JAIST Repository
https://dspace.jaist.ac.jp/

Title	ネットワークシミュレータSSFNetの分散プログラミングフレームワークNekoへの統合
Author(s)	松下，誠和
Citation	
Issue Date	2005-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/1919
Rights	
Description	Supervisor:Defago Xavier，情報科学研究科，修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

ネットワークシミュレータSSFNetの分散プログラミングフレームワークNekoへの統合

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

松下 誠和

2005年3月

修 士 論 文

ネットワークシミュレータSSFNetの 分散プログラミングフレームワークNekoへの統合

指導教官　Defago Xavier 特任助教授

審査委員主査　Defago Xavier 特任助教授
審査委員　片山卓也 教授
審査委員　鈴木正人 助教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

310105 松下 誠和

提出年月: 2005年2月

Copyright © 2005 by Matsushita Tomokazu

概 要

現在、ネットワーク化された計算機は、毎日の生活の最も必要な関連として提供されている。そして、社会の必要不可欠な部分は、ネットワーク化された計算機を構成する分散システムに頼って形成されている。例えば、トランザクション処理を有するような金融機関の ATM システムや航空機会社の時間の制約が厳しい発券機システムなどである。従って、各分散システムの安全性と耐故障性を構成することは、最っとも高い優先度とされる。

このような分散システムを構築するためには、Replication 技術やその技術を達成するために用いられる Atomic Broadcast のような分散アルゴリズムの開発、性能評価、および、研究が必要である。分散アルゴリズムの開発および性能評価を行うことは、複雑かつ多くの時間を要する作業である。なぜなら、複雑なシステムを想定してアルゴリズムを開発および性能評価を行わなければなず、容易に評価環境を構築できないからである。性能評価を行うためには、シミュレーション環境でアルゴリズムを実行、および、実システム環境でアルゴリズムを計測などが挙げられるがいずれの方法も、評価したいアルゴリズムをプログラミング言語を用いて、研究者自身で実装しなけらばならない。この複雑な作業による分散アルゴリズムの研究者への負担は、分散アルゴリズムを開発、および性能評価するすべての行程をサポートする単一の環境が存在しないことが主な要因である。

本研究の対象である Neko は、分散アルゴリズムを開発および性能評価を行うための分散プログラミングフレームワークである。Neko 単体で分散アルゴリズムの開発、および、性能評価に必要なすべての行程を行うことが可能である。つまり、Neko では、Neko のフレームワークを用いて実装された分散アルゴリズムを Neko 上に実装されているシミュレーション環境および実システムの環境の両方で性能評価可能であり、開発者が異なる複数の環境を用いる必要がない。また、Neko のフレームワークは、分散アルゴリズムを実装するために必要な機能（単純なメッセージインタフェース、異なるアルゴリズムを階層的に構築する構造 etc）を提供している。開発者は、このような Neko のフレームワークを用いることで、簡単に分散アルゴリズムを実装することが可能である。しかし、Neko のシミュレーション環境を用いて性能評価を行う場合、重大な問題点がある。問題点とは、Neko 上に実装されたシミュレーション環境は、実世界で用いられるような計算機ネットワークをモデル化して実装されていないので、現実的なシミュレーションネットワーク環境での性能評価を行えないことである。

この問題を解決するために本研究では、現実的な計算機ネットワークをモデル化し実装しているネットワークシミュレータである SSFNet を分散プログラミングフレームワークである Neko への統合を行う。具体的な統合方法は、Neko と SSFNet の間に Neko が SSFNet のシミュレーション環境を利用するためのインタフェースを構築する。統合する上で実装上の問題がいくつかあり、Neko と SSFNet のスケジューラの同期、アドレス方式の違い、メッセージ形式の違いなどである。これらの問題を解決することによって、Neko のネットワークシミュレーション環境として SSFNet を用いるが可能となった。このような方法

で統合し、現実的なネットワークシミュレーション環境上で性能評価を行えることが可能となった Neko 上で、分散アルゴリズムの１つである Atomic Broadcast を用いて性能評価を行った。Atomic Broadcast とは、分散システム中の複数のプロセスがネットワーク上からの要求（メッセージ）を全順序で目的となる計算機に配達するアルゴリズムである。Atomic Broadcast は、Consensus とよばれる分散アルゴリズムを用いて実現する種類が存在する。本研究では、２種類の良く知られた Consensus を Atomic Broadcas の性能評価基準を用いて比較した。１つ目のアルゴリズムは、集中的な通信よりメッセージの配達する順番を決定（合意）を用いており ([8]) 、２つ目のアルゴリズムは、集中的な通信を用いていない ([7]) ２種類の Consensus の上に構成される Atomic Broadcast を Neko 上で実装されている Ethernet を単純にモデル化したシミュレーションネットワークと、本研究で統合した Neko を用いて性能評価し、比較を行った。また、Neko と SSFNet のシミュレーションネットワークのモデルを比較することにより、統合した Neko の有効性を示した。本研究により、分散アルゴリズムの研究者へ適切な分散アルゴリズムの開発と性能評価ための有用な結果を得られるような高性能な分散プログラミングフレームワーク Neko の提供が可能となった。

目 次

第1章　はじめに **1**

第2章　Neko と SSFNet について **3**
2.1 Neko とは . . . 3
2.1.1 Neko の問題点 . . . 7
2.2 SSFNet とは . . . 7
2.2.1 SSFNet のアーキテクチャ . . . 10

第3章　統合方法 **13**
3.1 統合デザインと実装上の問題の解決 . . . 14
3.1.1 SSFNet Networks . . . 17
3.2 動作テスト . . . 18

第4章　How to use Neko with SSFNet **20**
4.1 インストール方法 . . . 20
4.2 コンフィグファイルの設定 . . . 21
4.3 実行 . . . 26

第5章　性能評価 **29**
5.1 Atomic Broadcast とは . . . 29
5.1.1 Atomic Broadcast の定義 . . . 29
5.1.2 Atomic Broadcast アルゴリズム . . . 31
5.2 Evaluation Neko with SSFNet by Two Consensus Algorithm . . . 35
5.2.1 定義 . . . 36
5.2.2 ２つの Consensus Algorithms . . . 36
5.2.3 ベンチマーク . . . 38
5.2.4 シミュレーションモデルの比較 . . . 39
5.3 評価結果 . . . 41

第6章　まとめ **44**
6.1 謝辞 . . . 44

第1章　はじめに

分散アルゴリズムとは、分散システムにおける耐故障性などを保証するために用いられ、複数の計算機を利用して問題を解決する方法である。例えば、データベース等のサーバへデータの変更を行うような多くのシステムには、耐故障性をを高めるために Replication 技術が用いられている。Replication とは、クライアントからの要求を複数のプロセスに送信することにより、単一のプロセスでシステムを保持するよりも耐故障性を高める技術のひとつである。単一のプロセスでクライアントからの要求を処理する場合に、そのプロセスが停止すれば、そのクライアントからの要求を処理できないだけではなく、システム全体が停止してしまいサービスを提供できなくなる。Replication は、複数のプロセスの中から、 要求に対する処理や返答を主に行うリーダープロセスを決めておく。もし、そのリーダープロセスが何らかの故障でクラッシュしてしまった場合、他のプロセスが代わりにリーダープロセスとなり、システムを続行させる。しかし、システムを続行させるには、リーダープロセスと同じ要求を同じ順番で受信しておき、かつ、同じ順番で処理しなければばならない。このような技術を実現するためには、クライアントの要求をすべてのプロセスが同じ順番（全順序）で処理する必要がある。

全順序で要求を処理するために Atomic Broadcast という分散アルゴリズムを用いて、すべての要求を全順序になるようにプロセスに送信する。Atomic　Broadcast は、２つの性質を要求される。１つ目の性質は、送信されたの要求がすべてのサーバに届くか、あるいはどのサーバに届かないかのどちらかを保証することである。２つ目性質は、すべての要求が全順序ですべてのプロセスに届けられることである。この分散アルゴリズムを実現するためには、非常に複雑な処理をシステムが行わなければならない。本研究では、この Atomic Broadcast を用いて、第５章で性能評価を行っている。その章で、Atomic Broadcast の詳細について解説する。

分散アルゴリズムの研究者が、このような Replication 技術や Atomic Broadcast などの分散アルゴリズムを開発する場合、他の同じ種類の分散アルゴリズムとの有効性を示すためには性能評価が必要がある。しかし、性能評価を行うためには、実システム上にプログラミング言語などを用いて、分散アルゴリズムを実装し評価しなければならず多くの時間を要してしまう。さらに、多くの分散アルゴリズムが開発されているが、それらの性能評価を上記のような理由で、多く行われていない。研究者の負担を軽減するために、我々は分散アルゴリズムを容易に性能評価するためのツールを提供する必要がある。

本研究の目的は、NEKO のネットワークシミュレーションの問題点を SSFNet と統合することで解決し、拡張された NEKO の有効性を示すことである。NEKO と SSFNet 間

の問題であるスケジューラの同期、ネットワークアドレスやメッセージ形式の違いを解決し、NEKO と SSFNet のインタフェースを作成し統合する。そして、分散アルゴリズムである Atomic Broadcast を拡張された NEKO に実装し性能評価を行うことによって有効性を示す。

ここで、この論文の構成を解説しておく。第 2 章で、本研究の対象である Neko と SSFNet について解説し、第 3 章では Neko と SSFNet の統合についえ解説する。第 4 章では、第 3 章で統合したツールの利用の仕方を紹介し、第 5 章でツールと Atomic Broadcast の性能評価を行う。

第2章　NekoとSSFNetについて

本章では、NekoとSSFNetについて紹介する。NekoとSSFNetの構成、基本的な機能および、特徴について詳細に説明する。Nekoについては、その特徴的なレイヤー構造、メッセージを通信する方法やNekoの構成から得られる主要な部分について解説する。SSFNetについては、SSFNetのコンセプト、SSFNetで利用可能プロトコルやサンプルDMLファイルを用いての簡単なパラメタの解説する。

2.1　Nekoとは

NEKO[1] とは、Java言語で実装された分散アルゴリズムを性能評価および開発するためのフレームワークである。離散的なイベント駆動のシミュレータを持っており、NEKOのAPIを用いて分散アルゴリズムを実装することにより論理時間 [9] や実時間での経過時間、レイテンシやスループットを計測することが可能である。NEKOのAPIを用いて実装された同一のソースコードで複数の計算機を用いた実ネットワークでの実験と単一の計算機でのシミュレーションが可能である。つまり、分散アルゴリズムの性能評価に必要な行程をNekoだけを用いることのみでできる。シンプルなメッセージパッシング方式により、分散アルゴリズムを実装することが可能であり、分散アルゴリズムを実装するために必要としている機能をAPIとして用意している。これらの研究者は容易に分散アルゴリズムの実装と性能評価が行うことが可能である。

NEKOのアーキテクチャ（図2.1）は、アプケーションレベルとネットワークレベルの二つの主要な部分に分けることができる。アプリケーションレベルでは、NEKOのAPIを用いて分散アルゴリズムの振る舞いを実装する。複数のプロセスがシンプルなメッセージパッシング用のインタフェースを用いて、メッセージを非同期な伝播が可能である。メッセージを送信するプロセスは、sendという基本的な操作を用いてメッセージをネットワーク上に送り出す。そして、ネットワークは、そのメッセージを受信するプロセスへ、deliverという操作を用いて届ける。このようなプロセスは、複数のレイヤーによってプログラミングされる。

　　Nekoでは、メーセージを伝搬するための基盤であるネットワークは、いくつかの方法を用いて制御することが可能である。1つ目は、ネットワークは、あらかじめ定義されたネットワークである実ネットワークかシミュレートされたネットワークを用いてインスタンス化できるということ。２つ目は、Nekoは複数のネットワークを並行に適切に利用できるということ。最後に、Nekoに実装されたネットワークは、簡単にプログラミング

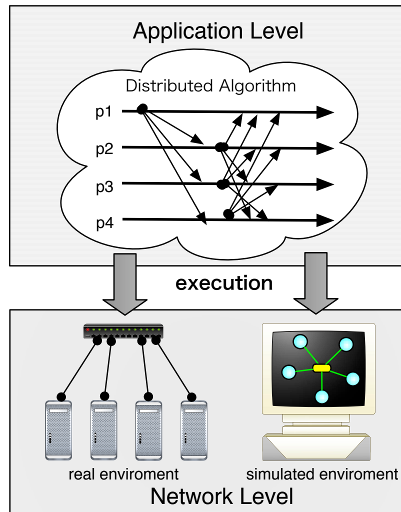

図 2.1: Architecture of Neko

でき、かつ新しいユーザー自身か定義したネットワークを追加可能である。次に、Neko のアプリケーションレベルとネットワークレベルの詳細について紹介する。

アプリケーションレベル：Neko のアプリケーションは、階層的なレイヤーとして構築される。送信されるメッセージは、send メソッドを用いて下位のレイヤーに渡される。そして、配達されるメッセージは、deliver メソッドを用いて上位のレイヤーに渡される。レイヤーは、*Ative* と *Passive* のどちらか一方の種類を用いる。Passive レイヤー（図 2.2）は、メッセージをコントロールするためのスレッドを持ち合わせていない。つまり、下位のレイヤーから、メッセージが Passive レイヤーに渡されるということは、下位のレイヤーは、その Passive レイヤーの deliver メソッドを呼んでいることになる。一方、Active レイヤー（図 2.3）は、メッセージをコントロールするためのスレッドを持ち合わせている。このレイヤーは、メッセージを下位のレイヤーから、receive メッソッドを用いて、能動的に受け取ることができる。Active レイヤーは、FIFO 形式のメッセージキューを持っており、下位のレイヤーからメッセージが Active レイヤーに渡されるということは、その Active レイヤーのメッセージキューへ追加していることになる。Active レイヤーが receive メソッドを呼ぶと、メッセージがメーセージキューに到着するまでブロックする。さらに、receive メソッドは、タイムアウトによりブロックする時間を制御可能である。recieve メソッドへの引数として、0 を与えることは、ブロッキングされないことである。つまり、メッセージの到着を待たないことである。Active レイヤーは、deliver メソッドを用いることで Passive レイヤーのように振る舞うことができる。これは、Active レイヤーのメッセージキューを迂回する方法として用いることと同じである。

