NAIST-IS-DD1061002

博士論文

ネットワークソフトウェア検証のための大規模な検証環境構築に関する研究

榎本 真俊

2018 年 3 月 15 日

奈良先端科学技術大学院大学
情報科学研究科

本論文は奈良先端科学技術大学院大学情報科学研究科に
博士 (工学) 授与の要件として提出した博士論文である。

榎本 真俊

審査委員：

門林 雄基教授	（主指導教員）
岡田 実教授	（副指導教員）
安本 慶一教授	（副指導教員）
篠田 陽一教授	（北陸先端科学技術大学院大学）

ネットワークソフトウェア検証のための大規模な検証環境構築に関する研究*

榎本 真俊

内容梗概

ネットワークソフトウェアやプロトコルの検証はそれらの実環境への展開の前に十分行われなければならない．ネットワークソフトウェアやプロトコルが動作する環境はインターネットを始めとして，センサーネットワークや企業，工場内のネットワークなど大規模な環境での動作が想定されている．しかし，検証者が用意できる資源や検証施設が有する資源は限られているため検証者が必要とする大規模検証環境の構築は困難である．一方，グリッドコンピューティングやクラウドコンピューティングの分野では，垂直スケールと水平スケールにより，計算能力の高度化や規模の拡張を行っている．

本研究では，垂直スケールおよび水平スケールの概念を用いてネットワークソフトウェアやプロトコルの大規模検証環境の構築手法を提案する．提案手法は，垂直スケールの概念を大規模検証環境の構築に適用した (1) 与えられた検証資源に対してハードウェアの仮想化技術を用いて検証のためのノードを増加させるためのメモリ割り当てアルゴリズムと，水平スケールの概念を適用した (2) 異なるソフトウェアおよび運用者により提供されている検証施設の資源を透過的に利用する技術により検証に利用可能な資源を増加させる技術で構成される．まず，(1) に関しては，ネットワークソフトウェアやプロトコルの検証環境を構築するために，仮想計算機に割り当てが必要な資源について検討を行う．検討結果から，本論文では，特に物理メモリ量を対象とし，仮想計算機への資源の割り当て量推定アル

*奈良先端科学技術大学院大学 情報科学研究科 博士論文, NAIST-IS-DD1061002, 2018 年 3 月 15 日.

ゴリズムについて検討を行う．推定式の導出には，検証に用いるプロトコルおよびソフトウェアの静的解析及び動態解析を用いて検証環境構築のための必要なメモリ量の推定手法を提案し，規模追従性に関して検証を行う．(2) では，はじめに，ネットワークに関するソフトウェアやプロトコルの検証を対象とした検証施設の構成要素について検討を行い，検証施設同士の連携手法を提案する．本研究では，検討した構成要素の中で検証施設管理ツールの連携について，検証施設が有する資源に対する設定投入の連携手法である Collaborative Testbed Federator(CTF) の提案し，検証に使用されるソフトウェアの制御に関する連携手法として Ditto Subsystem を提案する．これらの連携手法の提案により，検証者は透過的に検証施設にまたがった環境での検証が可能になる．

本研究では，提案手法の規模拡張性と有用性を検証する．最後にネットワークソフトウェアやプロトコル検証の今後の展開と課題についてまとめる．

キーワード

ネットワーク, ソフトウェア検証, テストベッド, 大規模検証環境構築手法, テストベッド連携

Studies of building a large scale experiment environment for network software verification*

Masatoshi Enomoto

Abstract

We need to verify sufficiently network software and protocol verification before deploying to its execution environment. Execution environment for these is assumed a large-scale environment, for instance Internet, sensor network, and corporate and factory network. However, constructing a large-scale environment is difficult, because of the limitation of resources that users can prepare. Computing power is improved with scale up and scale out in cloud computing and grid computing.

In this research, I propose a building large-scale experiment environment method using the concept of scale up and scale out. The proposed method is mainly composed of two technologies, (1) a memory allocation algorithm for increasing the number of nodes for verification using hardware virtualization technology and (2) federation method among testbeds that are managed by different software.

Regarding (1), I propose the method for estimating amount of memory which is required to construct verification environment by static analysis and dynamic analysis of software. In (2), at first, I examine the components of testbed for network emulation. From the result of consideration, we propose the Collaborative Testbed Federator (CTF) which is a cooperative method of setting a parameter to each testbed resources. CTF is enabling to build federated environment without changing testbed management tools. Next, we propose the Ditto Subsystem which is a cooperation method of controlling software used for verification.

*Doctoral Dissertation, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD1061002, March 15, 2018.

In this research, I examine the validity of the proposed method through a study of the scale extensibility and the usefulness of the verification environment. Finally, I summarize a prospects and issue of network software and protocol verification in the future.

Keywords:

network, software verification, testbed, large scale exepriment environment, testbed federation

目 次

1. 序論 **2**
1.1 科学的検証と手法 2
1.2 ネットワーク分野における検証手法 2
1.3 大規模検証環境における課題 5
1.4 大規模な検証環境の構築と検証 7
1.5 本研究の工学的貢献 9
1.6 本論文の構成 10

2. 大規模な実験環境構築に関する現状と課題 **11**
2.1 シミュレーションによる検証手法 11
2.1.1 ネットワークに関するシミュレータ 11
2.1.2 シミュレータの大規模化および高速化 12
2.2 エミュレーションによる検証手法 13
2.2.1 エミュレーション環境構築の補助ツール 13
2.2.2 エミュレーションの規模拡大手法 15
2.3 まとめ 16

第 **I** 部　大規模検証環境構築のための効率的な計算機への資源割り当て **18**

3. 大規模模擬ネットワーク構築のための仮想計算機へのメモリ割り当てアルゴリズム **18**
3.1 概要 18
3.2 定式化する資源の決定 18
3.3 消費メモリの定式化 20
3.3.1 OS が必要とするメモリ量 21
3.3.2 仮想計算機を利用した場合の必要メモリ量 22
3.3.3 目的関数の定式化 23

3.4 ネットワーク関連の資源の定式化 24
3.4.1 ソケット数による制約条件 25
3.4.2 ファイル記述子の制約条件 25
3.4.3 送受信パケット数の定式化 26
3.4.4 目的関数の定式化 35
3.4.5 送受信バッファサイズの定式化 37
3.4.6 トラフィック量に関する目的関数の定式化 39
3.5 目的関数の組み合わせに関する考察 41
3.6 提案手法 44
3.6.1 要求事項 44
3.7 EBGP ルータのメモリ消費量モデル 45
3.8 Quagga におけるケーススタディ 47
3.9 小規模実験による動的解析と回帰式のパラメータ導出 48
3.10 実験環境 48
3.11 経路フィルタ 49
3.11.1 計測 1:隣接関係ごとの隣接 AS 数と Quagga の消費メモリ量の関係 50
3.11.2 計測 2:Quagga が保持する経路数と Quagga の消費メモリ量の関係 53
3.12 重みパラメータの選定 57
3.13 選定した重みパラメータの規模追従性の検証 62
3.14 議論 63
3.14.1 提案手法の汎化をおこなうために 64
3.15 まとめ 64

第 II 部 大規模検証環境構築のための検証施設間の資源および機能の連携 66

4. 検証施設の資源および機能の連携 66
4.1 背景 66

4.2 検証施設の連携における現状と問題点 67

5. 検証施設連携時の機能の連携 69

5.1 概要 69

5.2 背景 70

5.3 Ditto Subsytem の提案 71

5.3.1 StarBED と DeterLab の差異 71

5.3.2 資源予約と割り当て 72

5.3.3 Ditto subsystem の要求 73

5.4 TopDL 記述の SpringOS の設定手続きへの変換 74

5.5 Testbed Controller による検証施設管理ツールの制御 75

5.6 SpringOS が対応していないパラメータ設定を行うための機能の追加 78

5.6.1 Resource Mapper 79

5.7 議論 79

5.7.1 Translator 79

5.7.2 Testbed Controller 80

5.7.3 Resource Mapper 80

5.7.4 Function Adapter 80

5.7.5 Ditto subsystem 80

5.7.6 Ditto subsystem の仮想化環境への適用 81

5.7.7 異なる記述形式を組み合わせた実験環境記述 82

5.7.8 Ditto subsystem の他の検証施設管理ツールおよび検証環境構築記述への適用について 82

5.8 まとめ 83

6. 検証施設連携時の資源の連携 84

6.1 概要 84

6.2 背景 84

6.3 検証施設の資源管理手法の現状と連携を行う上での問題点 84

6.4 CTF の設計 86

6.5 TT4S の設計 89
6.5.1 アカウントマップ機能 89
6.5.2 資源の確保と動作確認機能 89
6.5.3 StarBED 内の実験環境構築 90
6.5.4 実験ネットワークの接続 91
6.6 StarBED と DeterLab の連携例 92
6.6.1 EC で指定可能な設定情報 92
6.6.2 StarBED および DeterLab での実験環境構築手順 92
6.6.3 検証施設間ネットワークの構築 93
6.7 議論 93
6.7.1 資源確保の方法について 94
6.7.2 資源のコントロールについて 95
6.7.3 検証施設間の実験ネットワークの接続について 95
6.7.4 CTF の他の検証施設との連携の可能性について 96
6.8 まとめ 97

7. 議論 **98**
7.1 仮想計算機を用いた検証環境構築時の忠実度 98
7.2 複数の大規模検証施設にまたがった検証環境の監視および観測 . . 100
7.3 複数の大規模検証施設の資源利用時の資源制御 101
7.4 他の用途への応用の可能性 102
7.4.1 クラウドコンピューティング 103
7.4.2 制御システムを有する施設での検証への応用 103

