タプル空間へのデバッグインターフェース

11. まとめ

Distance Vector 型ルーティングを行う時、ルーティングテーブルが構築されるまでにかかる時間は、ノード数に対して２乗オーダ的に増加する。これは、ルーティングテーブルの更新情報をノードが受け取ると、自身のルーティングテーブルを更新し、接続している他のノードへ更新情報を配信するため、メッセージ数が２乗オーダに増える為だと考えられる。対して、コンパクトルーティングを行う実装では、持つべきテーブルはローカルエリアと全 Landmark の情報となるので、Landmark の数が少ない、即ちローカルエリアの大きさが大きければ大きいほどテーブルサイズは小さくなる。またネットワーク全体を伝搬する Landmark のルーティング情報も Distance Vector 型のルーティング情報と違い、転送先とアドレスの２つしか持たないので、総じて Distance Vector 型よりルーティングテーブルの構築にかかる時間においてスケーラビリティに優れるという結果になった。

Protocol Engine の実装については Python を用いた差分を拡張するプログラミングスタイルを紹介し、実際のルーティングテーブルの構築、更新処理について説明した。

またテスト駆動開発で分散プログラミングを行うのには開発環境の整備も重要な要素であることを、今回の実装を通して知る事が出来た。現在の Federated Linda は他の提供されている分散プログラミング環境 (CORBA[8],JAXTA[9],Overlay Weaver[10]) と比べて開発の為の環境整備が不十分であり、その環境整備も必要である。

今後の課題としては、今回実装したコンパクトルーティングの実装を与えられたトポロジ情報に沿って自動的に最適な Landmark を決定するように改良することや、タプル空間へのインターフェースを利用して動作中の Fedarated Linda での分散プログラムをデバッグするツールの開発などが挙げられる。

参考文献

1) 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 河野 真治, 仲宗根 雅臣, 卒業論文, 1998
2) 大域 ID を持たない連邦型タプルスペース Federated Linda 安村 恭一, 河野 真治, 第 99 回 情報処理学会 システムソフトウェアとオペレーティング・システム研究発表会, 2005.
3) 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, 日本ソフトウェア科学会第 22 回大会, 2005.
4) Compact routing with Minimum Stretch Cowen, SODA, 1999
5) Compact Name-Independent Routing with Minimum Stretch Ittai Abraham, Cyril Gavoille, et al, 2004
6) Compact Routing for Flat Networks Kazuo Iwama, Masaki Okita, Proceedings of 17th International Symposium on Distributed Computing (DISC 2003), Sorrento, Italy, pp.196–210, October 2003.
7) Grid/P2P プログラミングモデルと関連技術 田浦 健次朗, 東京大学, PPL Summer School 2005 資料集
8) CORBA. http://www.omg.org/
9) JXTA. http://www.jxta.org/
10) Overlay Weaver. http://overlayweaver.sourceforge.net/
11) WSDL(Web Services Description Language). http://www.w3.org/TR/wsdl20/
12) Simple Object Access Protocol (SOAP). http://www.xml.org/
13) XML SOAP. http://www.w3.org/TR/soap/