開発者は、階層的な構造のようにアプリケーションを構築することを義務づけられているわけではない。つまり、レイヤーは違う方法によって結合させることができる。例え

ば、いくつかのレイヤーの 1 つの Neko メッセージのメッセージタイプに基づいて作成されたチャネルから、メッセージがあるレイヤーに伝播されたとする。そのメッセージタイプによって、レイヤーは、開発者が定義したメソッドにより相互にメッセージを受け渡すことができる。それらのメソッドは、send メソッドや deliver/receive メソッドによる制限を受けない。つまり、開発者は、すべてのタイプの Java オブジェクトを定義し、Neko 上で利用することが可能である。

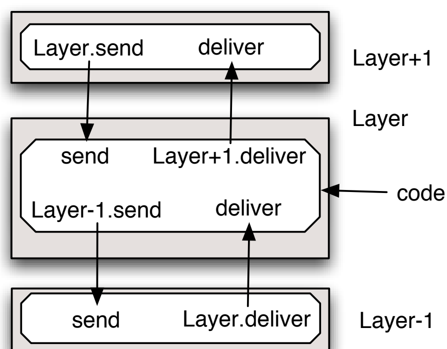

図 2.2: Passive Layer

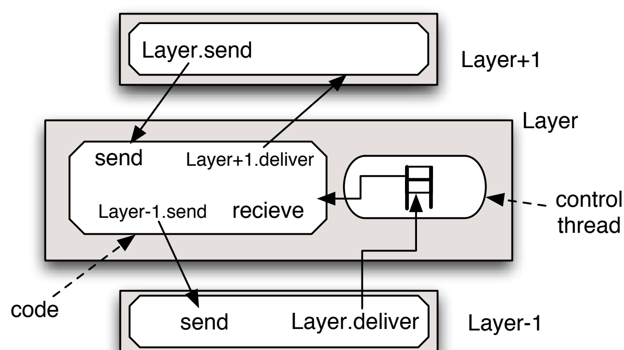

図 2.3: Active Layer

Neko プロセス：分散アプリケーションの各プロセスは、関連づけられた Neko プロセスを持っている。その Neko プロセスは、ネットワークとアプリケーションのレイヤーに設置されており、次に 3 つの重要な振る舞いに関するルールがる。

1. プロセスは、各プロセスのアドレスのような広い情報を持っている。そのプロセスのすべてのレイヤーは、すべてのプロセスにアクセスすることができる。それは、一般的な Single Multiple Data(SPMD) プログラミングに似ており、あるプログラム

は、いくつかのプロセスにより動作しており、Neko プロセスから得られるアドレスによって分岐している。

2. Neko プロセスは、プロセスによる送信や受信の履歴を取得する。

3. もし、アプリケーションが並行にいくつかのネットワークを使うなら、Neko プロセスは、適切なネットワークによるメッセージを処理する。(例えば、２つの違う物理的なネットワーク上で通信を行う場合や、違う二つのプロトコルで同じ物理的なネットワークを利用する場合)

Neko メッセージ：すべての基本的な通信 (send、deliver、receive）は、Neko メッセージのインスタンスによって転送される。あるメッセージは、ユニキャストまたは、マルチキャストのメッセージである。各メッセージは、Java のオブジェクトから成る content といわれる部分と下記に示すようなヘッダーによって構成されている。

Addressing(source（送信元アドレス）， destinations（目的地アドレス）)： アドレスの情報は、メッセージを送信したプロセスのアドレスと目的のプロセスのアドレスから成り立っている。アドレスは、小さな整数であり、０から開始され番号づけされる。

Network：Neko メッセージがいくつかのネットワークで用いられるとき、各メッセージは転送に用いられるべきネットッワークの ID に運ばれる。これは、メッセージが送信される時に、はっきりと区別される。

Message type 各メッセージは、開発者が定義したタイプの整数によるフィールドを持っている。それは、違うプロトコルに属するメッセージとしてを見分けるために使うことができる。

ネットワークレベル

Neko のネットワークは、低い下位のレイヤー（図 2.1）を構成する。開発者は、アプリケーションのコードを変更することなく、コンフィグファイルによってネットワークを区別することができる。

Real Network 実ネットワークは、Java ソケットから構築される（あるいは、他のネットワークライブラリー)。そのネットワークは、Java による連続化された Neko メッセージの content を用いる。Neko 上に実装されている TCPNetwork は、高信頼なメッセージ伝播方式である TCP/IP により構築される。TCP のコネクションは、各 1 組のプロセスの間に起動時に確立される。UDPNetwork は、信頼性の低いメッセージ伝播方式である UDP/IP により構築される。これらの実ネットワークのパフォーマンスについては、Urban らによって、[1] で性能評価を行っている。

Simulated Network

現在、Neko は単純なバージョンの Ethernet、および、FDDI のネットワークをシミュレート可能である。複雑な現象における Ehternet 上のコリジョンなどは、モデリングされていない。しかし、異なったモデリングによるネットワークを単純なネットワークイン

タフェースによって差し替え可能である。違う種類のシミュレーションネットワークは、分散アルゴリズムをデバック時に有用なものを提供できる。Neko に実装されているネットワークは、ランダムな時間を計算するか、または、定数による時間の経過の後にメッセージを伝播する。これは、モデリングした実際のパフォーマンスは、現実的な結果ではない。

2.1.1 Neko の問題点

NEKO のネットワークレベルには、複雑なネットワークモデルのシミュレーションができないという大きな問題点がある。NEKO のシミュレーションネットワークは、計算機同士が直接接続された完全結合のトポロジしか想定されていない。(図 2.4) さらに、ネットワーク間の帯域幅や遅延や NIC のレイテンシーなどを柔軟に設定できない。メッセージの伝播時間は、定数によってあらかじめ決められるか、または、ランダムに生成された遅延時間を用いている。ネットワークのトラフィックおよびバッファリングによる遅延などを実ネットワークを想定したシミュレーションを行っていない。よって、NEKO では現実的なネットワークシミュレート環境上で分散アルゴリズムを性能評価できない。

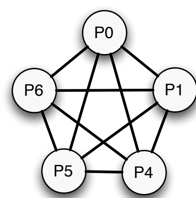

図 2.4: Neko Topology

しかし、Neko は、このような問題点は把握しており、開発者が簡単に他のネットワークを利用できるように設計されている。本研究では、この Neko の問題点である現実的なシミュレーションネットワーク環境を構築ができないことを SSFNet というネットワークシミュレータが Neko のネットワークとして振る舞うことで解決する。

2.2 SSFNet とは

SSFNet[2] (図 2.5) とは、Java 言語で実装されたネットワークシミュレータで、離散的なイベント駆動のシミュレータで動作している。SSFNet は、ネットワークとインターネットプロトコルをシミュレーションするために SSF (Scalable Simulation Framework) ベースのモデルである。SSFNet は、Java 言語以外に C ＋＋言語でも実装さており、SSFNet

のパッケージは、大規模なネットワークモデルを構築するために、直接または、継承ができるクラスとして提供されている。そのSSFNetのライブラリーは、ネットワークの要素(ホスト、ルーター、ネットワーク、インタフェースカード、LAN）のためのコンポーネントモジュールとネットワークプロトコル（IP、UDP、TCP、BGP、OSPF、ICMP、HTTPなど）含んでいる。SSFNetはいくつか　大規模なモデルをサポートすることについてのいくつかの難しい方法を組み入れており、それらによって、非常に高いパフォーマンスを手に入れている。１つ目は、モデルの環境設定ための高度なサポートは、適切な結果を得るために重要なことである。２つ目は、大規模なネットワークモデルは、簡単モデルなどを高度に組み立てることにより、複数の要素による構成を用いた指向で最適に管理、構築されなければ成らない。３つ目は、高いシミュレーション性能を成し遂げることは、シミュレートされたコンポーネント間の相互作用を理解することと、SSFのパフォーマンスモデルに基づく集合的な計算結果と通信の要求を解釈することに深く依存する。

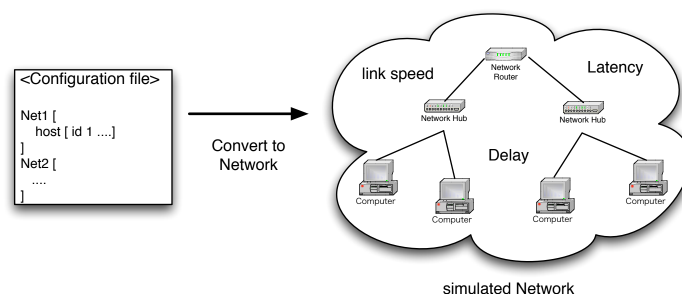

図 2.5: SSFNet

self-configuratble model SSFNetのモデルは、開発者自身が自由に構成することが可能である。つまり、各SSFNetクラスのインスタンスがクエリ形式のパラメータデータベースを用いることにより、それ自身を自動的に構成可能であるということである。コンフィグレーションのデータは、親のエンティティーにより、自分自身の構成情報の一部を子のエンティティーに渡すことで階層的に構築される。そのコンフィグレーションデータベースは、階層的なスキーマを持っている。スキーマの情報は、利用に耐えうる変数とデータベースから情報を検索するような機能を蓄えた入れ子状の基盤である。

シミュレートされた１００万のコンポーネントにおよぶ規模は、必要なコンフィグレーション管理ためにオブジェクトデータベース技術を用いている。計画方針を除いて、それは、大規模なモデルへのパラメータへ一貫性と効率を満たすことは極端に難しいものである。構築されたデータベースは、典型的なコンフィグレーションデータと主要なモデリングエラーの原因を除くことにも対応する。

Model composition from components

SSFのモデルは、各大規模なモデルに組み立てられたコンポーネントための木構造の

コンフィグレーションパラメータを区別するために、単純な階層的な属性の木構造による DML(Domain Modeling Language）表記法を用いている。

開発者は、基本的な定義のデータベースを簡単に参考にすることで、任意のネットワークコンフィグレーションをすばやく構成可能である。各ネットワークコンフィグレーションデータベースは、各コンポーネントとして実装された実行可能な C++また、Java を用いることにより価値ある成果になる。

Self-configurable models

DML の仕様は、1 組の key/value から成り立っている。value は、属性のリストから成っている。最上位レベルのノードは、Net で SSFNet ライブラリの Net クラスと一致する。Net は、自分自身で構成する。つまり、Net は特有のインタフェースを実装しており、クエリを用いてそれ自身のパラメタを獲得することデータベースとして機能する。

Net は、データベースを読み込み、そして、Roter と Host のインスタンスを生成する属性のリストを記憶する。また、Roter と Host も基本的に自分自身で SSFNet のクラスとして構成する。そして、NIC(ネットワークインタフェースカード) がそれらに適切に接続される。いったん構築されると、各インスタンスは、それらの特有のコンフィグレーションデータベースの一部の参照を受け取る。そして、また、順番に自身で構成することができる。この作業は、全体のモデルが周期的に構築、構成されるまで、子のエンティティーがさらに下の子のエンティティーを構成するのと同時に続けられる。

Network Protocol Graphs and Network Interface Models

Router と Host は、x-kernel のデザインパターンを用いて、特有のネットワークプロトコルのグラフによりパラメタ化される。各ホストはプロトコルグラフを含んでいる。プロトコルグラフは、ここのプロトコルセッションによって組み立てられ、アプリケーションのアクティビティー、または、その Host の NIC にパケットが到着し応答があったとき、上位のプロトコルにメッセージを渡りしたり、下位のプロトコルに渡したりする。図 2.7 は、図 2.6 におけるプロトコルスタックの論理的な構造を示したものである。

SSFNet は、新しい TCP の実装を含んでいる。それは、RFC の要求に基づいたすべての TCP の状態、フロー制御とネットワークの混雑をコントロールする技術である。また、それらは、slow start, Jacobson's RTT estimation, Karn/Partridge アルゴリズム, exponential retransmit timer backoff, adaptive acknowledgment、および fast retransmit を含んでいる。それらは、簡単に継承し、デザインされており変更している。例えば、send-window、receive-window、および incoming message demultiplexing は　、すべて明瞭なクラスとして提供されている。

SSFNet では、実世界のインターネットと同じ振る舞いとして、TCP は IP の上に働き、さらに、IP は、仮のプロトコルが表した構成された NIC の上で働く。各 NIC は、そとの世界による IP パケットのイベントを交換するための蓄積された SSF チャンネル　（Buffer）を維持している。NIC は、同じリンクタイプの違う NIC と接続される。そして、特徴的な物理的なリンク、パケットフォローのバッファリング、および flow interleaving と scheduling

のためのオプションを自信で構成することをサポートしている。

2.2.1 SSFNet のアーキテクチャ

SSFNet のアーキテクチャ(図 2.8）は、プロトコルレベルとネットワークレベルに分けることができる。プロトコルレベルでは、実際のプロトコルスタックを構成するように、プロトコルを実装することが可能である。プロトコルスタックの最上位層の部分は、実際に JAVA で実装するように、計算機同士でどのようなやり取りをするか (プロトコル) を開発者自身が実装する。最上位層以外の層は、DML を用いて SSFNet で用意されたクラスを用いて構成する。開発者は、自由にプロトコルを定義することが可能であり、SSFNet の API を用いることでプロトコルを実装することが可能である。