8. 結論 **105**
8.1 本研究の成果 105
8.2 本研究分野の課題と展望 106

謝辞 **108**

参考文献 **109**

付録 **117**

A. 業績リスト **117**

図目次

1 Uplinkの経路フィルタ 48
2 各隣接関係におけるbgpdのメモリ消費量 50
3 トポロジ全体の経路数とQuagga bgpdおよびzebraのメモリ消費量 51
4 500ASにおけるw=1の場合のQuaggaの消費メモリ量推定結果と実際にQuaggaが使用したメモリ量 57
5 500ASにおける隣接AS数ごとのQuaggaの消費メモリ量の推定結果と実際にQuaggaが使用したメモリ量の誤差 ($\omega = 1$) 58
6 500ASにおける1隣接ASあたりの割り当てメモリ量と消費メモリ量の差 ($\omega = 1.4$) 60
7 1000ASにおける1隣接ASあたりの割り当てメモリ量と消費メモリ量の差 ($\omega = 1.4$) 61
8 1500ASにおける1隣接ASあたりの割り当てメモリ量と消費メモリ量の差 ($\omega = 1.4$) 62
9 テストベッド連携の概念 67
11 Fllowchart of Net Constellation 77
12 CTF architecture 86
13 TT4Sの処理の流れ 88
14 StarBEDおよびDeterLabでの環境構築の流れ 91

表目次

1 目的関数の組み合わせ(ノード数決定) 43
2 実験環境のハードウェア仕様 48
3 StarBEDとDeterLabのノード間のスループット 94
4 StarBEDとDeterLab上のノード間のジッター 94

1. 序論

1.1 科学的検証と手法

なぜ我々は実験を行うのか？

人類はこれまでさまざまな分野で未知の問題に取り組んで来ている．人類にとって未知の現象や事実を説明するための手段として，観察や思考，実験や検証を行ってきた．未知の事象を説明するためには，自身が持ちうる知識を用いて仮説や論理を立てその仮設や論理を検証することが必要である．

人類は実験のために様々な器具や装置を開発してきた．薬品の反応を観察するための器具である試験管，光を分散，屈折させるための三稜鏡，微小な物体を観察するための顕微鏡や天体から出る可視光，赤外線や紫外線を観測する天体望遠鏡などがある．

実験室はこれらの器具や装置を使用するために条件や環境を整えた仮説の検証を行うために理想に近い環境として整備されており，実験者は実験室で仮説の検証のための実験に取り組んでいる．理想的な実験環境とは個々の仮説によって異なるはずであり，実験室は様々な仮説の検証を行うことができるよう構築されている．実験室は検証を行うためにより理想的な環境となるように環境や機材が用意され，検証者は実験室の中で自身のたてた仮説の検証をおこなっている．

1.2 ネットワーク分野における検証手法

情報科学の分野の中で，ネットワークを扱う分野でも研究者は多くの検証プラットフォームを構築してきた．ネットワーク分野において，検証の手法として大きく２種類の検証方法がある．一つはネットワークのプロトコルや無線通信における接続性を実際の機器やソフトウェアを用いるのではなく，モデル化することで演算によって検証を行うシミュレーションと呼ばれる手法である．シミュレーションプラットフォームの代表的なものとして，OPNET [1] がある．OPNET はネットワークシミュレータとして開発され，パケットレベルのシミュレーションが可能である．さらに，モジュールを組み込むことで無線通信や地形に依存した電波伝

搬の計算も可能となる．OPNET は OPNETMIL3 Inc. により開発が開始され，現在も開発が続いている．1988 年に Keshav は REAL と呼ばれるネットワークシミュレータを提案した [2]．REAL の後，REAL を発展させる形で NS version1 が提案された [3]．その後，機能の追加とともに現在は NS version3 [4] が開発されている．NS はネットワークプロトコルの検証を目的として開発されている．NS はイベント駆動型のシミュレータとして設計，および実装されている．

スーパーコンピュータやグリッドコンピュータまた，Graphics Proccessing Unit (GPU) クラスタの高性能化により，多くのノードを複雑なパラメータで計算可能となり大規模な検証が可能になった．PC の低価格化や大規模な検証環境の利用および仮想計算機技術のエミュレーションよる検証ノードの多重化が可能になっている．

一方，モデルによる計算を用いたシミュレーションによる検証ではなく，実際の機器やソフトウェアを動作させることで検証を行う手法がある．実際にソフトウェアを動作させ検証を行う手法は，エミュレーションと呼ばれている．エミュレーションは，ソフトウェアの開発者が保有する PC サーバを用意することにより行う．1990 年代までは，PC は高価であり検証規模は大きくすることが困難であった．しかしながら，ソフトウェアの動作環境の一つであるあるインターネットの爆発的な大規模化や，分散処理技術の発展によりソフトウェアの動作する規模は大きくなり，検証者が用意できる検証規模との乖離が大きくなっていた．この問題を解決するために，多くの PC サーバや広帯域なネットワークを用意しその資源を研究者が共有して利用する大規模な検証施設が世界各国に構築されてきた．ネットワークに関する技術を検証するための検証施設として，ユタ大学の Jay らは 1999 年に後の Emulab の基礎となる “A Large-Scale Network Testbed” というタイトルの発表を行っている．このときはユタ大学に 105 台の PC サーバを配置し PC 間を 100Mb で接続していた．この中で著者らは中規模から大規模な設定可能な検証施設を，“network for research” と定義した．これがネットワークのための検証施設の基礎的な考え方になっている． その後，2001 年にユタ大学では Emulab が構築され，その後 Emulab の研究者らよって開発された Emulab の検証の補助ツールである Emulab-tools は世界各国のテストベッドで採用されている [5]．代

表的な Emulab-tools を採用した検証施設に南カリフォルニア大学の情報科学研究所がセキュリティの検証のために拡張さている Deterlab [6] がある．これ以降にユタ大学以外でもネットワークの検証を目的としたテストベッドが構築・運用が開始された．2002 年に情報通信研究機構 (NICT) の前身の通信・放送機構 (TAO) により StarBED [7] の運用が開始された．StarBED はその後，2006 年に StarBED2，2011 年に StarBED^3 となり現在は 2016 年から StarBED4 として運用されている．2002 年 Chun らにより PlanetLab プロジェクトが開始した [8]．PlanetLab は企業や大学が PC サーバを提供することで他の拠点が有する PC サーバを含めて研究に利用可能にした．PlanetLab には多くの企業や大学が参加し多くの PC サーバが PlanetLab の資源となり，研究者は自身が保有する環境よりもより大規模な検証環境を容易に手に入れることができるようになった．ヨーロッパでは 2004 年に PlanetLab Europe が立ち上がり，以降ヨーロッパ圏内の大学で PlanetLab に関する議論が活発に行われるようになった．2009 年に OneLab [9] の運用が開始されている．OneLab は PlanetLab 上に構築された検証環境である．

Emulab などのように一つの拠点に PC サーバと PC サーバ間のネットワークによる検証環境を提供するような検証施設が構築される一方，以前から，大学などの複数の拠点間を広帯域なネットワークで接続することにより検証環境を検証者に提供するようなものも構築されてきた．アメリカでは 1997 年に Internet2 プロジェクト [10] が開始され Internet2 のプラットフォームの上で次世代のネットワーク技術の開発やアプリケーションの検証が行われてきた．日本では，1999 年に NICT の前身の TAO により Japan Gigabit Network (JGN) [11] の運用が開始され，Internet2 と同様に次世代のネットワーク技術の開発や利活用に関する研究開発が行われてきた．JGN は 2004 年に JGN2，2008 年に JGN2plus，現在は 2011 年に JGN-X となり国内外の研究者に利用されている．このような融資による資源の提供による検証環境や，複数の大学や国家による大規模検証施設の構築により研究者は容易に従来よりも多くの資源を使った検証が可能になった．これらのように大規模に検証環境を構築するような施設や設備が構築されてきた一方で，PC の価格の下落や Xen [12] や KVM [13] などの仮想計算機の開発やクラウドコンピューティングも研究者により多くの資源を利用可能としている．

シミュレーションは検証に環境や構成の柔軟さや，高い観測性や再現性がある．そのため，プロトコルやシステムのモデルの検証には有効である．一方で開発したソフトウェアの動作に関してはシミュレーションでは検証を行うことができない．エミュレーションは，ソフトウェアやシステムが動作する環境に近い環境での検証を行うため，動作環境で挙動やおきうる中断または停止の原因の調査が可能である．一方で，検証環境の構築や制御にかかるコストが大きくなってしまう．また，オペレーティングシステム上での動作を行う場合，割り込み処理等によりソフトウェアの挙動の再現性がシミュレーションより悪い．

本研究では，ソフトウェアの検証を目的とし，エミュレーションによる検証環境の構築を行う．現在のシステムやソフトウェアが動作する実環境は，インターネットを含め大規模になっている．次節では，エミュレーションを行う環境の大規模化に関する課題について議論する．

1.3 大規模検証環境における課題

ネットワークに関するプロトコルやソフトウェアの検証に利用可能な検証環境は大規模になってきている．しかしながら，どの検証環境でもインターネットの規模の増加には追従できていない．また，検証に用いる環境が大きくなることによって起きうる課題もある．本節では，エミュレーションにおける検証環境の大規模化に関する課題について検討を行う．