ネットワークレベルでは、DML を用いて現実的なネットワークモデルを定義する。DML では、ネットワークのトポロジ、リンク間の遅延、ネットワークカードの Latency および各計算機のバッファなどを詳細に定義することが可能である。DML 上で、各計算機へ振る舞いを実装したクラスを割り当てることで、各計算機の振る舞いを定義することができる。

```
Net [
    frequency 1000000000
    router [
        id 255
        graph [ProtocolSession [name ip use SSF.OS.IP]]
        interface [ id 0 buffer 16000 bitrate 100000000 latency 0.0001 ]
        ]
        host [ id 1
            nhi_route [dest default interface 0 next_hop 255(0)]
            interface [id 0 bitrate 100000000 latency 0.0001]
            graph [
            ProtocolSession [name Server use SSF.OS.TCP.Server port 10]
            ProtocolSession [name socket use SSF.OS.Socket.socketMaster]
            ProtocolSession [name tcp use SSF.OS.TCP.tcpSessionMaster]
            ProtocolSession [name udp use SSF.OS.UDP.udpSessionMaster]
            ProtocolSession [ name ip use SSF.OS.IP]
            ]
      ]
      host [ id 2
            nhi_route [dest default interface 0 next_hop 255(0)]
            interface [id 0 bitrate 100000000 latency 0.0001]
            graph [
            ProtocolSession [name client use SSF.OS.TCP.Client message-size 8]
            ProtocolSession [name socket use SSF.OS.Socket.socketMaster]
            ProtocolSession [name tcp use SSF.OS.TCP.tcpSessionMaster]
            ProtocolSession [name udp use SSF.OS.UDP.udpSessionMaster]
            ProtocolSession [ name ip use SSF.OS.IP]
            ]
      ]
      link [attach 255(0)
            attach 1(0)
            attach 2(0)
            delay 0.001
       ]
]
```

図 2.6: Sample SSFNet DML configuration file

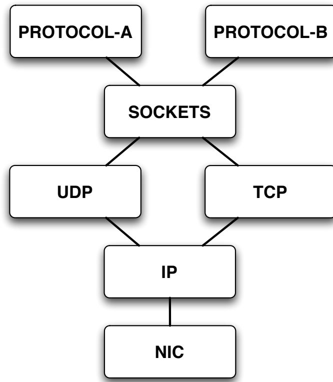

図 2.7: SSFNet protocol graph

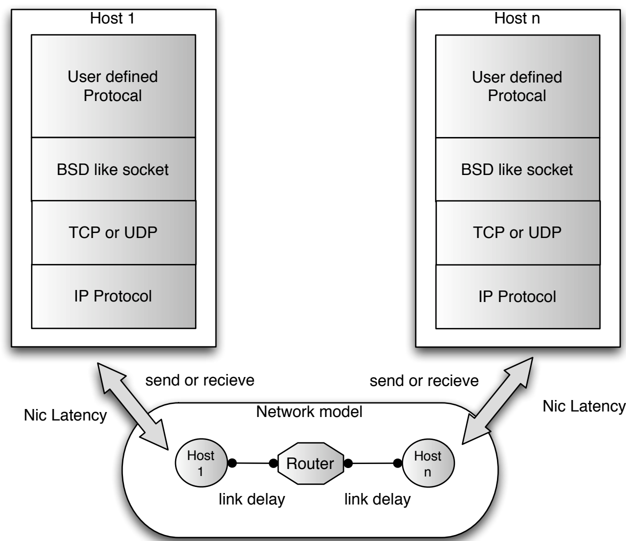

図 2.8: Architecture of SSFNet

第3章　統合方法

Nekoでは、Nekoの柔軟なインタフェースを利用して、開発者自身がモデル化したシミュレートネットワークのクラスをNekoへ実装し利用することが可能である。SSFNetでは、開発者がDMLファイルを用いてプロトコルスタックを自由に構成することができ、プロトコルスタックの最上位層に、開発者自身がプロトコルをモデル化し実装したクラスを利用可能である。本研究では、NekoとSSFNetを統合することが目的である。つまり、Nekoのシミュレートネットワークとして、SSFNetを用いることで、分散アルゴリズムを現実的なネットワークでシミュレートできるようすることである。具体的な方法は、NekoとSSFNetの間にインタフェースを作成することである。Neko側のインタフェースとして、SSFNetNetworkというシミュレートネットワークのクラスを作成する。SSFNet側のインタフェースとして、NekoProxyというSSFNetのプロトコルスタックの最上位層のクラスを作成する。SSFNetNetworkがNekoProxyを介して、SSFNetのシミュレーションネットワークを用いてメッセージを通信するために、SSFNetのTCPおよびUDPソケットインタフェースを利用できるようにする。これは、あたかも、Nekoの実ネットワークにおける振る舞いと非常に似ている。実ネットワークをNekoが利用する場合、Javaのソケットクラスを用いて、異なる計算機上のプロセスと通信を行い分散アルゴリズムを実行している。つまり、SSFNetのネットワークを実ネットワークとして利用していることである。実際、NekoProxyのクラスの中には、Nekoが実ネットワークを用いる場合に使われているクラスである、TCPNetworとUDPNetworkを継承しているクラスがある。

しかし、統合する上で下記のようないくつかのNekoとSSFNet間の特有の問題がある。

- 2つのイベント駆動のスケジューラの同期
- アドレスのマッピング方式
- メッセージ形式の違い
- コンフィグファイルの変更

この章では、次のような3つの統合する上で考慮した問題について解説していく。

- NekoとSSFNetの全体的な統合のデザインと、デザイン中のクラスとその役割によってわけられる個別の部分。
- 上記に示したNekoとSSFNet間の特有の問題の詳細と解決方法。

- Neko 側のインタフェースとして実装した SSFNetNetowork には、TCPNetwork と UDPNetwork の２つの異なった種類のシミュレートネットワークが存在する。２つの種類の SSFNetNetwork の利用方法と詳細。

3.1 統合デザインと実装上の問題の解決

SSFNet の全体的な統合のデザインと、実際に実装したクラス名を用いて、そのクラスの役割を説明し、Neko と SSFNet の統合方法について解説する。Neko と SSFNet を統合する上で下記のようないくつかの実装上の制限がある。

- NekoがSSFNetをコントロールする、つまり、Nekoのシュミレーション時間でSSFNetを駆動させる。
- Neko と SSFNet の両方とも、システムの根幹を成すスケジューラの様な中心となる実装をいっさい変更しない。つまりインタフェースとなる一部のパッケージにより統合する。これは、SSFNet のバージョンが上がるような大きな変更に耐えるためである。
- Neko のスケジューラと SSFNet のスケジューラを同期的に並行に動作させる。

基本的な設計指針は、Neko と SSFNet の両方にインタフェースとなるクラスを作成し、両方のシミュレータのスケジューラを並行に動作させながら、シミュレーションを行うことである。全体的な設計デザインは、図 3.1 のように表すことができる。メッセージがプロセス P_iから P_j へ伝播される例をもとに各クラスの役割について解説する。最上位で Neko プロセスが動作し、開発者が実装した分散アルゴリズムを実行している。Neko プロセス P_i は、Neko 上で Network Layer でシミュレートネットワークとして認識されている SSFNet Networks を介して、SSFNet 上の Host である NekoProxy にメッセージを伝播している。SSFNet Networks を用いて、SSFNet の Host である NekoProxy にメッセージを仲介して貰う時、Neko の ID と SSFNet の IP アドレスを 1 対 1 に対応させた Mapping Table を用いる。(**アドレス方式の違い**) NekoProxy は、あらかじめ作成された TCP か UDP のソケットを用いて通信する。しかし、この時、Neko 上と SSFNet 上の両方のスケジューラのシミュレーション時間の一貫性が取れていない。つまり、スケジューラの同期がここで問題になる。この問題の解決方法については、**スケジューラの同期**で詳しく述べる。そして、SSFNet のスケジューラでメッセージの伝播をシミュレートし、SSFNet によってメッセージが他の NekoProxy に送信される。メッセージが受信されると、Neko が NekoProxy からメッセージを取得し、SSFNet Networks を用いて Neko プロセス $_j$ へメッセージを伝播しようと試みる。しかし、この時、分散アルゴリズムのメッセージとして実装されているクラスである Neko メッセージは、複数の宛先 ID を保持している。送信時には、この宛先 ID を SSFNet の IP アドレスと対応させて送信することは可能である。しかし、SSFNet Networks を用いてメッセージを宛先の Neko プロセスは伝搬するとき、そのメッセージは

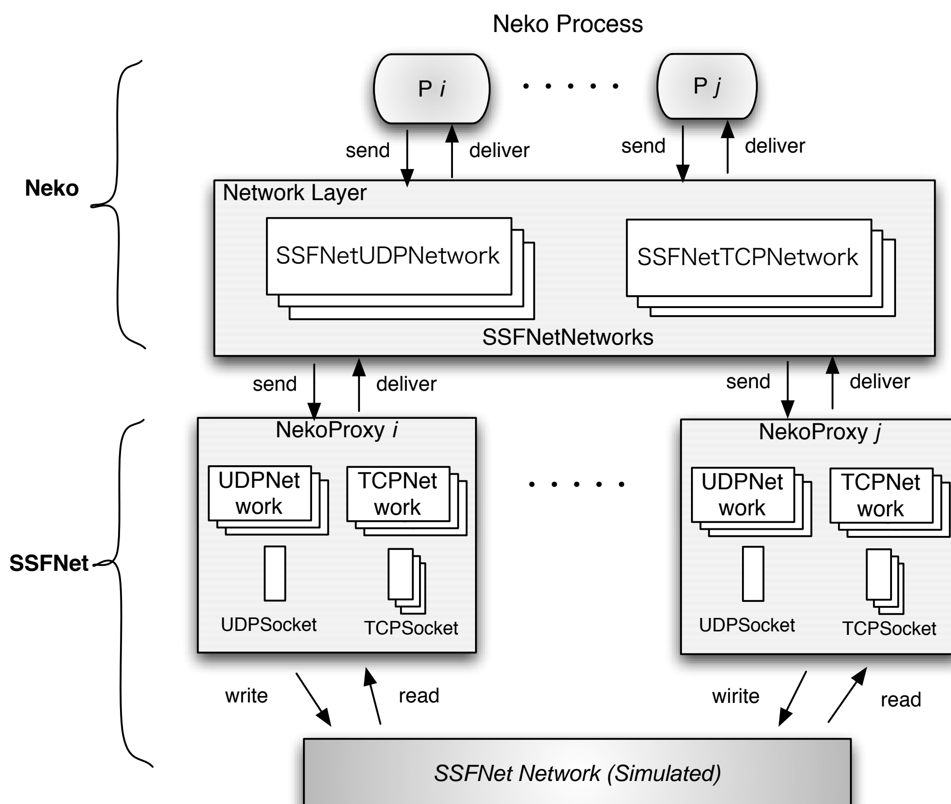

図 3.1: Integration Design

：

一意指定された Neko プロセスへのメッセージである。つまり、複数の宛先を含んでいると、そのメッセージを一意指定された Neko プロセスへ伝播することができない。この問題の解決方法は、**ネットワークアドレスの違い**と**メッセージ形式の違い**で詳しく述べる。最後に、Neko プロセス P_j は、自分自身に伝搬されたメッセージを受信し、そのメッセージにより分散アルゴリズムを実行していく。

このような順序で、メッセージが SSFNet をシミュレートネットワークとして伝播していき分散アルゴリズムが展開されていく。Neko 側の役割は、Neko プロセスからのメッセージを SSFNet へ渡すことで、これは、send イベントを生成していることに対応する。SSFNet 側の役割は、Neko からの送信イベントを処理し、そのイベントに関するメッセージを Neko へ渡すことであり、これは、deliver イベントを生成していることに対応する。つまり、シミュレートネットワークとして SSFNet を用いる場合、Neko 側の役割は、send イベントを生成することで、SSFNet 側の役割は、deliver イベントを生成することである。２つのスケジューラを用いて、この２つのイベントが一貫性を持って処理されるように、つまり、メッセージの伝播が適切に行われるように Neko へ SSFNet の統合を行った。

スケジューラの同期

2つの離散的なイベント駆動のシミュレータをイベント発生におけるシミュレーション時間の一貫性を持って、並行に動作させることは容易ではない。2つのスケジューラを単純に動作させた場合、どちらかのスケジューラがシミュレーション時間上、先に進んでいるか、もしくは、まだその時間に達していないので、メッセージを各プロセスへ伝播できない。つまり、これは、まったくシミュレータとして機能していないことに相当する。

この問題は、Neko 側の send イベントと SSFNet 側の deliver イベントを一貫性を持って、処理できるように、スケジューラを下記のようなルールに従って、交互に動作させることで解決した。

- Neko のスケジューラは、send イベントをその send イベントが起こる時間付きで、SSFNet へ渡し（SSFNet のソケットを用いてメッセージを SSFNet へ渡していること）停止した後、SSFNet のスケジューラを動作させる。
- また、Neko のスケジューラは、SSFNet からの deliver イベントを実行したら（Neko プロセスへ目セージを伝播すること）停止し、SSFNet のスケジューラを動作させる。
- SSFNet のスケジューラは、Neko から渡された send イベントをいっしょに渡された時間に従って、実行したら（メッセージの伝播を SSFNet でシミュレートし deliver イベントを生成）delliver イベントか、Neko スケジューラ中の次のイベントの時間まで動作する。deliver イベントの場合は、そのイベントを Neko へそのイベントが起こる時間付きで渡し停止した後、Neko のスケジューラを動作させる。

この同期方法を単純に説明すると、Neko の次のイベントのシミュレーション時間まで SSFNe のスケジューラを動作させる機会を与えることである。このルールに基づいてスケジューラを動作させることで、シミュレーション時間上の一貫性を持って、メッセージの伝播を処理することが可能となる。