前節において，大規模検証施設の構築により研究者が利用可能な資源が増えている点について述べた．しかし，ほぼすべての大規模検証施設の資源は多くの研究者によって共有して利用することを想定しており，一人の研究者がすべての資源を利用することはできない．そのため，利用可能な資源が増減する可能性が高く，検証の直前まで自身がどの程度の計算資源を利用できるかわからない．また，割り当てられた資源が検証に必要であると考えられる量を満たさない場合は割り当てられた規模に合わせて検証規模を小さくするか，必要な規模が割り当てられるまで検証を行わないなどの制限がある．

このような検証環境の規模に関する制限の緩和の手法として大規模検証施設間の連携の手法が提案されている．大規模検証施設間の連携手法は，異なる２つ

の施設の間で計算資源の融通および資源の制御の共通化を目的として研究が行われている．主な大規模連携施設間の連携の研究には，Deterlab により行われている Emulab の資源管理ソフトウェアにより管理されているテストベッド同士を連携する Deterlab Federation Architecture [14], PlanetLab Europe の資源を連携することを目的とした Panlab [15] や，ヨーロッパの大規模検証施設の連携を目的とした Fire project [16] がある．また，連携の概念を提唱した Slice-based Federation Architecture (SFA) [17] がある．

テストベッド連携の手法により，それぞれの大規模検証施設で利用されていない資源の有効活用や，単一の大規模検証設備の資源だけでは足りなかった規模の検証が可能になっただけではなく，新たな検証の可能性が見出された．それぞれの大規模検証設備では，固有の資源管理や制御のためのソフトウェアを開発や，他の大規模検証施設のソフトウェアを利用している場合も自分たちで利用しやすいように拡張を加えている．また，それぞれの大規模検証施設を利用する研究者や運用者によって独自のソフトウェアの開発が行われている点，また使用している機器が異なるため，大規模検証施設によって有する機能が異なる．これが大規模検証施設の連携により，複数の大規模検証施設が持っている機能やソフトウェアを組み合わせて，単一の大規模検証施設では行えなかった検証が可能となった．つまり，大規模検証施設の連携により，検証規模の拡大に対する寄与だけではなく，検証内容の強化に関しても寄与する手法となる．

以下に大規模な検証環境を構築する際に挙げられる課題を示す．

- 大規模検証施設で利用できる資源が固定ではない
  大規模検証施設では資源が複数の研究者で分けて使用するため，利用時期により一人の利用者が使用できる資源が増減しかつ使用できる資源の量の確定が検証の直前になってしまう．

- 異なる管理ソフトウェアを用いている大規模検証施設の連携
  Deterlab Federation Architecture や Panlab はそれぞれ Emulab が開発した管理ソフトウェアにより管理されている大規模検証施設間の連携，Panlab は PlanetLab の管理ツールである PLC [18] により管理された資源をもつ大規模検証施設を連携する手法である．そのため，StarBED のような独自かつ

他の大規模検証で利用されていない管理ソフトウェアをもつ大規模検証施設では容易に大規模検証施設連携を行うことができない．

- 大規模検証施設の機能の違いによる検証計画の煩雑化
  大規模検証施設の連携手法により資源を手に入れた研究者はそれぞれの資源がどの大規模検証施設のものか判断し，それぞれの大規模検証施設の機能に合わせて検証計画を作成しなくてはならない．

1.4 大規模な検証環境の構築と検証

ネットワークに関するプロトコルやソフトウェアに関する検証において，大規模検証環境でのテストをする場合の多くは検証者自身が用意できる計算機資源に限界があり，検証者が求める規模の検証が行えない場合であることが多い．多くの大規模検証施設やクラウドコンピューティングでは，検証者が求める規模の検証を満たすまたはそれに近い規模の計算機資源を用意することができる．一方で，大規模検証施設やクラウドコンピューティングは共有であることもしくは利用に対してかかるコストの麺から，すべての場合で検証者が求める規模の環境を用意できるとは限らない．

検証者が求める規模の検証環境が用意できたとしても，検証においては規模の大きさに比例してさまざまな課題がある．

制御に関する課題 検証に用いるノードの数が多くなればなるほど検証者が制御しなければならないノードの数は多くなる．現在，並列に制御コマンドをノードに対して発行する手法は存在しているが，一方で，これらの多くは同じ動作を複数のノードに対して発行することを想定している．検証においては，それぞれのノードが特定の役割を持って動作する場合があり，すべてのノードに同じコマンドを発行するだけではない場合もある．

監視および観測の課題 制御と同様に検証ノードに比例して監視および観測が必要なノード数は増える．また，検証時は動作に不具合がないかどうかの確認も含まれるため，できるだけ詳細な情報を集める必要があるため，その分収集するべきログや収集方法を考えなければならない．

忠実さの課題 検証を行う理由の一つに，作成したソフトウェアや設計したプロトコルが実環境での展開に問題ないかを検証するために行う．しかし，実環境と検証環境では十分な計算機資源があれば規模という観点では十分な検証を行うことが可能であるが，実環境とはPCや通信機器や通信の品質が異なってくる．さらに，本研究では，規模の増やす手法として，単一のPCサーバに対して計算ノードを多重化させ動作させる手法の提案を行っている．この時，仮想化技術を用いているため，PC上にオペレーティングシステムをインストールしたときに比べて，ソフトウェアの挙動が変わる可能性がある．

制御と監視および観測は規模が大きさに比例して実行が困難になる．一方，忠実さに関しては，基本的には規模の大きさには依存しない．しかし，本研究では，規模の拡大手法として仮想計算機による計算機資源を分割する手法を用いている．この手法では，ハードウェアとオペレーティングシステム(OS)の間に仮想計算機を実現する制御プログラムが介在することになる．そのため，OSとカーネルの間に一段階処理が発生することになり，ハードウェア上に直接OSをインストールしたときと挙動が変わる可能性がある．リアルタイム処理など検証の種類によっては検証実行時の挙動が検証者の想定と異なる可能性が発生する．そのため，規模と検証環境の忠実さは，採用する規模の拡大手法により依存関係が異なる．

検証においては，検証を行うソフトウェアまたはプロトコルが動作する環境と同じ環境，同じ動作条件で動作した上で問題点の洗い出しのためにソフトウェアやプロトコルの状態や動作状況が観測できることが検証環境を提供するという分野においては究極の目標である．一方で，現状では規模という点においても実環境として想定される環境の規模が検証環境の規模の増加に比べて大きい現状では，これまでの手法を拡張し検証者が入手できる計算機資源の増加またはその上で構築可能な検証環境の制限の緩和を行うことは重要であると考えられる．

本研究では，大規模な検証環境を構築する手法の提案を行うことで．1.3の問題の解決を行う．一方，先に述べた課題については，制御に関しては仮想計算機を用いた検証環境構築において各検証ノードに遠隔でコマンドを発行する手法と検証ノード起動時に自動的にシナリオを実行する手法2種類をためし，そこで得

られた知見について述べる．また，検証施設の連携手法においては検証施設で提供される制御方法が異なる場合での知見について述べる．監視及び観測手法に関しても仮想計算機を用いた検証環境構築において，各検証ノードの状態と検証対象ソフトウェアの動作状況の収集手法について得られた知見を述べる．最後に，忠実さに関しては，著者が携わったプロジェクトである模倣インターネットの構築の観点から，模倣インターネットの構築に寄与した仮想計算機を用いた検証環境の構築手法の部分で得られた知見を述べる．

1.5 本研究の工学的貢献

本研究では，検証を行う物が求める検証規模の検証を可能とするための仕組みの提供を目的とする．本研究における工学的な貢献を以下にあげる．

- 仮想計算機を用いた検証環境構築時の最適なメモリ資源割り当て手法の提案（３章）
  本研究では，仮想計算機を用いた検証環境を構築するときにそれぞれの仮想計算機にどの程度のメモリ量を割り当てればよいか，また，各計算資源にどのように仮想計算機を配置すればよいかを決定するアルゴリズムの提案を行っている．本研究では，現在のインターネットにおける経路交換で主に使われている BGP による経路交換及び経路決定ソフトウェアを例として提案アルゴリズムの妥当性の検証を行った．

- 大規模検証施設の機能の均一化による検証計画の簡易化手法の提案 (４章)
  本研究ではそれぞれの大規模検証施設において，異なる機能を均一化する手法の提案を行う．機能の均一化には両方の大規模検証施設において，管理ソフトウェアや独自機能を導入することで容易に達成できる一方，この方法では従来の大規模検証施設が使用している管理ソフトウェアとのアクセス制御の問題や，維持コストの増大等の課題が発生してしまう．そこで，Ditto subsystem と呼ぶ管理ソフトウェアに対するサブシステムを導入することで，従来の大規模検証施設の運用ポリシーやソフトウェアの変更なしに他のテストベッドの機能を追加可能にした．

- 異種ソフトウェアで管理された大規模検証環境の連携機構 (4 章)
  本研究では，異なる運用形態や資源管理ソフトウェアで制御されている大規模検証施設間の連携手法について提案を行う．本研究では，前機能の均一化と同様に大規模検証施設の資源管理ソフトウェアの変更を行うことなく連携を可能に手法の提案を行っている．本研究では，日本の NICT が有する大規模検証施設である StarBED とアメリカの南カリフォルニア大学の Information Sciencses Institute(ISI) が有する Deterlab の間にまたがって構築された検証環境の構築を行い，提案手法の妥当性の検証を行った．

1.6 本論文の構成

本論文は，序論，関連研究，提案手法と結論により構成される．提案手法は，第 I 部では大規模検証環境構築のための効率的な計算機への資源割り当ての手法と第 II 部では，大規模検証環境構築のための検証施設間の資源および機能の連携手法で構成されている．

第 2 章でシミュレーションおよびエミュレーションそれぞれにおける大規模検証の手法及び検証環境の構築手法について現状の関連研究をまとめる．

第 I 部では大規模検証環境構築のための効率的な計算機への資源割り当ての手法について提案を行う．第 3 章で，固定の計算機資源に対して構築可能な検証環境の規模を効率的に増加させる手法について述べる．

第 II 部では，大規模検証環境構築のための検証施設間の資源および機能の連携手法について提案を行う．第 4 章では，複数の大規模検証施設の資源を融通し合うことにより大規模な検証環境の構築するために必要な手法について，機能の均一化の点から提案手法を述べ，第 5 章にて資源の増加の点から提案手法を述べる．

第 6 章では本論文での成果を踏まえて検証環境の大規模化に関する課題と他の分野への応用について議論を行う．最後に 6 章にて本論文の結論を述べる．

2. 大規模な実験環境構築に関する現状と課題

本章では，これまで行われてきた検証手法や検証環境についてまとめ，1.4 で述べた課題についての現場についてまとめる．

2.1 シミュレーションによる検証手法

シミュレーションによる検証では，検証項目に特化したシミュレータと，汎用的に開発されたシミュレータの 2 種類が存在する．本節では，それぞれについてまとめる．

2.1.1 ネットワークに関するシミュレータ

検証する際に，検証目的を満たすためにシミュレータを自作することがある．これらは，検証の目的に特化されており，他の用途に使用することはできない．しかしながら，簡易な演算モデルの場合や，汎用的に開発されたシミュレータでは実装が難しいシミュレーションに関しては，効果的な手法である．

ネットワークに関するプロトコルや通信，デバイス等の動作のモデリングやシミュレーションを行うことを目的として汎用的に開発されたシミュレータがある．例えば，REAL [2], ns-1 [3], ns-2 [19], ns-3 [4], OPNET [1], OMNet++ [20], QualNet [21] や GloMoSim [22] がある．こられは，ライブラリやモジュールという形で自身の作成したプロトコルや通信方式，デバイスの挙動をシミュレータに組み込むことにより，任意のシミュレーションを実行することができるプラットフォームである．これらのシミュレータを使用したシミュレーションは様々行われておいる．しかし，これらのシミュレータは実オペレーティングシステムや実コードを動作させることはできない．ns-3 はシミュレーションの一部にエミュレーション環境と連携することにより実オペレーティングシステムや実コードを組み込むことは可能であるが，シミュレータ部分では他のシミュレータと同様に実オペレーティングシステムや実コードを動作させることはできない．

シミュレータは柔軟なネットワーク構成の変更や高い制御性や観測性，実験の再現性を要求するような検証をおこおなう際には有用な検証方法である．一方で，

実コードに関する検証や，コードを動作させるオペレーティングシステムの動作を含めた検証は向かない．

2.1.2 シミュレータの大規模化および高速化

シミュレータは事象をモデル化し，その数式化されたモデルを演算することで事象の結果を得る．シミュレータでモデル化する事象が有するパラメータの量や複雑さにより演算量が増加してしまうため，大量の CPU コアや大容量のメモリ，広帯域なバス帯域を持つようなスーパーコンピュータや CPU やメモリ，ストレージなどをネットワークにより接続することで計算を行うグリッドコンピュータなどを使用して計算量の増加に対応する方法がある．スーパーコンピュータは世界各国の大学や研究機関で構築されており，年に 2 度スーパーコンピュータの性能をランキングする TOP500 [23] が発表されている．TOP500 では 2017 年 11 月現在線形代数の数値演算のベンチマーク性能のランキングの他に，計算能力あたりの消費電力のランキングである Green500 やグラフの探査性能のランキングである Graph500 が発表されている．2017 年 11 月時点では中国の神威太湖之光がもっともよいスコアをだしている．一方日本では海洋研究開発機構の暁光 [24] が 4 位，理化学研究所の京 [25] が 10 位となっている．

SETI@home [26] や CERN の粒子加速器 LHC での実験データの解析のためのグリッドコンピュータである WorldWide HHC Computing Grid (WLCG) [27] がある．また，グリッドを構成するためのソフトウェアとしては，Globus [28] や Sun Grid Engine や Sun Grid Engine から発展した Open Grid Scheduler [29] や Son of Grid Scheduler [30] がある．

これらスーパーコンピュータやグリッドコンピュータでは，演算をより効率的に行うために，スケジューリングや資源割り当てについて様々な研究が行われている．Globus [28, 31] では，各コンピュータに Globus Resource Allocation Manger (GRAM) を配置し，それぞれの GRAM で管理されている資源の空き状況をリソースブローカーに渡し管理することで演算をどの GRAM で管理されたコンピュータに渡すかを決定している．Condor-G [32] では，Condor-G scheduler に入力された演算ジョブを Condor-G GridManager 経由で遠隔の Globus Gatekeeper に送る．

Globus Gatekeeper は，入力された演算ジョブを計算機資源に空きがある Globus JobManager に転送し，それぞれのコンピュータで演算が行われる．Gridsim [33] は Java や，cJVM 上でグリッドのシミュレーションを行うことができるプラットフォームである．Gridsim はブローカーやスケジューラを持ち，またツールキットとして job manager や resource allocation の機能を持つ．

スーパーコンピュータやグリッドコンピュータでは，上記のように演算のためのスケジューリングや資源割り当てを行うソフトウェアがさまざま開発されている．しかし，演算を目的としたスケジューリングや資源割り当ての方法や実ソフトウェアを動作させるための検証のためのスケジューリングや資源割り当てにそのまま適用することは困難である．

2.2 エミュレーションによる検証手法

2.2.1 エミュレーション環境構築の補助ツール

設計したプロトコルや開発したソフトウェアの検証を行うために，エミュレーションによる検証が行われる．エミュレーションでは，複数の PC にインストールしたオペレーティングシステム (OS) 上に検証対象のプロトコルが動作するソフトウェアや開発したソフトウェアをインストールし，動作させることで性能や動作の確認を行う．本節では，エミュレーションを行うために提案された手法やエミュレーションによる検証を目的とした検証施設についてまとめる．

ソフトウェアの検証を行う際に，手持ちや大学や研究室に存在するコンピュータを集めて検証環境を構築する場合がある．この場合，コンピュータの数は数台から多くても十数台程度の規模であることが多い．環境構築を行う際は，検証者自身がコンピュータやネットワークの設定を行う必要がある．設定を行う補助ツールとして [34–37] がある．これらは，オペレーティングシステムへのソフトウェアのインストールや設定の投入の他にネットワークの構成や機器の設定の投入が可能なものもある．検証者自身の独自ツールやこれらのツールを用いて構築された環境は検証終了後に分割され，個々のコンピュータとして利用される．

一方で，[38–45] はネットワークの検証を目的として汎用的に検証を行うこと

ができるエミュレーションプラットフォームである．これらは，ネットワークに関するソフトウェアやプロトコルを実装したものを検証するために必要な機能を実装している．これらは，検証者が検証に用いるネットワークトポロジをそれぞれのソフトウェアが求める形式で記述することにより，PC サーバとネットワークスイッチの構成に任意のネットワークを再現するために必要な機能を有している．また，一部は PC サーバやネットワーク機器の設定投入を補助するものもある．これらのうち，[39–41] に関しては，仮想化技術を用いて 1 台の PC サーバ上に複数のノードを多重化して配置を可能としている．これらは検証者が集めた PC サーバやネットワークスイッチでの使用も可能である．一方で，これらのソフトウェアを用いて検証者に貸し出すことを前提とした大規模な検証施設を構築および運用が行われている．

ネットワークに関するプロトコルやソフトウェアの検証を目的とした検証施設は，その目的に応じて構築されている．ネットワークエミュレーションを目的としたテストベッドとして，PlanetLab [8], StarBED [7], Emulab [46], Deterlab [6] がある．これらは，PC サーバと PC サーバ間のネットワークを提供する検証施設である．それぞれの検証施設では，資源の管理ソフトウェアを用意しており，そのソフトウェアを使用して資源の設定や制御を行う．StarBED, Emulab, Deterlab は PC サーバ自体を利用者に貸し出す方式を取っている一方，PlanetLab は PC サーバ上に複数の仮想化されたノードを起動し，その一部を利用者に貸し出す方法を取っている．また，StarBED, Emulab, Deterlab は施設内で閉じたネットワーク内を有線接続で検証のための通信を行う一方，PlanetLab はそれぞれのノードがインターネットに接続しており，検証用の通信はインターネットを介して行われる．

上記は汎用 PC サーバ上に環境を構築することを目的として構築された検証施設である．一方で，Internet of Things (IoT) 機器やセンサー機器などの汎用 PC 上での動作が想定されていない機器で動作するソフトウェアやプロトコルの検証を目的とした検証施設が構築されている．MoteLab [47], Kansei [48], Indriya [49] や TWIST [50] がある．これらは，センサーデバイスを閉空間の中に設置されており，利用者は外部からセンサーデバイスに接続して検証を行う．これらの検証施設では，それぞれのセンサーデバイスへの接続や設定を行うためのインターフェース

を開発している．また，それぞれのセンサーデバイスの相互通信は無線通信で行われる．WISEBED や Kansei は連携のための API を有しており，他の検証施設との連携した検証環境構築が想定されている．