アドレス方式の違い

NEKO では、プロセスを識別するためにシーケンシャルな ID を用いている。一方、SSFNet では、計算機の識別に IP アドレスを用いている。NEKO では、Neko メッセージに含まれている宛先 ID を用いて、メッセージの伝播を行っている。しかし、IP アドレスを指定して送信できないので SSFNet を利用して、他の NEKO プロセスに送信することができない。NEKO の ID と SSFNet の IP アドレスを 1 対 1 で対応させるマッピングテーブルクラスを作成した。NEKO からの送信時と SSFNet からの受信時にこのマッピングテーブルを用いることで、アドレス方式の違いを解決した。

メッセージ形式の違い

NEKO のメッセージは、1 つ以上の宛先 ID と文字列をメッセージとして送信する。一方、SSFNet は、送信時にメッセージサイズを指定してオブジェクトを送信する。よって、

SSFNetでは、NEKOメッセージをオブジェクトとして、変更を加えることなく、送信することが可能である。しかし、通常、NEKOではメッセージを送信する場合、NEKOメッセージの複数の宛先IDを指定してメッセージの送受信を行っている。SSFNetのソケットを用いて、Nekoメッセージを送信する場合は、マッピングテーブルを用いて従来の送信方法を用いて、IPアドレスを指定し、そのメッセージを宛先のSSFNet上のHostへ送信可能である。しかし、SSFNetNetworksを用いて、NekoメッセージをNekoプロセスに渡す場合、そのメッセージは、一意に指定したプロセスへの伝播されなければならない。しかし、Nekoメッセージの宛先IDを用いると、その宛先IDに対応したすべてのNekoプロセスへ送信してしまうことになり、不要なメッセージの複製を行ってしまっている。

この問題を解決するために、NEKOプロセスを一意に識別できる情報を付加したSSFNetメッセージクラスを作成した。SSFNetメッセージは、SSFNetのソケットを用いてメッセージを伝播する時に、宛先のHostのIPアドレスを用いて割り出した（マッピングテーブルより）NekoプロセスのIDを付加する。Nekoは、SSFNet Networksを用いてNekoプロセスにメッセージを渡す時に、Nekoメッセージとして伝播されてきたSSFNetメッセージから、最終的な宛先であるNekoプロセスのIDを用いて、そのメッセージを一意に指定されたNekoプロセスへ渡す。

3.1.1 SSFNet Networks

SSFNetをNekoのシミュレートネットワークとして用いるためのクラスであるSSFNet-Networks（図3.1）について詳しく説明する。シミュレートネットワークとしてSSFNetを指定したい場合、Nekoのコンフィグファイル上に、あるSSFNetNetworksの種類のクラスをパラメタとして指定することで利用可能である。

SSFNetNetworksは、TCPとUDPの２つの種類のクラスに分類している。TCPクラスのネットワークは、SSFNetTCPNetworkと呼ばれ、メッセージを伝播するためのプロトコルとして信頼性の高いTCP/IPを用いている。ネットワークシミュレータとして、SSFNetTCPNetworkが指定されると、Nekoの初期化時に、SSFNet上の各Hostがプロセス数に応じて、受信用と送信用のTCPソケット生成し、各Host同士がメッセージを伝播するためのTCPコネクションを張る。

一方、UDPクラスのネットワークは、SSFNetUDPNetworkと呼ばれ、メッセージを伝播するためのプロトコルとして、信頼性の低いUDP/IPを用いている。このネットワークは、Nekoの初期化時に、１つだけUDPソケットを生成する。

これらのネットワーククラスは、Nekoのコンフィグファイルを用いて、複数指定するこができる。例えば、TCPのクラスを２つ選択した場合は、各Host間に２組の送受信用のソケットが生成され、コネクションが張られる。UDPのクラスを選択した場合も同様である。また、TCPとUDPのクラスを混合して、送信することが可能であり、開発者は、ネットワークモデルに応じて複数のソケットを利用することができる。

3.2 動作テスト

本項では、前述の方法を用いて、統合を行ったNekoの動作テストを行う。シミュイレーションを行うアルゴリズムは、アトミックブロードキャストの1つであるLamport[9]のアルゴリズムを用いる。このアルゴリズムは、イベント数がプロセス数nに対して、$O(n^2)$の計算量で増加し、同一の時間に複数のイベントを処理する。また、すでにNekoを用いてクラスとして実装されており、テストケースとして適している。既存のNekoに実装されているシミュレーションネットワークであるBasicNetworkおよびMetricNetworkと、本研究で実装したネットワークであるSSFNetTCPNetworkおよびSSFNetUDPNetworkを用いてテストを行う。計測する対象は、既存のNeko単体とSSFNetを用いた場合のシミュレーションにおける実時間の比較と、SSFNetを用いた場合、最大どれくらいのプロセス数をシミュレート可能であるかの2つである。実時間の比較方法は、プロセス数が10で、Lamportのアルゴリズムを1000回連続で実行した時の実時間を計測する。最大プロセス数の計測は、プロセス数の値を変更しながら、Lamportのアルゴリズムを1回だけ実行し、いくらのプロセス数で実行できないかの境界を計測する。評価環境は、Cpu clockが2.6GHz、Memoryが約1GByteである。経過時間の計測には、Unixのコマンドであるtimeコマンドを用いる。

	Simulated Network class	execution time
Neko	BasicNetwork	5.0 [sec]
	MetricNework	65.5 [sec]
SSFNet	SSFNetUDPNetwork	63.5 [sec]
	SSFNetTCPNetwork	126.2 [sec]

表 3.1: Unit test

NekoのBasicNetworkとSSFNetを用いた場合を比較すると、実時間レベルで最大13倍長い。これ2つの理由がある。1つ目は、NekoのスケジューラのスレッドとSSFNetのスケジューラを交互に動作させているために、スレッドを切り替えるオーバーヘッドが生じている。2つ目は、SSFNetは、ネットワークを離散的なイベント駆動でシミュレートしているため、Nekoのネットワークシミュレータよりもシミュレーションに多くの時間を有するためにである。NekoのMetricNetworkとSSFNetのネットワークを比較すると、実時間レベルで最大2倍長い。MetricNetworkは、BasicNetworkと比べ複雑なシミュレートを行っており、SSFNetUDPNetworkとほぼ同じ経過時間でシミュレート可能なことがわかる。SSFNetを用いた場合の最大プロセス数は、Neko単体と比較して、かなり制限があることがわかる。SSFNetUDPNetworkを用いた場合のシミュレート可能な最大プロセス数は253である。この理由は、Nekoの実ネットワーク環境で実験を行う場合に用いられているUDPNetwokクラスを継承して用いているからである。実ネットワークで実験を行う時にUDPNetworkクラスは、コンフィグレーションファイルの実IPアドレスを利用

している。SSFNet では、IP アドレスとして、実際に IPv4 など用いられているアドレスのフォーマットを用いておらず、順番に 1 から番号付けしているために制限が起きてしまう。また、SSFNetTPCNetwork を用いた場合は、シミュレート可能な最大プロセス数は 53 である。この理由は、スレッドを多く生成しすぎて、Java の実行環境 “Memory out of errer” により停止してしますためである。SSFNetTCPNetwor では、各プロセスが送信用と受信用のソケットをプロセス数だけ生成する。そのソケットの数だけ、スレッドを用いて送信と受信を行っているので、プロセス数が増える度にスレッドを生成してしまう。

SSFNetUDPNetwork の問題は、Neko 上の UDPNetwork クラスをリファクタリングを行うことで、シミュレート可能なプロセス数を増やすことは可能あると考える。SSFNetTCPNetwork の問題は、JAVA の実行環境の問題か OS のスレッド生成の上限に関する問題かのいずれかであると考えられる。この問題についての解決方法は、現在調査中である。

第4章 How to use Neko with SSFNet

本章では、ネットワークシミュレーション環境としてSSFNetを用いてシミュレーションを行う方法を解説する。はじめに、Neko上でSSFNetを利用する場合のインストール方法について解説する。次に、サンプルアルゴリズムを用いて、NekoとSSFNetのコンフィグレーションファイルの設定方法とおよび実行方法について解説する。

4.1 インストール方法

はじめに、SSFNetとNekoをダウンロードしなければならない。SSFNetはライセンス規約により、自由に配布できないので、ユーザ自身がhttp://www.ssfnet.org/より、ダウンロードしなければならない。Nekoは、http://lsrwww.epfl.ch/neko/ から取得できる。次にSSFNetを任意のディレクトリにインストールする。インストール方法は、SSFNetのwebページに記載されているので、ここでは省略する。つぎに、Nekoを任意のディレクトリに展開する。nekoというが生成されるので、そのディレクトリは移動して、下記のようにコマンドを入力する。

```
例:SSFNetがインストールされたディレクトリ （/home/***/work/ssfnet）

$ ./configuer –java='Javaコマンドのパス' –ssfnet='/home/***/work/ssfnet/lib'
$ ../dest_neko/bin/ant
```

上記のような手順で、インストールすることにより、SSFNetのクラスパスがNekoのコマンド群に設定される。[1]

[1] Nekoのコマンド群を使うために、パス変数にNekoがインストールされたディレクトリを設定しなければならない (詳細は http://lsrwww.epfl.ch/neko/)

4.2　コンフィグファイルの設定

シュミレーションを実行するためのコンフィグファイルの設定方法ついて解説する。基本的には、ネットワークシミュレータとしてSSFNetを用いる場合、従来のNekoのコンフィグファイルにSSFNet特有のパラメタを追加するだけで実行可能である。SSFNet特有のパレメタとは、SSFNetの基本となるDMLファイル、開発者がネットワークモデルを記述したDMLファイル、メッセージサイズおよびネットワークモデルのHost数である。まずはじめに、Nekoのコンフィグファイルの記述方法を解説する。つぎに、SSFNetのコンフィグファイルであるDMLの記述方法とサンプルネットワークモデルの解説をする。シミュレーションを行うアルゴリズムは、すでにNeko上に実装されているLamportのAtomic Broadcastを用いる。

Nekoコンフィグファイルの設定

Nekoのコンフィグファイルの記述方法は、'パラメタ名 = 値 or 文字列　or 真偽値'の単純な形式である。開発者は、自分自身で実装したアルゴリズムに、このコンフィグファイルからパラメータを設定することも可能であり、自由なパラメタが設定可能ある。Nekoでは、コンフィグファイルからパラメタを取得する方法として、Apacheのorg.java.utilパッケージのconfigurationsクラスを利用しており、パラメタ名を複数指定することで配列としてパラメタを取得可能である。

図4.1を参照しながら各パラメタについて説明する。一行目のsimulationは、falseの場合は、分散アルゴリズムを実行する環境として、実ネットワークを用いる。実ネットワークを指定した場合、プロセスが実行している計算機のIPアドレスやポート番号などを記述しなければならない。本研究では、実ネットワーク特有のパラメタについては解説しない。５行目のprocess.numは、アルゴリズムを実行するプロセスの数である。この値を変更することにより、実行するプロセスの数を変更することが可能だが、シミュレーションネットワークとしてSSFNetを用いる場合、後に説明するhost.numの値以下でなくてはならない。なぜなら、SSFNetでは、DMLファイルにより、ネットワーク上で利用できる計算機の数が記述されており、その数以上のプロセスは、SSFNetの計算機に配置できないからである。７行目のhost.numは、SSFNetのシミュレーションネットワーク上にシミュレートされる計算機の数である。Nekoのプロセスは、SSFNetの計算機と1対1に対応しており、その計算機上でNekoのプロセスが実行されることになる。８行目のalgorithmは、プロセス上で実行される分散アルゴリズムのクラス名である。９行目のprocess.initializerは、Nekoのプロセスの振る舞いであるアルゴリズムを階層的に構築するためのクラス名である。このクラスがalgorithmで指定されたアルゴリズムのクラス名を動的に読み込み、階層的にアルゴリズムを構築する。10行目のnetworkは、シミュレーションネットワークのクラス名である。このパラメタを変更することにより、シミュレーションネットワークの変更が可能である。このサンプルコンフィグファイルでは、SSFNetUDPNetworkを指定しているが、他のネットワークや、複数のネットワークを指定することが可能である。この値を用いて、ネットワークの切り替えを簡単に可能にして

いる。１２行目からは、シミュレーションネットワークとして SSFNet を用いた場合の固有のパラメタである。１３行目の message.size は、Neko のプロセスが通信に用いる１つあたりのメッセージのサイズであり、単位は byte である。１４行目の my.dml は、開発者が記述した DML ファイルの絶対パスである。DML ファイルは、複数のファイルから構成することができるため、複数の DML ファイルを指定したい場合は、複数の my.dml パラメタを用いてすべての DML ファイルの絶対パスを記述しなければならない。１６行目の ssfnet.dml は、SSFNet が記述した DML ファイルの基本となる構成を記述したファイルである。このファイルは、SSFNet 単体で実行させる場合でもかならず指定しなければならない。１８行目の direct.mapping は、SSFNet のある計算機に対して、Neko のプロセスを指定して対応させたいときに用いる。パラメタに true の値を指定すると、２０行目から２６行目のような ssfnet.process を複数記述して、Neko のプロセスの ID と SSFNet の NHI アドレスを用いて記述する。記述形式は、'Neko のプロセス ID / SSFNet の NHI アドレス' である。

以上が、Neko のコンフィグファイルの記述方法と各パラメタについての説明である。このサンプルコンフィグファイルは、必要最低限の実行に必要な設定である。この他のパラメタとして、log4j を用いたログの取得方法やその出力ファイル名の指定など、さまざまなパラメタが存在する。さらに詳細な情報を知りたい場合は、インストールの章で紹介した Neko の web ページで調べてほしい。