クラウドコンピューティングでは，[34,36,51–53] などのオーケストレーションに使用されるツールの開発が行われている．これらのツールは多くの場合データセンターのサーバやネットワークスイッチ，ストレージなどに対して設定の投入や監視，観測を行う機能を提供している．これらのツールの機能の一部はエミュレーションの補助ツールと同じであるが，クラウドの場合はエミュレーションのためのツールと違い，検証の再現性や検証ごとの構成変更がないという点で，機能が異なっている．また，エミュレーションツールの場合は多くの場合ネットワークトポロジやそれぞれのノードの動作がシナリオファイルとして入力されるが，クラウドのオーケストレーションツールでは，インストールするソフトウェアやネットワーク機器へのパラメータ等が入力となる違いがある．

2.2.2 エミュレーションの規模拡大手法

SFA は実験ノードをスライスと定義し，スライスの貸し出しとスライスを接続するネットワークを貸し出すことを目的としたテストベッド連携アーキテクチャである．SFA は PlanetLab [8]，Onelab，Panlab や GENI で採用されている．SFA では，ユーザは中央サーバ (Centralized Federator) にノード情報と実験トポロジを入力すると，実験にトポロジに合わせて実験環境が構築される．SFA は利点として，MyPLC [54] と呼ばれるコントローラを用いて連携する方法が確立されている．MyPLC は PlanetLab や GENI で使われており，これらのテストベッドと連携する場合は MyPLC を利用すれば良い．SFA を導入する場合，テストベッドのシステム構築形態が Slice-base の管理形態に固定されるため，現在他のシステム構築形態でテストベッドを運用している場合は変更が必要になる．

DFA は Emulab や Deterlab といった Emulab-base の検証施設を連携することを想定したアーキテクチャである．DFA はプラグインを用いてそれぞれの検証施設の運用方針を変更せずにテストベッド連携を行うことを目的としている．ユーザは中央サーバである Experiment Controller (EC) に tcl で記述された実験トポロジ

を入力する．EC はユーザが入力した実験トポロジを topodl という中間言語に変換し，各テストベッドにある Access Controller (AC) に変換した情報を渡す．AC では topodl に変換された情報もとに，検証施設内に実験環境を構築する．DFA は利点として，AC を検証施設側が実装するため，各検証施設の運用形態に合わせた検証施設間の連携を可能にしている．一方，Emulab-tools を用いた検証施設を想定している．そのため，テストベッドのシステム形態は Emulab-tools に限定されてしまう．

CFA は API を基本としたテストベッド連携アーキテクチャであり，CFA は各テストベッドが独自の API を提供することでテストベッド連携を行なっている．CFA はユーザに対してテストベッド間で共通する API に関しては RESTful の共通 API (TA API) を定め，中央サーバを介して各テストベッドへのインターフェースを提供している．ユーザは提供されたインターフェースを利用して各テストベッドのリソースの操作を行う．CFA の利点は CFA は API を提供するのみのアーキテクチャのため，各テストベッドの自治が可能であることである．一方，ユーザが共通 API に定義されていないテストベッド固有の資源や機能を使う場合，ユーザは個々のテストベッドが提供する API を直接使用しなければならない．そのため，CFA を採用する場合，共通 API と固有の API の開発を行う必要がある．

2.3 まとめ

検証においては，検証項目や目的によって様々な手法をとることができる．シミュレーションは，設計段階のプロトコルやソフトウェアに関して，モデルによる検証に有利である．シミュレーションでは，柔軟な構成がネットワークトポロジの数に依存しない台数の PC サーバで検証を行うことができる．また，モデルが複雑な場合は，計算量が多くなりシミュレーションシナリオで定義した時間の結果を得るために，それよりも多くの実時間を要してしまう．この点に関しては，スーパーコンピュータやグリッドコンピュータなどにより，現在解決がなされている．一方で，実装されたソフトウェアやプロトコルに関してはエミュレーションにより検証が行われている．エミュレーションは検証に用いる構成によって使用する PC サーバやネットワーク機器の台数が異なってくるため，シミュレーショ

ンほど柔軟な構成での検証を行うことができない．しかし，シミュレーションでは検証することができない，実コードの検証を行うことができる．また，エミュレーションは実時間での動作となるため，検証終了時間がエミュレーションのシナリオに依存しない．しかし，検証に用いるトポロジが大きくなるとそれに比例して用意しなければならない PC サーバやネットワーク機器の台数が多くなってしまう．これは，大規模な検証施設の構築や仮想化技術によりノードを 1 台の PC サーバに多重化することで現在解決がなされている．

本研究では，実コードの検証のための大規模な検証環境の構築を目的としているが，これまではさまざまな検証が行われてきている．しかしながら，それぞれの解決手法のみでは，現在最大規模の実動作環境であるインターネットの規模には及ばない．また，仮想化技術により 1 台の PC サーバ上に複数の検証ノードを配置するプラットフォームは存在しているが，これらは，検証ノードにどの資源をどの程度割り当てるかについては，検証者に一任されている．検証を安定的に行うことを考えると割り当てを過剰にせざるを得ず，割り当てを小さくしすぎると検証環境に安定さがなくなってしまうことが考えられる．そこで，本研究では，スーパーコンピュータやグリッドコンピュータで研究が行われてきた資源配置をエミュレーション環境構築時の仮想計算機に対する資源割り当てとして考え，さらに大規模検証施設の連携の 2 つの手法の組み合わせにより，これまでより大規模な検証環境の構築を目指す．本研究が目指すべき大規模検証環境は大規模検証環境同士が連携し，複数の大規模検証環境にまたがった PC サーバとネットワーク機器で構成された検証環境上に，資源割り当て手法により効率的に配置された仮想計算機により安定的な検証環境を構築することである．

第 **I** 部

大規模検証環境構築のための効率的な計算機への資源割り当て

3. 大規模模擬ネットワーク構築のための仮想計算機へのメモリ割り当てアルゴリズム

3.1 概要

ネットワークを利用したソフトウェアや技術はますます大規模化している．これらの検証を効率よく行うために大規模サーバクラスタであるテストベッドが開発されているが，サーバの台数はネットワークを利用したソフトウェアや技術が動作する環境よりも小さな規模での検証しか行うことができない. 本節では，はじめに経路制御ソフトウェアに関して割り当てが必要な資源の洗い出しを行い，その中で検証環境構築のために割り当てが必要な資源について選定を行った上で，選定した資源について資源割り当てアルゴリズムの提案を行う．

3.2 定式化する資源の決定

経路制御のプロトコルには，Border Gateway Protocol (BGP) [55] や Open Shortest Path First (OSPF) [56]，Routing Information Protocol (RIP) [57] や Intermediate System to Intermediate System (IS–IS) [58] がある．これらの経路制御プロトコルでは，経路情報をルータ間で交換し，パケットの転送に関して最適な経路を決定するプロトコルである．そのため，これらの経路制御プロトコルは大きく，交換した経路制御情報と決定した経路情報を保持することなる．本研究では，これら経路制御ソフトウェアの中で，インターネット上で外部の組織と経路情報を交換する際に用いられる BGP について，事前実験として仮想計算機を用いた場合に

資源割り当てが必要であると考えられる資源の洗い出しを行った．事前実験として，260 台の物理計算機上で Xen による 10000 台の仮想計算機の起動による BGP ネットワークの構築を行った．本事前実験により，得られたエラーメッセージをもとに，仮想計算機への資源の割り当て不足が発生した資源についてまとめる．ここで，Xen の特権ドメインを dom0，それ以外のドメインを domU とする．

netfront: rx->offset: 0

Xen の domU のネットワークバッファの枯渇によるエラー

Neighbour table overflow

Xen の domU の arp キャッシュに保存数するエントリ数が規定値を超えたことを示すエラー

No buffer space available

Xen の domU の送受信バッファの枯渇によるエラー

TCP: Treason uncloaked

短時間でパケットが大量を大量に受信したため，Xen の domU カーネルが Denial of Service (DoS) 攻撃と判定したことによるエラー

Can't make socket : Too many open files

Xen の domU で使用可能なファイル記述子の上限を超えてファイル記述子を使用したことによるエラー

Out of Memory: Kill process

Xen の domU に割り当てたメモリサイズ以上のメモリを，アプリケーションが確保しようとしたことによるエラー

この事前実験により得られた結果から，定式化する資源は，メモリとネットワークパケット数，ネットワークトラフィック，アプリケーションが使用するソケット数とファイル記述子 の 5 点であると考えられる．各資源でノード数を決定するた

めの目的関数と仮想ノードの配置を行うための目的関数の 2 種類の目的関数を考える．まず始めにメモリに関して目的関数と制約条件の設定を行う．次に，ネットワーク関連の資源の定式化を行う．ネットワーク関連の資源はソケット数，ファイル記述子, ネットワークトラフィックとパケット数である．はじめに，ソケット数とファイル記述子の目的関数と制約条件を設定し，次にネットワークトラフィック，パケット数の目的関数と制約条件の設定を行う．最後に設定した目的関数の組み合わせを考える．また，条件として，定式化の際は仮想マシンのオーバヘッドはないものとして考える．これは，仮想マシンのアーキテクチャによってオーバヘッドが発生するものが異なるため，一般化が難しいためである．まず始めにメモリについて考える．

3.3 消費メモリの定式化

まず始めに物理ノード数または仮想ノード数を決定するための目的関数を考える．この目的関数としては，実験に使用する物理ノード数が決まっている場合，決まっている物理ノード数に配置できる最大数の仮想ノード数を求める場合と，実験に使用する仮想ノード数が決まっている場合に，その仮想ノードを配置するために必要な最小の物理ノード数の 2 つが考えられる．物理ノード数から最大の仮想ノード数を求める目的関数は，実験に使える物理ノードで行える最大規模が知りたいときに用いられ，仮想ノード数から最小の物理ノード数を求める場合は物理ノードを有償で借りる場合に，できるだけコストを抑えて実験を行いたい場合に用いられる．

次に仮想ノードの配置を決定するための目的関数を考える．この目的関数として，各物理ノード上の仮想ノードのメモリの合計の分散が最小になるような仮想ノードの配置が考えられる．ソフトウェアの負荷と消費メモリが関係するようなソフトウェアの場合は，消費メモリを均一にすることにより物理ノード全体の負荷を均一にすることができる．

以上より，物理ノードが持つメモリ量を考慮に入れ仮想ノードを物理ノードへ配置する場合，目的関数は以下の 3 つが考えられる．

物理ノード数または仮想ノード数を決定するための目的関数

目的関数 1 与えられた仮想ノードを配置するために必要な物理ノードの最小の数を考える．

目的関数 2 与えられた物理ノードに配置される最大の仮想ノードの数を考える．

仮想ノードの配置を決定するための目的関数

目的関数 3 各物理ノード上の仮想ノードのメモリの合計の分散が最小になるような仮想ノードの配置

これらは，整数計画問題のひとつである，ビンパッキング問題としてとらえることができる．ビンパッキング問題とは，様々な大きさの荷物をある大きさの瓶につめる場合に，最小となる瓶の数を決定する問題であり，一般に NP-困難な問題として知られている．メモリ資源割り当ての場合は，様々な大きさの荷物が仮想ノードに当たり，瓶が物理ノードと考えることができるため，ビンパッキング問題に関する定式化の様式に合わせてメモリ資源について定式化を行う．

3.3.1 OS が必要とするメモリ量

はじめに，OS が必要とするメモリ量を考える．この OS では，n 種類のアプリケーションが動作する可能性があり，アプリケーション i $(1 \leq i \leq n)$ が消費するメモリ量を M_{app_i} とする．ただし，アプリケーションが動作していない場合は，$M_{app} = 0$ とする．このとき，OS のシステムが必要とするメモリ量を M_{sys} とすると OS 全体として必要となるメモリ量 M_{OS} は，

$$M_{OS} = M_{sys} + \sum_{i=1}^{n} M_{app_i} \tag{1}$$

となる．

3.3.2 仮想計算機を利用した場合の必要メモリ量

次に仮想化技術を用いて OS を多重化した場合を考える．ここで，物理ノードには 1 台のホストノードと複数台の仮想ノードが動作しているとする．ただし，物理ノード上の仮想ノードの数は制限ないものとする．まずホストノードが必要とするメモリ量を考える．ホストノードでは，n 種類のソフトウェアが動作する可能性があり，アプリケーション i $(1 \leq i \leq n)$ が消費するメモリ量を M_{app_i} とする．ただし，アプリケーションがホストノード上で動いていない場合，$M_{app} = 0$ とする．

このとき，ホストノードのシステムが必要とするメモリ量を M_{H_sys} とすると，ホストノード全体が必要とするメモリ量 M_H は以下のように定式化される．

$$M_H = M_{H_sys} + \sum_{i=1}^{n} M_{app_i} \tag{2}$$

仮想ノードでも，n 種類のソフトウェアが動作する可能性がある．アプリケーション i が消費するメモリ量を M_{app_i} とし，仮想ノードのシステムが必要とするメモリ量を $M_{G:sys}$ とすると，仮想ノードが必要とするメモリ量 M_G は以下のように定式化される．

$$M_G = M_{G:sys} + \sum_{i=1}^{n} M_{app_i} \tag{3}$$

以上より，仮想化技術を用いて OS を多重化した場合，物理ノードが必要とするメモリ量 M_P は，物理ノード上に r 台の仮想 OS が動作しているとすると，

$$M_P = M_H + \sum_{l=1}^{r} M_{Gl} \tag{4}$$

となる．ただし，ホストノードと割り当てられた仮想 OS のメモリ値の合計は物理ノードが持つメモリ値 $M_{P_{max}}$ を越えることはできない．よって

$$M_{Pmax} \geq M_P \tag{5}$$

を満たさなければならない．

3.3.3 目的関数の定式化

以上の式を用いて目的関数 1 から 3 の定式化を行う．

仮想ノードの配置に必要な最小の物理ノード数　x_{jk} は物理ノード k $(1 \leq k \leq p)$ に仮想ノード j $(1 \leq j \leq q)$ が所属しているかどうかの判定，y_k は物理ノード k を使用するかしないかの判定に用いられる変数とし，物理ノードに配置する仮想ノード数 q は定数で与えられた物理ノード数 p の中で使用された物理ノードの数 $N_{P_{used}}$ を最小にするような割り当て方法を考える．

物理ノード k に必要なメモリ値を M_{P_k}，物理ノード k が持つ物理メモリ値を $M_{P_max_k}$ とし，この条件を定式化すると，

$$minimize\ N_{P_{used}} = \sum_{k=1}^{p} y_k \tag{6}$$

$$subject\,to:$$

$$M_{H_k} + \sum_{j=1}^{q} x_{jk} M_{G_j} = M_{P_k} \leq M_{P_max_k} y_k \tag{7}$$

$$\sum_{k=1}^{p} x_{jk} = 1 \tag{8}$$

$$x_{jk} \in \{0,1\} \tag{9}$$

$$y_k \in \{0,1\} \tag{10}$$

となる．

物理ノードに配置可能な最大の仮想ノード数　物理ノード数 p を定数とし，物理ノードに配置する仮想ノード数 q を最大にするような割り当て方法を考える．この時，物理ノードに配置された仮想ノード数を $N_{G_{used}}$ とし，この条件を定式化すると，

$$maximize\ N_{G_{used}} = \sum_{j=1}^{q} x_{jk} \tag{11}$$

$$subject\,to:$$

$$M_{Hk} + \sum_{j=1}^{q} x_{jk} M_{Gj} = M_{Pk} \leq M_{P_max_k} \tag{12}$$

$$\sum_{k=1}^{p} x_{jk} = 1 \tag{13}$$

$$x_{jk} \in \{0, 1\} \tag{14}$$

となる．

物理ノードの消費メモリ量のメモリ値の均一化　物理ノード数 p，仮想ノード数 q ともに定数とし，各物理ノードに割り当てられた仮想 OS のメモリ値の合計の分散が最小になるような配置を考える．仮想マシンに仮想ノードを配置する場合，配置できる仮想ノード数は loopback デバイスの数に依存してしまう．そこで，loopback デバイスから発生する配置可能な仮想ノードの上限数を Max_{Guest} とする．この時，各物理ノードに割り当てられた仮想 OS のメモリ値の合計の平均を $\overline{M_P}$ とし，定式化すると

$$minimize\ \frac{1}{p} \sum_{k=1}^{p} \left(\overline{M_P} - M_{Pk} \right)^2 \tag{15}$$

$$subject\,to:$$

$$M_{Hk} + \sum_{j=1}^{q} x_{jk} M_{Gj} = M_{Pk} \leq M_{Pmaxk} \tag{16}$$

$$\sum_{k=1}^{p} x_{jk} = 1 \tag{17}$$

$$\sum_{j=1}^{q} x_{jk} \leq Max_{Guest} \tag{18}$$

$$x_{jk} \in \{0, 1\} \tag{19}$$

となる．

3.4 ネットワーク関連の資源の定式化

物理ノード上で多くの仮想ノードを動作させる場合，仮想ノード上の個々のアプリケーションが送受信するパケット数は問題ないが，物理ノード上の仮想ノー

ドの数が多くなれば，資源はすべての仮想ノードで共有することになるので，個々の仮想ノードが使用できる資源は制限されてしまう，また，送受信するパケット数が実験規模やトポロジの大きさに依存する場合は，物理ノードの配置は実験規模やトポロジの大きさによって変化する．この章では，OS が送受信するパケット数と通信量を考慮に入れた仮想ノードの配置を行うための目的関数と制約条件の設定を行い，定式化する．

3.4.1 ソケット数による制約条件

アプリケーションが動作するときに使用するソケットについて考える．OS 上では n 種類のアプリケーションが動作する可能性がある．アプリケーション $i\,(1 \leq i \leq n)$ が消費するソケットを So_i とすると，すべてのアプリケーションで使用するソケット数の合計は，OS 上で使用可能なソケット数を S_{max} とすると，

$$\sum_{i=0}^{n} x_{ij} So_i \leq S_{max} \tag{20}$$

$$\sum_{j=1}^{S} x_{ij} = 1 \tag{21}$$

$$x_{ij} \in \{0, 1\} \tag{22}$$

となる．よって，この条件を越える場合はカーネルのチューニングを行う必要がある．

3.4.2 ファイル記述子の制約条件

アプリケーションが動作するときに使用するファイル記述子について考える．ファイル記述子はアプリケーション毎にもてるため，アプリケーション i が消費するファイル記述子数 FD_i がカーネルで定義されているのファイル記述子の最大値 FD_{max} を越えなければ良い．よって，ファイル記述子に関する制約条件は，

$$\sum_{i=1}^{n} FD_i \leq FD_{max} \tag{23}$$

となる．この条件を満たさない場合はカーネルのチューニングおよびアプリケーションの再コンパイルが必要となる．

3.4.3 送受信パケット数の定式化

まず始めに，物理ノード数または仮想ノード数を決定するための目的関数を考える．この目的関数としては，実験に使用する物理ノード数が決まっている場合，実験環境においてパケット破棄が起きないで配置できる仮想ノードの最大数を求めるための目的関数が考えられる．この目的関数は，パケット破棄が起きない実験環境のなかで最大の実験規模を求めたいときに用いられる．