SSFNet　DML ファイルの設定

SSFNet のコンフィグファイルである DML ファイルの記述方法とその各パラメタについて説明する。サンプルの DML ファイルで記述されたネットワークモデルのトポロジと各パラメタは、図 4.2 のように表すことができる。単純な Local Area Network をモデリングしており、１つのルータに対して、１つのハブを中継して７つの計算機が接続されている。各計算機とルータに ID 番号が割り振られているが、これは IP アドレスではなく、SSFNet 固有の NHI アドレスである。すべてのリンクスピード、リンク間のパケット遅延および NIC(Network Interface Card) 上の１つのパケットに対する遅延であるレイテンシは、同じ値にモデリングしており対象である。各計算機上にある NIC のレイテンシーは、１パケットごとの送信遅延を表現しており 100ms であり、NIC の Buffer は、8000byte である。ルーターのインタフェースのレイテンシーも 100ms であり、Buffer は、16000byte である。ハブは、SSFNet ではシミュレートされていないが、複数の計算機をルータに接続する場合において、DML ファイルの属性としてハブを用いることにより簡単にルータに各計算機を接続する状態を記述可能となる。もし、このハブを用いないとルーターは、接続される各ホストの数だけインタフェースを用意しなければならないし、それらが接続されていることを DML で記述しなければならないため、多くの手間がかかる。

サムプル DML ファイル (図 4.3 および図 4.4) について説明する。この DML ファイルは、Neko のコンフィグファイルの１４行目の my.dml で指定している ` evaluation-net-p7.dml ' のファイルに記述されている。図 4.3 は、主にネットワークのトポロジーを記述しており、図 4.4 は、ネットワークで用いられる各属性の値を記述している。

```
1: simulation = true
2: #
3: # The number of communicating processes.
4: #
5: process.num = 5
6: # The number of Host on DML (Network model)
7: host.num = 7
8: algorithm = lse.neko.examples.basic.Lamport
9: process.initializer = lse.neko.examples.basic.TestInitializer
10: network = lse.neko.sim.ssfnet.udp.SSFNetUDPNetwork
11: #network = lse.neko.sim.ssfnet.tcp.SSFNetTCPNetwork
12: #User defined parameter on SSFNet's DML file
13: message.size = 200
14: my.dml = /home/m-tomo/work/nekoTest/finalTest/evaluation-net-p7.dml
15: #ssfnet dml
16: ssfnet.dml = /home/m-tomo/usr/ssfnet/examples/net.dml
17: # correspondent Neko process to SSFNet Host.
18: direct.mapping = false
19: #direct.mapping = true
20: ssfnet.process = 0/7
21: ssfnet.process = 1/6
22: ssfnet.process = 2/5
23: ssfnet.process = 3/4
24: ssfnet.process = 5/3
25: ssfnet.process = 6/2
26: ssfnet.process = 7/1
```

図 4.1: Sample Neko configuration file

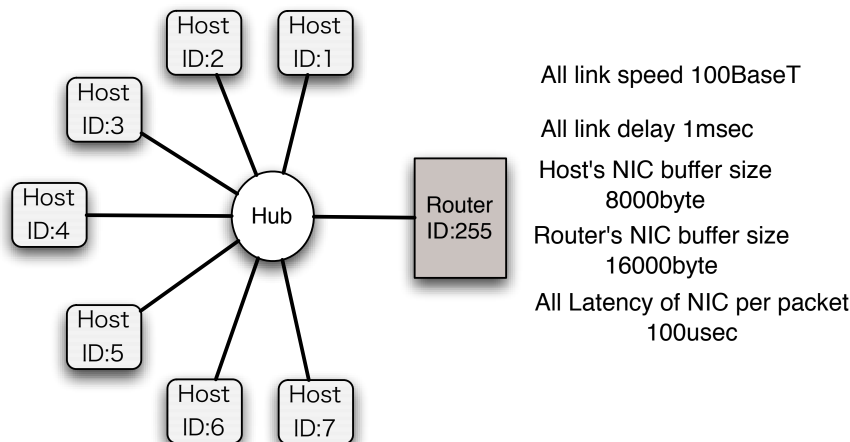

図 4.2: Network model of sample SSFNet DML file

はじめに、図 4.3 から説明する。一行目の Net は、Net の中に Net を記述することも可能であり、実ネットワークの世界でいうサブネットワークを定義している。２行目の frequency は、最上位の Net に記述することが可能であり、最小のシミュレーション時間の単位を定義している。つまり、frequency は、1GHz なので 1ns である。この値は、各インタフェースの bitrate と関係しており、もし、bitrate が 100,000,000（100MbaseT）ならば、100msec 以上でなければならない。よって、この frequency は、1GbaseT のリンクスピードまでシミュレート可能である。３行目の router は、TCP/IP 参照モデルのネットワーク層に分類されるルータと呼ばれる機器を定義している。NHI アドレスとして、255 番の ID 番号を割り振っており、graph によりプロトコルとして IP を用いている。もし、LAN 以上の大きなネットワークモデルを記述する場合は、graph に OSPF や BGP のプロトコルを IP プロトコルの上位に記述しなければならない。interface は、計算機が外部つまり、ネットワークにパケットを送信するためのインタフェースを記述する属性である。すべての計算機 (Host,router,etc) は、graph と interface を持たなければならない。interface にも、NHI アドレスが割り振られており、このインタフェースの NHI アドレスは｀255:0’となり、このアドレスを用いて、他の機器は接続する。_extend は、他の DML ファイル名か同一ファイル内の他のトップレベルの属性の下位の属性のパラメタをすべての参照することができる。つまり、図 4.4 の dictionary の 100BaseT の２つの値を参照していることになる。９行目の host は、計算機を定義しており、idrange を用いることにより、一度に複数の host を定義することが可能である。この場合は、ID1 から ID7 までを指定しているので、合計７つの計算機を定義している。nhi_route は、ハブを中継して計算機（このモデルの場合、ルーター）に接続すること定義している。つまり、このように定義することにより、簡単にルータに接続している状態を定義可能ある。この host も router と同様に、dictionary の process 属性の値をすべてを参照している。interface について記述されおり、さらに、host の graph は、複数のプロトコルから記述されている。graph は、そ

の計算機のプトロコルスタックを定義している。計算機のプロトコルスタックの構成と各属性については、後の図4.4の説明時に詳細に解説する。１３行目のlinkは、TCP/IP参照モデルのデータリンク層を定義しており、NHIアドレスを用いて各計算機とルータの接続を定義してる。_findを用いて、１つだけ他の属性を参照することが可能であり、リンク間の遅延であるdelayを参照している。以上がネットワークのトポロジの部分についての説明と記述方法である。

```
1: Net [
2:        frequency 1000000000
3:        router [
4:              id 255
5:                   graph [ProtocolSession [name ip use SSF.OS.IP]]
6:                  interface [ id 0 buffer 16000 _extends .dictionary.100BaseT ]
7:        ]
8:        ####all latency of nic is 0.0001 [sec]####
9:        host [ idrange [from 1 to 7]
10:             nhi_route [dest default interface 0 next_hop 255(0)]
11:             _extends .dictionary.process
12:       ]
13:       link [attach 255(0)
14:             attach 1(0)
15:             attach 2(0)
16:             attach 3(0)
17:             attach 4(0)
18:             attach 5(0)
19:             attach 6(0)
20:             attach 7(0)
21:             _find .dictionary.link_delay.delay
22:       ]
23: ]
```

図 4.3: Sample SSFNet DML file　(part of Net)

次に、ネットワークの各パラメータが定義されたdictionary（図4.4）について説明する。3行目で参照されてある１２行目のNekoProxy_standard.graphで定義されてあるプロトコルスタックについて説明する。最上位のProtocolSessionクラスつまりプロトコルは、本研究で作成したNekoとSSFNetのインタフェースであるNekoProxyクラスを指定している。DMLファイル中の計算機のプロトコルスタックを構築する定義するときに、最上位層にNekoProxyクラスを指定し、NekoのコンフィグファイルのnetworkにSSFNetNetworkクラスを指定するだけで、Nekoのシミュレーションネットワークとして、SSFNetを用いることができる。次の下位層であるsocketMasterクラスは、ソケットインタフェースをシミュレートするクラスであり、下位層のプロトコルであるtcpSeeionMasterクラスかudpSesseionMasterクラスを用いて、TCPソケット、UDPソケットを生成する。最下位層のIPクラスは、TCP/IP参照モデルのトランスポート層をシミュレートしている。このgraphつまりプロトコルスタックは、SSFNetをNekoのシミュレートネットワークとして用いる場合は、常に同様の定義である。従って、計算機のプロトコルスタックを定義する場合は、必ずこの記述を用いらなければならない。７行目から１１行目は、各計算機のインタフェースの仕様を定義したもので、例として記述している。２８行目のtcpinit-default-ssfnetは、TCPソケットを用いる場合のTCPプロトコルの仕様を定義したものである。ここに定義した仕様は、SSFNetがデフォルトで定義してある仕様をそのまま用いた。開発者は、自分自身で定義したネットワークモデルに従って、この仕様を変更することができる。最後に４６行目のupdinitは、UDPプロトコルの仕様を定義しており、変更できるパラメタとして、一度に送信可能なパケットサイズを定義できる。

4.3 実行

設定したコンフィグファイルを用いて、シミュレーションする方法とその出力結果について説明する。シミュレートするアルゴリズムは、LamportのAtomic Broadcastで、シミュレートネットワークとしてUDPソケットを用いて通信を行うSSFNetUDPNetworkを用いる。Nekoのコンフィグファイル名は、sample.configとする。ターミナルで下記のように、nekoコマンド[2]にNekoのコンフィグファイルを引数として実行する。

```
$ neko sample.config
```

SSFNetがネットワークモデル初期かし、そのネットワークモデルの情報を標準出力した後、Nekoのシミュレーションが開始される。そして、各プロセスがアルゴリズムを終了したことを出力し、最後にアルゴリズムによって、経過した時間を出力する。出力結果は下記の通りである。

[2] dest_neko/binにパスが通していること

```
SSFNet の初期化情報が出力される
.
.
p0 finished
p1 finished
p2 finished
p4 finished
finishing at 2.3568
```

通常は、この出力結果を性能評価に用いるわけではない。Neko のコンフィグファイルに、メッセージの通信履歴を取るために log4j クラスを用いるように設定する。もしくは、アルゴリズム中に異なった形式をファイルに出力するように実装することもできる。開発者は、いずれの場合も、その通信履歴を用いてグラフ化などを行い性能評価に利用する。この章では、SSFNet を Neko のシミュレートネットワークとして用いるために、コンフィグファイルの設定方法と実行方法および実行結果について説明した。もっと詳細な方法を知りたければ、インストールの説明時に紹介した Neko と SSFNet の web サイトで入手可能である。

```
1: dictionary [
2:          process [
3:          interface [id 0 buffer 8000 _extends .dictionary.100BaseT]
4:          _find .dictionary.NekoProxy_standard.graph
5:          ]
6:          # standard network interface
7:          10BaseT [bitrate 10000000  _find  .dictionary.nic_latency.latency]
8:          100BaseT [bitrate 100000000 _find  .dictionary.nic_latency.latency
]
9:          1GBaseT [bitrate 1000000000 _find .dictionary.nic_latency.latency ]
10:        nic_latency [latency 0.0001]
11:        link_delay [delay 0.001]
12:        NekoProxy_standard [graph [
13:              ProtocolSession [
14:                    name NekoProxy use lse.neko.sim.ssfnet.NekoProxy
15:
16:              ]
17:              ProtocolSession [name socket use SSF.OS.Socket.socketMaster]
18:              ProtocolSession [
19:                    name tcp use SSF.OS.TCP.tcpSessionMaster
20:                    #_extends .dictionary.tcpinit-default-ssfnet
21:              ]
22:              ProtocolSession [
23:                    name udp use SSF.OS.UDP.udpSessionMaster
24:                    #_extends .dictionary.udpinit
25:               ]
26:               ProtocolSession [ name ip use SSF.OS.IP]
27:        ]]
28:        tcpinit-default-ssfnet[
29:               ISS 0       # initial sequence number
30:               MSS 1024               # maximum segment size
31:               RcvWndSize  16          # receive buffer size in units of MSS
32:               SendWndSize 16        # maximum send window size in units of MSS
33:               SendBufferSize 16     # send buffer size in units of MSS
34:               MaxConWnd 64          #
35:               MaxRexmitTimes 12     # maximum number of retransmission times
36:               TCP_SLOW_INTERVAL 0.5 # granularity of TCP slow timer, sec
37:               TCP_FAST_INTERVAL 0.2 # granularity of TCP fast(delay-ack), sec
38:               MSL 3                 # maximum segment lifetime, sec
39:               MaxIdleTime 600       # maximum idle time, sec
40:               delayed_ack true      # delayed ack option, true or false
42:               fast_recovery 60      # use fast recovery algorithm, true/false
43:               show_report false     # print summary connection report, true/false
44:               debug    false
                                     # if true, dump verbose TCP diagnostics to files
45:        ]
46:        udpinit [
47:               max_datagram_size 10000 # max UDP datagram size (payload bytes)
48:               debug false             # print verbose UDP diagnostics
49:        ]
50: ]
```


図 4.4: Sample SSFNet DML file　(part of dictionary)

第5章　性能評価

本研究でSSFNetと統合したNekoを用いて、分散アルゴリズムの1つであるAtomic Broadcastの性能評価を行う。Atomic Broadcastについては、後に詳細を説明する。性能評価を行う目的は、複雑な分散アルゴリズムをシミュレートネットワークとしてSSFNetを選択した場合、つまり、本研究で実装し統合したNeko上でシミュレートが可能なことを示すことで、その統合されたNekoの有効性を示すことである。また、その有効性を示しつつ、Atomic Broadcastの性能評価も同時に行う。本研究で性能評価を行うAtomic Broadcastは、すでにNeko上で実装されているアルゴリズムである。Urbanら[5]によって行われた評価結果を参考にし、Atomic Broadcastに必要不可欠な分散アルゴリズムであるConsensusの異なった２種類の性能評価を行う。２つの異なったアルゴリズムの特徴を浮かび上がらせることで、システムの設計者により、システムに適切なアルゴリズムの選択を提供する判断材料とさせる。しかし、分散アルゴリズムをこのように評価するためには、計算機とネットワークのリソースを考慮にいれなければならない。従って、シミュレートネットワークとしてSSFNetを用いることで、より現実的な性能評価結果が得られる。つまり、システム設計者は、より適切な評価に基づいた判断ができる。