次に仮想ノードの配置を決定する目的関数について考える．この目的関数として，スイッチが処理するパケット数を最小化させる配置と時刻 t にスイッチの各ポートが送受信処理するパケット数が均一になるような配置が考えられる．スイッチが処理するパケット数の最小化は，物理ノードを割り当てる際に物理ネットワークやスイッチに与える影響を最小にする配置を求め，時刻 t にスイッチの各ポートが送受信処理するパケット数が均一になるような配置は，各ポートで処理するパケット数を均一化することによってスイッチのあるポートに負荷が偏らないような配置を求めることになる．

以上より，送受信のパケット数は以下の 3 つの目的関数が考えられる．

物理ノード数または仮想ノード数を決定するための目的関数

目的関数 4 パケット破棄を起こさない範囲で物理ノードに配置できる仮想ノードの最大数

仮想ノードの配置を決定する目的関数

目的関数 5 スイッチが処理するパケット数の最小化

目的関数 6 時刻 t にスイッチの各ポートで送受信処理されるパケット数の分散が最小になるような配置

メモリと同様に，パケット数も送受信パケット数を荷物，送受信キューの数を瓶としたビンパッキング問題ととらえることができる．よって，以下ではビンパッキング問題の定式化を行う．

OS が送信するパケット数　はじめに，OS から出されるパケット数について考える．OS では，ネットワークにパケットを出すアプリケーションが n 種類動く可能性がある．アプリケーション i が時刻 t に $P_{Sapp_{i}t}$ パケット送信するとする．このとき，時刻 t に OS で出されるパケット数の合計値 P_{St} は

$$P_{St} = \sum_{i=1}^{n} P_{Sapp_{i}t} \tag{24}$$

となる．

次にソケットについて考える．

送信に用いる TCP ポートを $port_s$ $(1 \leq s \leq 65535)$ とするパケット数 $P_{Sport_{s}t}$ は，y_e をパケット $P_{Sport_{s}t,k}$ の送信ポートが $port_s$ の場合に 1 それ以外の場合に 0 とすると，時刻 t に $port_s$ が送信するパケット数 $P_{Sport_{s}t}$ は

$$P_{Sport_{s}t} = \sum_{e=1}^{P_{Sapp_{i}t}} y_e P_{Sapp_{i}t,e} \tag{25}$$

$$y_e \in \{0, 1\} \tag{26}$$

となる．TCP ポートとソケットは 1 : 1 で対応するため，ソケット l から送信されるパケット数は bfnd された $port_s$ からパケット数に等しい．よって，時刻 t にソケット l から送信されるパケット数 $P_{St,l}$ は

$$P_{St,l} = P_{Sport_{s}t} \tag{27}$$

となる．

OS がスイッチに送信するパケット数　次に，各 OS が接続しているスイッチに送信するパケット数を考える．

ネットワークエミュレーションテストベッドにおいて，物理ノードとホストノードの場合は各スイッチのイーサネットポートに接続している OS は 1 台である．しかし，仮想ノードの場合，1 つのイーサネットポートに複数の仮想ノードが接続している．よって，物理ノードとホストノードの場合と仮想ノードの場合について考える．

物理ノード，ホストノードの場合　物理ノードまたはホストノード k が接続しているスイッチのイーサネットポートを m とする．

物理ノードまたはホストノード k では，送信パケットの宛先が自分自身以外の場合，スイッチによるパケットの転送処理が行われる．x_e をパケット $P_{St,e}$ $(1 \leq e \leq P_{St})$ の宛先が自分以外の場合は 1，宛先が自身の場合は 0 とすると，時刻 t に物理ノードまたはホストノード k が繋がっているスイッチのイーサネットポート m に送信するパケット数を $P_{St,k,m}$ は

$$P_{St,k,m} = \sum_{e=1}^{P_{St}} x_e P_{St,e} \quad (28)$$

$$x_e \in \{0, 1\} \quad (29)$$

となる．

仮想ノードの場合　仮想ノード j が接続しているスイッチのイーサネットポートを m とする．

仮想ノード j では，送信パケットの宛先が自分以外かつ自身が所属している物理ノード上の仮想ノードではない場合にスイッチにパケットを転送する．y_e はパケットの宛先が自身を含め自身と同じ物理ノードに所属していない場合に 1, 自身と同じ物理ノードに所属している場合は 0 とすると，時刻 t に仮想ノード j が繋がっているスイッチのイーサネットポート m に送信するパケット数 $P_{St,k,m}$ は

$$P_{St,k,m} = \sum_{e=1}^{P_{St}} y_e P_{St,e} \tag{30}$$

$$y_e \in \{0,1\} \tag{31}$$

となる．

OS が受信するパケット数　物理ノードやホストノードと仮想ノードはネットワークが切り離されているため相互に通信はできない．また，物理ノード 1 台に 1 つのホストノードしか動かないため，物理ノードとホストノードが受信するパケット数は同様に考えることができる．以上より，OS が受信するパケット数を物理ノードとホストノードの場合と仮想ノードの場合に分けて考える．

物理ノードまたはホストノードの場合　ある時刻 t に OSk' が受信するパケットは，管理ネットワークに接続している p 台の物理ノードまたはホストノードが出すパケットの中で，宛先が k' となっているパケットの合計数である．x_e をパケット $P_{t,k}$ の宛先が k' の場合に 1，それ以外の場合は 0 とすると，k' が受信するパケット数を $P_{Rt,k}$ は，

$$P_{Rt,k'} = \sum_{k=1}^{p} \sum_{e=1}^{P_{Stk}} x_e P_{St,k,e} \tag{32}$$

となる．

仮想ノードの場合　ある時刻 t に仮想ノード j' が受信するパケットは，実験ネットワークに接続している q 台の仮想ノードが出すパケットの中で，宛先が j' となっているパケットの合計数である．x_e をパケット $P_{t,j}$ の宛先が j' の場合に 1，それ以外の場合は 0 とすると，j' が受信するパケット数を $P_{Rt,j'}$ は以下のようになる．

$$P_{Rt,j'} = \sum_{j=1}^{q} \sum_{e=1}^{P_{Stj}} x_e P_{St,j,e} \tag{33}$$

パケット破棄の可能性

OS でパケット破棄が起きる条件　OS では，アプリケーションから出されたパケットは，いったん送信キューに入れられ，順次送信処理が行われる．しかし，プロセッサーの送信処理の性能よりアプリケーションが出すパケット数の方が大きければ送信キューが埋まってしまい，パケットロスが発生してしまう．1 秒間にプロセッサーが処理できる送信キューの数を K_S とし，OS 全体で使用されているソケットの数を So_{used} とすると，時刻 t にソケット l の送信キューで処理されるパケットの数 $K_{St,l}$ は

$$K_{St,l} = \frac{K_S}{So_{usedt}} \tag{34}$$

となる．

次に，時刻 t でソケット l の送信キューに残っているパケット数について考える．

時刻 $t-1$ にソケット l の送信キューに残っているパケット数は，$t-1$ までに送信要求が出されたパケット数から $t-1$ までにソケット l の送信キューで処理可能なパケット数を引いたものである．よって，この値に時刻 t で新たに送信要求が出されたパケット数を加え，時刻 t にソケット l で処理できるパケット数 $K_{St,l}$ を引くことにより時刻 t の時点でソケット l の送信キューに残るパケット数を求めることができる．よって，時刻 t にソケット l の送信キューに残るパケット数を $P_{Restt,l}$ とすると，

$$P_{Restt,l} = \sum_{T=1}^{t} \left(P_{Sl,T} + K_{Sl,T} \right) \tag{35}$$

$$\text{もし},\, P_{Sl,T} + K_{Sl,T} < 0 \text{ ならば } P_{Sl,T} + K_{Sl,T} = 0 \tag{36}$$

もしくは

$$P_{Restt,l} = P_{Restt-1} - K_{St,l}t + P_{St,l} \tag{37}$$

で表される．

ここで，パケット破棄を起こさないためには，時刻tにソケットlで，送信キューに残っているパケット数が送信キューの数L_{Sl}を越えなければ良い．よって，時刻tの送信キューの状態を，時刻tで送信キューに残っているパケット数から以下のように定義する．

$$P_{Restt,l} = 0 \Leftrightarrow \phi \tag{38}$$

$$0 < P_{Restt,l} < L_{Sl} \quad \Leftrightarrow \text{パケット破棄は起きない} \tag{39}$$

$$L_{Sl} < P_{Restt,l} \quad \Leftrightarrow \text{パケット破棄が発生} \tag{40}$$

受信でも送信時と同様に，受信キューにパケットを入れ順次処理を行う．1秒間にソケットlでプロセッサーが処理できる受信キューの数をK_{Rl}とすると，ソケット$So_{m,t}$の受信キューで処理されるパケットの数$K_{Rt,l}$は

$$K_{Rt,l} = \frac{K_R}{So_{used_t}} \tag{41}$$

となる．

次に，時刻tでソケットlの受信キューに残っているパケット数について考える．

時刻$t-1$にソケットlの受信キューに残っているパケット数は，$t-1$までに他のノードから送られてきたパケット数から$t-1$までにlの受信キューで処理可能なパケット数を引いたものである．よって，この値に，時刻tで新たに他のノードから送られてきたパケット数を加え，時刻tにlの受信キューで処理できるパケット数K_Sを引くことにより，時刻tの時点でソケットlの受信キューに残るパケット数を求めることができる．よって，時刻tにソケットlの受信キューに残るパケット数を$P_{Restt,l}$とすると，