5.1　Atomic Broadcastとは

Atomic Broadcast(Total Order Broadcastとも呼ばれる)は、分散システムにおいて基本となる問題である。Atomic Broadcastは、すべてのメッセージを同じ順序（全順序）で届けるプリミテブなBroad castとして定義される。例えば、Process Replication（プロセスの複製技術)、複製かされたデータベースのトランザクション処理をサポートするために用いられる。Atomic Broadcastといっても、約６０以上のアルゴリズムが存在している。Atomic Broadcastの基本的な定義と５つのクラスに分類したAtomic Broadcastの特徴について説明していく。

5.1.1　Atomic Broadcastの定義

Atomic Broadcastは、２つのプリミティブな操作から成り立っている。(mはメッセージ)

- **A-broadcast(m)**
- **A-deliver(m)**

さらに、前提条件２つがあり、すべてのメッセージは、一意に識別可能であることと、送信プロセスのよって与えられたその識別子はメッセージといっしょに含まれて運ばれることである。Atomic Broadcast を定義する特性について説明する。ここでいう正確なプロセスとは、クラッシュなどの理由により、まちがった内容のメッセージを送信することや、メッセージの処理が極端に遅いプロセスではなく、正常に期待した振る舞いを行うプロセスのことである。

VALIDITY : 正確性 正確なプロセスがメッセージ m を A-broadcast する場合、最終的にメッセージ m を A-deliver することである。

AGREEMENT : 同意 正確なプロセスがメッセージ m を A-deliver する場合、最終的には、すべての正確なプロセスはメッセージ m を A-deliver する

INTEGRITY : 完全 任意のメッセージ m のために、すべてのプロセスはメッセージ m を多くて 1 度しか A-broadcast しない。これは、メッセージ m が前もって、送信プロセスによりメッセージ m を A-broadcast されている場合に限る

TOTAL ORDER : 全順序 正確なプロセス p と q がメッセージ m とメッセージ m' を A-deliver した場合、p はメッセージ m' の前にメッセージ m を A-deliver する。これは、q がメッセージ m' の前にメッセージ m を A-deliver する場合に限る。

AGREEMENT と TOTAL OEDER にさらに Uniform 特性を加えることにより、Uniform Atomic Broadcast を定義することが可能である。

UNIFORM AGREEMENT 正確なプロセスがメッセージ m を A-deliver する場合、すべての正確なプロセスは、最終的にメッセージ m を A-deliver する。

UNIFORM TOTAL ORDER プロセス p とプロセス q（正確または、不正確なプロセス）が、メッセージ m とメッセージ m' を A-deliver する場合、プロセス p はメッセージ m を A-deliver する。これは、プロセス q がメッセージ m' を A-deliver する場合に限る。

Uniform 特性は、ただひとつのプロセスでも A-broadcast または、A-deliver できない場合は、すべてのプロセスも A-broadcast または、A-deliver しないということである。Unifrom 特性は、Atomic commit または、Active Replication の用いるような特定のアプリケーションに必要とされている。例えば、複製されたプロセスがある副作用を伴うような可能性があるシステムなどに利用される。それらのシステムは、異なったプロセスからの矛盾するような振る舞いが生じる Atomic Broadcast を用いている。しかし、Uniform 特性は、すべてのアプリケーションで必要としないし、残念なことに大幅なコストがかかる。システム設計者は、Uniform または、non-Uniform な Atomic Broadcast を区別して考慮しなければならない。

5.1.2 Atomic Broadcast アルゴリズム

Atomic Braodcast を約６０のアルゴリズムを５つのクラスに分類し、５つのクラスについて説明する。異なる３つのタイプの１つのプロセス（sender、destinations、sequencer）によって、生成されたメッセージの順番によって分類される。その他の要因で、３つのタイプから、下記のような５つのクラスに分類することができる。

1. Fixed sequencer
2. Moving sequencer
3. Privilege based
4. Communication history
5. Destinations agreement

Fixed Sequencer

Fixed Sequencer（図 5.1）は、非常に簡単アルゴリズムである。システム内の１つのプロセスが Sequencer に選ばれ、すべてのメッセージの順番を決める。図 5.1 では、ｐ１が Sequencer である。アルゴリズムは、あるプロセス p がメッセージ m を broadcast したい場合、まずはじめに Sequencer であるｐ１に対して、メッセージ m を送信する。Sequencer である p1 がメッセージを受信すると、メッセージ m にシーケンス番号を割り当て、シーケンス番号といっしょにメッセージ m を他のすべてのプロセスへ送信する。non-uniform の場合は、プロセスがメッセージ m とシーケンス番号を受信すると、即座にメッセージ m を deliver する。uniform の場合は、Sequencer が ack メッセージを用いて、すべてのプロセスが安定（正確なプロセス）していることを確認してから、メッセージ m を deliver する。このアルゴリズムは、めったに uniform であることはない。なぜなら、他のクラスのアルゴリズムより比較的コストが高いからである。

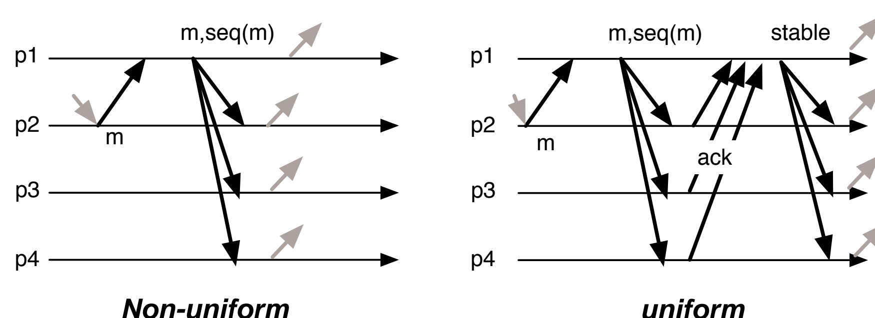

図 5.1: Fixed Sequenser

Moving Sequenser

Mobing Sequenser（図 5.2）の Sequencer の役割は、あるプロセスから他のプロセスへ渡されることである。これは、トークン（優先権番号）によって行われ、トークンはシーケンス番号を運び、プロセス間を常に移動している。

Non-uniform の場合は、プロセス p ２がメッセージ m を broaccast する場合、p ２はすべての他のプロセスにメッセージ m を送信する。各プロセスは、メッセージ m を受信すると receiveQueue にメッセージ m を蓄える。トークンを保持しているプロセス p1 は、receiveQueue に蓄えられている最初にメッセージである m にシーケンス番号を割り当てる。そして、トークンを次のプロセス p2 に送信し、シーケンス番号といっしょにメッセージ m を brodcast する。あるプロセスは、以下のような場合に、メッセージ m を deliver することが可能となる。

- メッセージ m が受信されいる場合
- メッセージ m のシーケンス番号が受信されている場合
- メッセージ m の前の各メッセージが deliver されている場合

uniform の場合は、宛先プロセスがメッセージ m を deliver できる前に、メッセージ m が安定するまで待たなければならないことと似ている。(例えば、そのプロセスがすべてのプロセスにより応答されるまで）メッセージ m が安定すること確認するには、ack をかき集めたトークンにより行われる。トークンは、常に、プロセス間を循環している必要はなく最終的にストップする。

Non-uniform は、メッセージ m が安定することを確認する必要がないため、uniform よりも早くトークンを止めることができる。

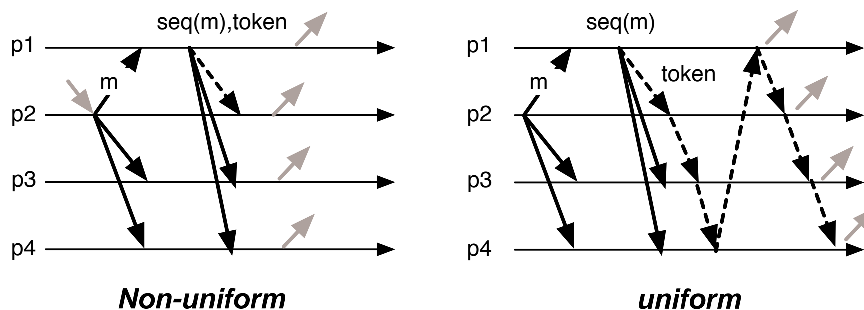

図 5.2: Moving Sequenser

Privilege-Based

Privilege Based（図5.3）では、delivery する順番を決定するのは、送信プロセスである。あるプロセスがメッセージ m を broadcast する場合、そのプロセスは、最初に privirege を取得しなければならない。Non-uniform の場合、プロセス p1 がメッセージ m を broadcast

する場合、プロセス p1 はトークンを受信するまで m を sendQueue に蓄えておく。トークンは、シーケンス番号を運び常にプロセス間を循環している。プロセス p1 がトークンを受信すると、sendQueue から最初のメッセージ m を取り出し、トークンによって運ばれたシーケンス番号を用いて、メッセージ m にシーケンス番号を割り当てる。そして、シーケンス番号といっしょにメッセージ m を broadcast する。次に、プロセス p1 はトークンのシーケンス番号を 1 つインクリメントし次のプロセス p2 へトークンを送信する。送信するメッセージの数を減らすために、プロセスは、単一のメッセージ中として、メッセージ m といっしょにトークンを送信することができる。プロセスは、メッセージ m を受信したときに、そのシーケンス番号によってメッセージ m を deliver することができる。uniform の場合は、トークンはさらに ack を運んでいる。メッセージ m が deliver される前に、プロセスはメッセージ m が安定するまで待たなければならない。これを実現するためには、十分なトークンの round trip を必要とする。

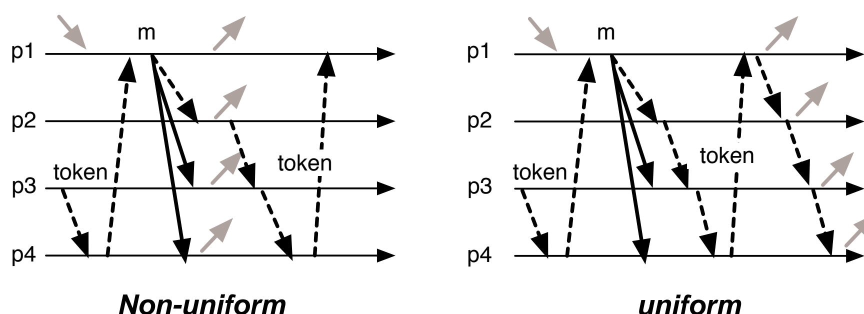

図 5.3: Privilege Based

Communication History

Communication History 図 5.4 は、メッセージを deliver する順番を Privilege Based のように送信プロセスによって決定する。Privilege Based と違う点は、各プロセスがいつでもメッセージを送信可能なことである。宛先のプロセスは、もはや deliver するメッセージが total order（全順序）を破らないこと確認するために、他のプロセスから生成されたメッセージを監視する。ほとんどの Communication History では、メッセージの causal history によって定義された不完全な順番を用いる。そして、あらかじめ決定された方法に従って順序化されたメッセージが一致する時、この不完全な順番を total order に変換する。

Communication History のアルゴリズムを説明する。ある不完全な順番は、*A-broacast(m)* イベントの論理時間を含んでいる各メッセージ m を “time stamp” するための論理クロックを用いて生成される。この不完全な順番は、後に、つながりを調停するための送信プロセスの ID 上の lexicographical order を用いて、total order に変換される。つながりとは、もし 2 つのメッセージ m と m’ が同じ論理的な time stamp を持っているなら、メッセージ m は m’ の前である。*(id(sender(m)) id(sender(m’)))* この時、*id(p)* は、プロセス p の

ID である。プロセス p は、将来低い time stamp を運ばないメッセージ m' を受信していないことを知っているなら、1 度だけあるメッセージ m を deliver することができる。

このアルゴリズムは、まだ確立されていない。なぜなら、静かなプロセス（何の反応もないプロセス）は、deliver からの他のプロセスを妨害する可能性があるからである。これを避けるためには、プロセス p が少しの時間の遅延の後、空のメッセージを broadcast すること要求される。

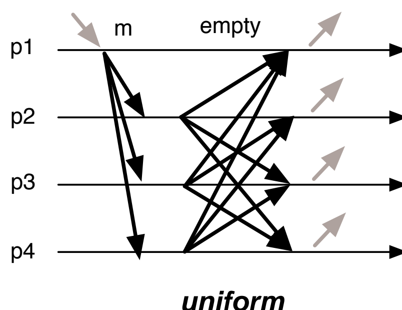

図 5.4: Communication History

Destinations Agreement

Destinations Agreement 図 5.5 では、メッセージを deliver する順番は、下記の２つのいずれかの方法で、宛先のプロセスによって決定される。

- 順番の情報は、決定的に結合された各プロセスによって生成される。
- 順番の情報は、例えば consensus のようなアルゴリズムを用いて、プロセス間で合意の上で得られる。

メッセージ m を broadcast するために、プロセス p1 はメッセージ m をすべてのプロセスに送ることおよび、メッセージ m の delivery のために coorfinator（調整役）として振る舞う。他のプロセスは、メッセージ m を受信すると、各プロセスは、ack と logical times tamp　を送信元のプロセス p1 へ送信する。プロセス p1 はすべての time stamp を収集すると、すべての受け取った time stamp から、最大値を計算し、timestamp *TS(m)* を決める。最後に、プロセス p1 は、*TS(m)* をすべてのプロセスに送信し、各プロセスはメッセージ m を *TS(m)* に従って deliver する。

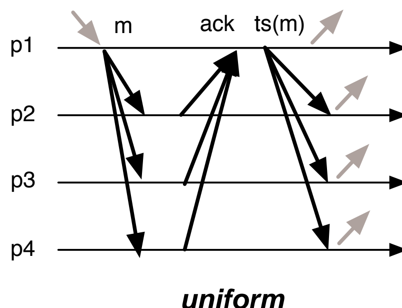

図 5.5: Destinations Agreement

5.2 Evaluating Neko + SSFNet with Two Consensus Algorithm

Atomic Broadcast を用いた二つの異なった Consensus のアルゴリズムについて、比較および性能評価を行う。この性能評価は、Urban らによって、[5] で、Consensus のアルゴリズムを Neko に実装し行われている。本研究で行う性能評価と Urban[5] で行った性能評価の違いは、シミュレートを行う環境であるシミュレートネットワークに SSFNet を用いることである。そして、Urban ら [5] による評価結果での評価結果と、本研究での評価結果を比較することで Neko の有効性を示す。

分散合意問題、すなわち、Consensus、Atomic Broadcast 等は、トランザクションや時間に制約があるようなアプリケーションを含んだ分散システムの耐故障性のために必要不可欠とされている。Unbun らによる [5] では、２つの知られた Consensus のアルゴリズムの比較を提供している。１つ目のアルゴリズムは、あるプロセスだけに通信が集中するようなパターンを用いており、２つ目のアルゴリズムは、通信が集中しないパターン、つまり、各プロセスに均等に通信が行われるようなアルゴリズムである。他の特筆すべき特徴は、この２つのアルゴリズムとも非常に似ている。この性能評価では、Consensus アルゴリズムの性能評価ための一般的な方法論も提案している。その方法論について簡単に説明する。この２つの Consensus アルゴリズムは、各プロセスが他のプロセスへ Atomic Broadcast を送信するようなシステムで分析されている。Atomic Broadcast アルゴリズムは、メッセージの delivery する順番を決定するために、Consensus の順番付けを実行する。Atomic Broadcast の性能評価を行うことは、現実的なシナリオの使い方として、Consensus アルゴリズムの性能評価を理解するの非常に良い方法である。この性能評価で用いられている Atomic Broadcast は、２つの Consensus アルゴリズムの内のどちらか一方を用いている。このアルゴリズムを比較するために、Neko を用いて多様な異なるシミュレートネットワークを用いている。ネットワークモデルは、Contention aware metrics [6] と呼ばれるモデルを用いている。このモデルは、メッセージを交換するために、計算機およびネットワーク上で複数の計算機（プロセス）が同じリソースを用いており、Neko のシミュレー

トネットワークの MetricNetwork として実装されている。Atomic Broadcast のための性能測定基準は、*early latency* である。この時間は、メッセージ m の送信と最も早いメッセージ m の delivery の間の経過である。Urban ら [5] は、（１）failuers および suspicions も起きない環境で実行した状態の latency の評価と、（２）プロセスがクラッシュした後の一時的な latancy の評価の二つを行っている。本研究では、（１）に限定して、SSFNet を用いた評価と MetricNetwork を用いた評価との比較を行う。

5.2.1 定義

性能評価で用いるシステムモデルは、分散システムのための広く確立されたモデルである。そのモデルは、メッセージ交換をだけを用いて通信を行うプロセスから成り立っている。システムは、非同期で通信を行うタイミングの振る舞いなどの前提条件はない。ネットワークは、メッセージを失わないし、また、重複したメッセージを作らないといったような外見上の信頼性がある。慣例的に、これは、失ったメッセージを再送することで簡単に達成できる。

次に、分散合意問題の非公式な定義について説明する。Consensus の問題において、各プロセスは、initial 値を提案する。Uniform Consensus は、提案された値から、すべてのプロセスが同じ値を決定することを続いて行う。

Atomic broadcast は、5.1 章で説明した通り、*A-broadcast(m)* と *A-deliver(m)* の２つの基本的な操作により定義される。Uniform Atomic Broacast は、２つのことを保証する。それは、（１）もし、あるメッセージがあるプロセスによって A-broadcast されるなら、すべての正確なプロセスは、いつかはそのメッセージを A-deliver することと、（２）すべてのプロセスは、同じ順番でメッセージを A-deliver することである。

5.2.2 ２つの Consensus Algorithms

２つの Consensus アルゴリズムについて、それらの共通のポイントと異なるポイントを解説する。そして、Consensus の上に構築される Atomic Broadcast のついて紹介する。

Consensus アルゴリズム

Consensus を解決するために、Chandra-Toueg $\diamond S$ アルゴリズム [7] と Mostefaoui-Raynalno の $\diamond S$ のアルゴリズム [8] を用いる。これからは、前者を CT アルゴリズムと後者を MR アルゴリズムとしてそのアルゴリズムを呼ぶ。

共通のポイント

このアルゴリズムは、多くの前提条件と特性を共有している。両方のアルゴリズムは、$\diamond S$ failure detectors を用いた非同期モデル向けに設計されている。両方とも、交替制のコーディネータ（調整役）法に基づいており、各プロセスは非同期なラウンドのシーケンスを実行する。そして、各ラウンドにおいて、プロセスは、コーディネータの役割を取得する。そのコーディネータの役割は、すべてのプロセス上で決定的な値を押し付けること

である。もし、それが成功すると、そのコーディネータは終了する。このケースでは、新しいラウンドから開始される。

ラウンドの実行と異なるポイント

Consensus の各ラウンドが実行すると、CT アルゴリズムは、集中的な通信の問い合わせを用いる。（図 5.6 中の左）一方、MR アルゴリズムは、集中的な通信の問い合わせを用いない。（図 5.6 中の右）この２つの図は、各２つのアルゴリズムの１つのラウンドを実行を表している。

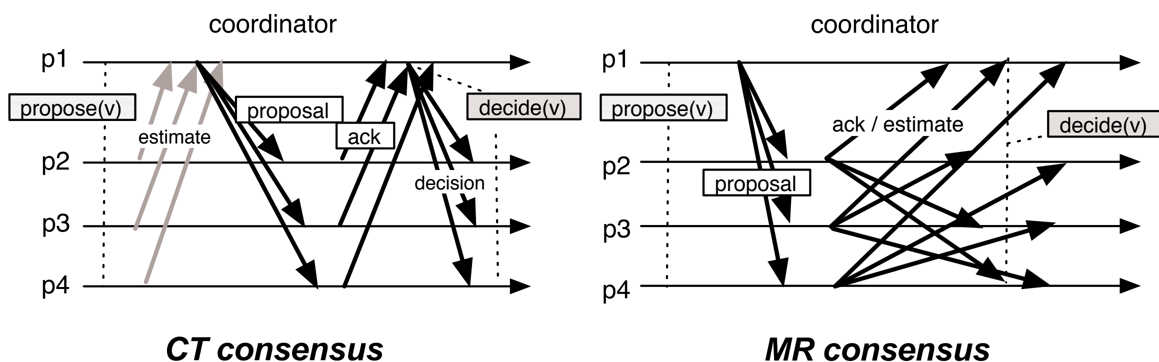

図 5.6: Example run of the both consensus algorithm

- CT アルゴリズムでは、コーディネータは、はじめにプロセスの多数決からの決定値のために estimates メッセージを集めている。このフェーズは２番目のラウンドとその後に一意必要なことである。
- 両方のアルゴリズムにおいて、コーディネータは proposal メッセージを送信する。
- その proposal が受信されると、プロセスは、acknowledgment(ack メッセージ) を送信する。CT アルゴリズムでは、ack メッセージはコーディネータだけに送信される。MR アルゴリズムでは、ack メッセージはすべてのプロセスに送信される。それに加えて、MR アルゴリズムのプロセスは、ack メッセージの上にプロセスの最新の estimate メッセージを便乗させる。順番に proposal を選択するための次のラウンドのコーディネータを認める。これは、MR アルゴリズムが estimate メッセージを送信するための独立した行程を要求していないからである。似た方法の便乗する estimate は、次のラウンドのコーディネータが ack メッセージを受信しないので、CT アルゴリズムでは不可能である。
- プロセスの多数決からの ack を受信すると、そのコーディネータ（CT アルゴリズム）とすべてのプロセス (MR アルゴリズム) は決心する。CT アルゴリズムのコーディネータは、その決定をすべてのプロセスに送信することを必要としている。(decision

メッセージ）これは、MT アルゴリズムでは必要ない。なぜなら、各プロセスが独立に決定しているからである。

The Chandra-Toueg atomic broadcast algorithm

このアルゴリズムは、プロセスがすべてのプロセスに送信したメッセージによって、A-broadcast を実行する。あるプロセスが各メッセージを受信する時に、そのプロセスは delivery する順番が決定されるまで、そのメッセージをバッファにためておく。その delivery する順番は、Consensus のシーケンスが番号付け（１、２、etc）することにより決定される。その値は、はじめに提案され、各 Consensus の決定値はメッセージ ID のシーケンスである。その delivery する順番は、Consensus(１、２、etc) のようなシーケンスの連鎖によって与えられる。

このアルゴリズムは、Consensus のアルゴリズムを理解することから、システムモデルと耐故障性の保証を引き継がれていく。この性能評価では、両方の CT および MR Consensus アルゴリズムで実現された Atomic Broadcast を用いる。

5.2.3 ベンチマーク

この章では、性能評価基準、および workload(仕事量) から成るベンチマークについて説明する。順序よく意義のある結果を得るために、評価（Consensus）に基づいてのコンポーネントの用語よりもむしろ、評価（プロセスが Atomic Broadcast を送信する）に基づいてのシステムの用語を述べる。

性能評価基準

主なる性能評価基準は、Atomic Broadcast の ealry latency である。Early latency$L$ は、以下のように単体の Atomic Broadcast のために定義される。A-broadcast(m) がプロセス p_j ある時間 t_0 と、A-deliver(m) が p_i により時間 t_i で起こったと仮定する。($i =$ １、.....、n) すると、latency は、時間が最初のメッセージ m の A-deliver（例、$L = (min_{i=1,....,n} t_i) - t_0$）までの経過した時間である。この性能評価では、$L$ が多くのメッセージを超えることと、いくつかの実行を計算している。

Workload:仕事量

latency は、いつも一定の仕事量に基づいて計測されている。この性能評価では、単純な仕事量を選択している。それは、（１）すべての宛先プロセスは同じ一定の割合で Atomic Broadcast メッセージを送信する、および、（２）A-bradcast イベントは、stochastic process から来ることである。Atomic Broadcast の全体の割合のことを *throughput* と呼び、T がそれを示す。一般的に throughput T に依存して、どれくらいの latency$L$ か決定される。

5.2.4 シミュレーションモデルの比較

性能評価への研究方法は、実システム上での測定を得るために適しているような一般的な結果を取得可能なシミュレーションである。それは、異なった環境の変化をシミュレートするためにネットワークモデルのパラメータをを変更できることである。性能評価では、Nekoを用いて分散アルゴリズムが実装されたおり、Nekoのシミュレートネットワークとして実装されたMetricNetworkを用いている。本研究では、さらにシミュレートネットワークとしてSSFNetを用いて性能評価を行う。一方、SSNetは、現実的なネットワークモデルを定義することが可能である。この２つのシミュレートネットワークは、共通する特性を持っており、ネットワークモデルを比較することでSSFNetの有効性を示すことができる。

MetricNetwork model in Neko

Nekoのシミュレートネットワークであり、性能評価に用いられいるMetricNetwork（図5.7）について解説する。MetricNetworkは、実世界におけるEthernetを単純にモデル化したものである。モデル中の要所となる点は、リソースを複数のプロセスで共有し計算していることである。この点は、分散アルゴリズムの性能のための限定的な要素であるリソースの複数のプロセスでの共有として重要である。計算機とネットワーク自身の両方ともは、ボトルネックになる可能性もある。モデル中でのリソースの上限の種類は、メッセージの伝達手段を提供するプロセス間で共有されているネットワークリソースと、ネットワーキングをするためのレイヤースタックとネットワークの制御によるプロセスの処理を提供するプロセスごとのCPUリソースである。あるメッセージmはプロセスpiからプロセスpjへ転送されることは、（1）CPU i、（2）ネットワーク、および（3）CPU jの順番にリソースを用いている。メッセージmは、各ステージがもし混雑しているなら、メッセージキューで待って次のステージに押し出される。その時間は、ネットワークリソース上で1 time unit 費やされる。また、CPUリソース上でλtime unit費やされ、これは、メッセージを送信と受信することが単純なコストを持っていると考えることできる。

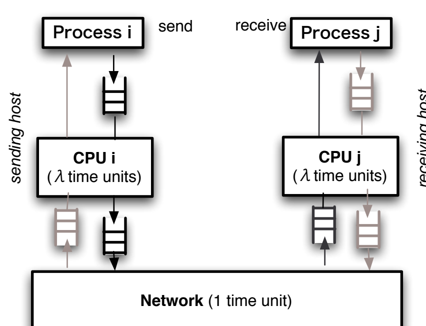

図 5.7: MetricNetwork in Neko

その λ パラメータ（$0 \leq \lambda$）は、ネットワーク上をメッセージが転送されることを計算機が比較する上でのメッセージの処理速度の関連を表している。異なった値は、異なったネットワーク環境をモデル化する。この性能評価では、λ のために設定の種類を用いての実験を考慮している。さらに、このネットワーク環境はユニキャストとマルチキャストによるメッセージの通信も考慮している。

Network model in SSFNet

SSFNet を用いてモデル化することができるネットワークについて解説する。(図 5.8) SSFNet は、現実的なネットワークモデルを構築可能である。protocol は、実世界に存在する SSFNet が実装したプロトコル（e.g TCP/IP、UDP/IP etc）用いることができ、そのプロトコルを用いてプロトコルスタックを構成することが可能である。SSFNet の各計算機（Host、Router）は、NIC を持っておりパラメタとして、buffer 値や送出するパケットあたりの latancy(送出するまでの遅延) を各計算機ごとに設定可能である。また、計算機同士を接続している link の送信遅延である delay と link のバンド幅を各リンクごとに設定可能である。そして、10000 台以上の計算機を同時にシミュレーション可能であり、そのネットワークトポロジの開発者が自由に定義可能である。

Neko の MetricNetwork は、単純な Ethenet をモデル化しているので、この SSFNet のモデルと似ている。それは、Neko における λ の値は、SSFNet における latency であり、Neko におけるネットワーク遅延は、SSFNet で定義できる link delay とバンド幅に対応することができる。

SSFNet のモデルにおいて、$host_i$から $host_j$ までのパケットの伝播流れを説明する。$host_i$ は、定義されたプロトコルに従って、自分自身の NIC を用いてパケットの送信を試みる。この時、もし NIC の buffer が混雑しているならば、そのパケットは、混雑が解消するまで待って、パケットを送出する。送出する時、NIC 上に定義された latency に従って、そのパケットの送出を送らせる。次に、接続されている Router へ、link に定義されたバンド幅に従って経過時間が決定され、定義された delayl によって、経過時間に遅延を足した時間後、その link を通じて伝播される。Router の Host と同じように buffer と latency が定義されているので、その定義に従った経過時間がシミュレートされる。その後、同じように link、Host の NIC の順番で伝播され、プロトコルに従って、$host_j$ へパケットが送信される。

Neko のシミュレートネットワークとして SSFNet を用いる場合、メッセージのサイズを定義することが可能であるが、SSFNet では、パケットに分割してそのメッセージは、目的の Host へ送信される。つまり、上記のような順序のシミュレートを SSFNet は、分割されたパケット毎に行う。これは、実ネットワークを精密にモデル化していることにより行われ、Neko のシミュレートネットワークである MetricNetwork よりも、現実的でかつ有効なシミュレート結果が得られる。

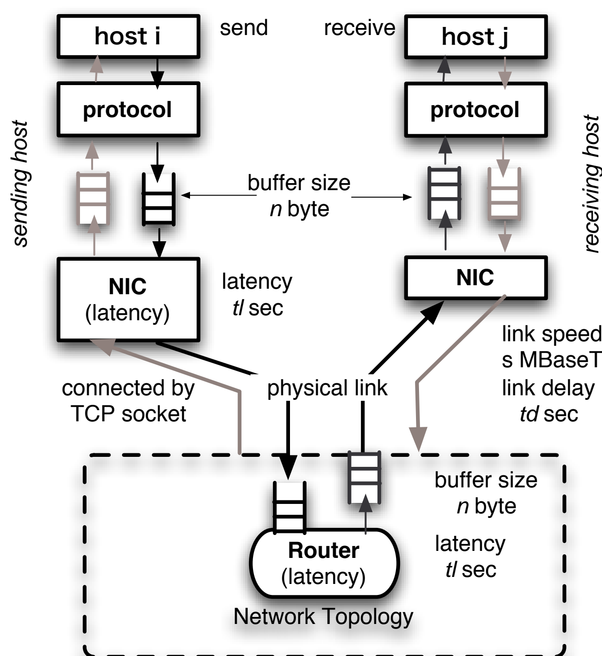

図 5.8: SSFNetNetwok in SSFNet

5.3 評価結果

この２つのネットワークモデルを用いた性能評価の結果を報告する。Neko の MetricNetwork で λ: 0.1、１、１０のように設定し、プロセス数 n=7を用いてシミュレーションを行った結果と、SSFNet の SSFNetTCPNetwork で、すべての $latencyofNIC = 100[usec]$ 、$linkdelay = 1msec$、および、$linkspeed = 100MBaseT$ と、各計算機上の NIC の Buffer を 8000byte、ルータの Buffer を 16000byte に設定して、プロセス数 n=7を用いて、２種類のネットワークモデルで結果を取得した。SSFNet のネットワークトポロジーは、７つの計算機がルータに接続している単純な Local Area Network を定義している。この各パラメタは、4 章で説明した DML ファイルのパラメタを用いている。図 5.9 は、MetricNetwork を用いた３つの異なった λ による結果で、図 5.10 は、SSFNetTCPNetwok を用いた結果である。各図は、latency 対 Throughput のグラフを示している。

The case　of using MetricNetwrok

- CT アルゴリズムは、$\lambda = 10$ の場合、最も悪いパフォーマンスである。(図 5.9) この理由は、CT アルゴリズムが MT アルゴリズムよりも多くの負荷をコーディネータへ与えているからである。その要因は、CT アルゴリズムは、proposal メッセージと ack メッセージを収集し、さらに、決定値を図 5.6 の左記のように送信しなければならないためである。

- MT アルゴリズムは、$\lambda = 1$ の場合、最も悪いパフォーマンスである。(図 5.9) パ

フォーマンスの相違点は、throughput がおおよそ比例していることである。この振る舞いの理由は、CT アルゴリズムとは違って、CPU リソースの負荷が重要なことではないからである。従って、結論付けられる事実は、MR アルゴリズムはネットワークに負荷を多く与える。throughput の増大は、ネットワークバッファのモデルにおける高負荷な割り込み時間を導いている。

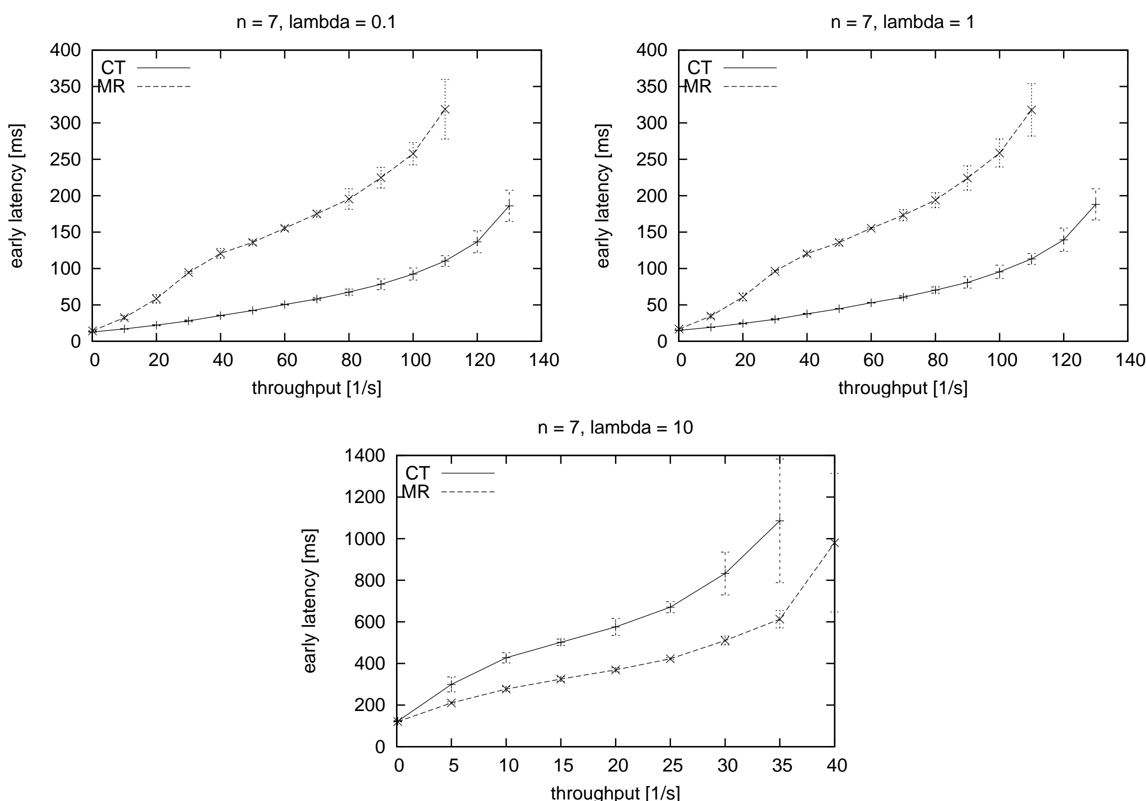

図 5.9: Result by using MetricNetwork in Neko ($\lambda = 0.1, 1, 10$)

The case of using SSFNetTCPNetwork

- SSFNetTCPNetwork を用いたネットワークモデルにおいて、CT アルゴリズムは、MT アルゴリズムよりも、悪い性能である。(図5.10) このグラフは、MetricNetwork を用いた事例における $\lambda = 10$ のグラフと非常によく似ている。MR アルゴリズムは、MetricNetwork と同じでおおよそ比例して throughput が増加している。CT アルゴリズムは、MetricNetwork の時と非常に良く似た曲線を描いている。同じような結果が得られた理由は、SSFNet で定義したネットワークモデルが MetricNetwork の $\lambda = 10$ の場合と同じようなモデルといえるからである。SSFNet のネットワークモデルの *latency of nic* は、MetricNetwork の λ に当たると考えられるので、CT ア

ルゴリズムのコーディネータにおけるメッセージの送信がボトルネットとなっていることがわかる。

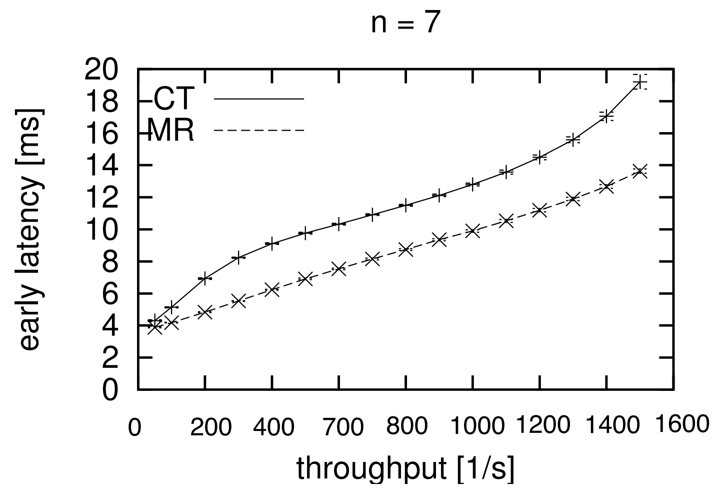

図 5.10: Result by using SSFNetTCPNetwork in SSFNet

第6章　まとめ

本研究では、Neko の問題点である現実的なシミュレートネットワーク環境を持っていない点を SSFNet のシミュレーションネットワークを利用することで解決した。Neko と SSFNet の概略と特徴について、詳細に説明を行った。具体的な方法は、Neko 側と SSFNet 側の両方に、インタフェースだけを作成し、Neko と SSFNet の中核の実装を変更することなく統合した。実装上の問題点として、Neko と SSFNet のスケジューラの同期、アドレス方式の違い、メッセージ方式の違いおよびコンフィグファイルの仕様変更などがあった。これらの問題を解決し実装したシミュレートネットワークとして SSFNet が利用可能な Neko の性能評価を分散アルゴリズムのひとつである Atomic Broadcast を用いて行った。その際、Atomic Broadcast の定義について詳細に解説し、性能評価で用いた Atmic Broadcast を実現するための分散アルゴリズムである 2 つの Consensus についても解説した。また、性能評価において、Neko のシミュレートネットワークである MetricNetwork のネットワークモデルについて解説し、SSFNet のネットワークモデルとの比較を行うことで *Neko with SSFNet* の有効性を示した。

6.1　謝辞

In the first, I'm most grateful to Xavier Défago. I took pleasure in our steady on our laboratory which gave me wonderful environment. I would like to say “Arigato” to Péter Urbán who is collaborate researcher. Your Neko is towering tool and gave me most enjoyment. Lastly, I was most interested life with partners in Xavier Lab.

参考文献

[1] Peter Urban, Xavier Defago, and Andre Schiper, “Neko: A single environment to simulate and prototype distributed algorithms.” Journal of Information Science and Engineering, 18(6):981-997, November 2002.

[2] J.Cowie,D.M.Micol,A.T.Ogielski, “Towards Realistic Million-Node Internet Simulations.” Proc Int Conf on Parallel Distributed Processing Techniques Applications (PDPTA’99), June, 1999.

[3] X.Defago, A.Schiper, and P.Urban, “Comparative performance analysis of ordering strategies in atomic broadcast algorithms.” IEICE Trans. on Information and Systems, Vol.E86-D, No.12, pp.2698-2709, December 2003.

[4] X.Defago, A.Schiper, and P.Urban, “Total order broadcast and multicast algorithms: Taxonomy and survey. ” Research Report IS-RR-2003-009, Japan Advanced Institute of Science and Technology, Ishikawa, Japan, September 2003.

[5] P.Urban, N.Hayashibara, A.Schiper, and T.Katayama, “Perfomance Comparison of a Rotating Coordinator and a Leader Based Consensus Algorithm.” In Proc, 23nd IEEE Int’t Symp, on Reliable Distributed System (SRDS), page 4-17, Florianopolis, Brazil, October 2004.

[6] P.Urban, X.Defago, and A.Schiper, “Contention-aware-metrics for distributed algorithms: Comparison of atomic broadcast algorithms.” Proc. 9th IEEE Int’l Conf. on Computer Communications and Nerworks (IC3N 2000), 2000, page 582-589.

[7] T.D. Chandra and S.Toueg, “Unreliable failure detectors for reliable distribbuted systems.” Journal of the ACM, 43(2), 1996, 225-267.

[8] A.Mostefaui and M.Raynal, “Solving consensus using Chandra-Toueg’s unreliable failure detectors: A general quorum-based approach.” Proc. 13th Int’l Symp. on Distributed Computing (DISC), Bratislabe, Slovak Republic, 1999, 49-63.

[9] L.Lamport,“ Time, clocks, and the ordering of events in a distributed system.” Commun. ACM, vol.21,no.7,pp.558-565, 1978