$$P_{Restt} = \sum_{T=1}^{t} \left(P_{Rl,T} - K_{Rl,T} \right) \tag{42}$$

$$\text{もし } P_{Rl,T} - K_{Rl,T} \text{ならば } P_{Rl,T} - K_{Rl,T} = 0 \tag{43}$$

もしくは

$$P_{Restt} = P_{Restt-1} - K_S t + P_{Rl,t} \tag{44}$$

と表すことができる．

パケット破棄を起こさないためには，時刻 t にソケット l で，受信キューに残っているパケット数が受信キューの数 L_{Rl} を越えなければ良い．よって，時刻 t の受信キューの状態を，t で受信キューに残っているパケット数から以下のように定義する．

$$P_{Restt,l} = 0 \Leftrightarrow \phi \tag{45}$$

$$0 < P_{Restt,l} < L_{Rl} \quad \Leftrightarrow \text{パケット破棄は起きない} \tag{46}$$

$$L_{Rl} < P_{Restt,l} \quad \Leftrightarrow \text{パケット破棄が発生} \tag{47}$$

となる．

スイッチでパケット破棄が起きる条件　次に物理ノードが接続しているスイッチのパケット破棄の可能性について考える．

物理ノードとホストノードの場合，送信パケットは宛先が自分自身でない場合は，必ずスイッチによる転送処理が行われる．

スイッチが 1 秒間に処理可能な受信パケット数を K_R とし，時刻 t に使用しているイーサネットポート数を EP_{usedt}，各受信キューで処理可能なパケット数を $K_{Rt,m}$ とすると，

$$K_{Rt,m} = \frac{K_R}{EP_{usedt}} \tag{48}$$

となる．

時刻 t でスイッチのイーサネットポート m の受信キューに残っているパケット数について考える．

時刻 $t-1$ にイーサネットポート m の受信キューに残っているパケット数は，$t-1$ までに他のノードから送られてきたパケット数から $t-1$ までに m の受信キューで処理可能なパケット数を引いたものである．よって，この値に時刻 t で m が接続しているノードから送られてきたパケット数を加え，時刻 t に m の受信キューで処理できるパケット数 K_R を引くことにより，時刻 t の時点で m の受信キューに残るパケット数を求めることができる．

時刻 t に物理ノードまたはホストノードの k と接続しているスイッチのイーサネットポート m が受信するパケット数 $P_{Rm,t}$ は，物理ノードまたはホストノードがスイッチに送信するパケット数であり，式 (28) に等しい．しかし，仮想ノードの場合は，物理ノード上で起動している仮想ノードが出すパケットの合計値がスイッチに送信されるパケット数であり，式 (30) に等しくなる．以上より，時刻 t に m の受信キューに残るパケット数を $P_{Restt,m}$ とすると，

$$P_{Restt,m} = \sum_{T=1}^{t} \left(P_{Rm,T} - K_{Rm,T} \right) \tag{49}$$

$$\text{もし } P_{Rm,T} - K_{Rm,T} < 0 \text{ ならば } P_{Rm,T} - K_{Rm,T} = 0 \tag{50}$$

もしくは

$$P_{Restt,m} = P_{Restt-1} - K_{Rm}t + P_{Rm,t} \tag{51}$$

となる．

パケット破棄を起こさないためには，時刻 t に m で，受信キューに残っているパケット数が受信キューの数 L_{Sm} を越えなければ良い．よって，時刻 t の受信キューの状態を，t で受信キューに残っているパケット数から以下のように定義する．

$$P_{Restt,m} = 0 \Leftrightarrow \phi \tag{52}$$

$$0 < P_{Restt,m} < L_{Sm} \Leftrightarrow \text{パケット破棄は起きない} \tag{53}$$

$$L_{Sm} < P_{Restt,m} \Leftrightarrow \text{パケット破棄が発生} \tag{54}$$

となる．

時刻 t でスイッチのイーサネットポート m の送信キューに残っているパケット数について考える．

時刻 $t-1$ にイーサネットポート m の送信キューに残っているパケット数は，$t-1$ までに m に接続しているノードが宛先となるパケット数から $t-1$ までに m の送信キューで処理可能なパケット数を引いたものである．よって，この値に時刻 t で m に接続しているノードが宛先となるパケット数を加え，時刻 t に m の送信キューで処理できるパケット数 K_S を引くことにより，時刻 t の時点で m の送信キューに残るパケット数を求めることができる．よって，時刻 t に m の送信キューに残るパケット数を $P_{Restt,m}$ とすると，

$$P_{Restt,m} = \sum_{T=1}^{t} P_{Sm,T} - K_{Sm,T} \tag{55}$$

$$\text{もし } P_{Sm,T} - K_{Sm,T} < 0 \text{ ならば } P_{Sm,T} - K_{Sm,T} = 0 \tag{56}$$

もしくは

$$P_{Restt,m} = P_{Restt-1} - K_S t + P_{Sm,t} \tag{57}$$

と表すことができる．

パケット破棄を起こさないためには，時刻 t に m で，送信キューに残っているパケット数が送信キューの数 L_{Sm} を越えなければ良い．よって，時刻 t の受信キューの状態を，t で受信キューに残っているパケット数から以下のように定義する．

$$P_{Restt,m} = 0 \Leftrightarrow \phi \tag{58}$$

$$0 < P_{Restt,m} < L_{Sm} \quad \Leftrightarrow \text{ パケット破棄は起きない} \tag{59}$$

$$L_{Sm} < P_{Restt,m} \quad \Leftrightarrow \text{ パケット破棄が発生} \tag{60}$$

となる．

3.4.4 目的関数の定式化

以下で，3 つの目的関数の定式化を行う．

パケット破棄を起こさない範囲で配置できる仮想ノードの最大数　OS および，スイッチの送受信キューでパケット破棄を起こさない範囲で物理ノードに配置できる仮想ノードの最大数を求めるために目的関数と制約条件の定式化を行う，ここで，OS およびスイッチでパケット破棄を起こさない条件は，式 (39)(46)(53)(59) より導かれる．以上より，x_{jm} を仮想ノードがスイッチのイーサネットポート m に接続していれば 1，接続していなければ 0，物理ノード数 p を定数とし，この条件を定式化すると，

$$maximize : \sum_{j=1}^{q} x_{jm} \tag{61}$$

$$subject\,to :$$

$$\sum_{m=1}^{p} x_{jm} = 1 \tag{62}$$

$$x_{jm} \in \{0, 1\} \tag{63}$$

$$0 \leq \sum_{T=1}^{t} (P_{SlT} - K_{Sl}T) < L_{Sl} \tag{64}$$

$$0 \leq \sum_{T=1}^{t} (P_{RlT} - K_{Rl}T) < L_{Rl} \tag{65}$$

$$0 \leq \sum_{j=1}^{q} x_{jm} \sum_{T=1}^{t} \Big(P_{Sm,T} - K_{Sm}T\Big) < L_{Sm} \tag{66}$$

$$0 \leq \sum_{j=1}^{q} x_{jm} \sum_{T=1}^{t} \Big(P_{Rm,T} - K_{Rm}t\Big) < L_{Rm} \tag{67}$$

となる．

スイッチが処理するパケット数の最小化　スイッチが処理するパケット数を最小化させるような仮想ノードの配置を行うための目的関数と制約条件の定式化を行う．

スイッチが処理するパケット数を最小化させるためには，仮想ノードがスイッチに送信数するパケット数を最小にする割り当てを行えば良い．仮想マシンに配置できる仮想ノードの loopback デバイスによる上限を Max_{Guest} とすると，制約条件は

$$
\begin{aligned}
&minimize:\ P_{St} = \sum_{k=1}^{p}\sum_{e=1}^{P_{S_t}} x_e P_{S_t k,e} &(68)\\
&subjet\,to:\\
&0 \leq \sum_{j=1}^{q} x_{jm} \sum_{T=1}^{t}\left(P_{Sm,T} - K_{Sm}T\right) < L_{Sm} &(69)\\
&0 \leq \sum_{j=1}^{q} x_{jm} \sum_{T=1}^{t}\left(P_{Rm,T} - K_{Rm}T\right) < L_{Rm} &(70)\\
&x_{jm} \in \{0,1\} &(71)\\
&\sum_{j=1}^{q} x_{jm} = Max_{Guest} &(72)
\end{aligned}
$$

となる．

スイッチの各ポートが処理するパケット数の均一化　時刻 t にスイッチのイーサネットポートが処理するパケット数は式 (53)(59) より導かれる．

物理ノード数 p とスイッチ数 s は定数とし，時刻 t に各イーサネットポート m で処理されたパケット数の平均を $\overline{P_{m,t}}$ として定式化すると，

$$
\begin{aligned}
&minimize:\ \frac{1}{p}\sum_{k=1}^{p}\left(\overline{P_{St,k}} - P_{Sm,t,k}\right)^2 &(73)\\
&subject\,to:\\
&0 \leq \sum_{j=1}^{q} x_{jm} \sum_{T=1}^{t}\left(P_{Sm,T} - K_{Sm}T\right) < L_{Sm} &(74)\\
&0 \leq \sum_{j=1}^{q} x_{jm} \sum_{T=1}^{t}\left(P_{Rm,T} - K_{Rm}T\right) < L_{Rm} &(75)\\
&\sum_{j=1}^{q} x_{jm} = Max_{Guest} &(76)
\end{aligned}
$$

となる．

3.4.5 送受信バッファサイズの定式化

まず始めに物理ノード数または仮想ノード数を決定する目的関数を考える．この目的関数としては，物理ノード数が決定している場合，オーバーフローを起こさないで配置できる仮想ノード数という目的関数がある．これは，スイッチのポート，物理ノードと仮想ノードのネットワークインターフェースカード (NIC) でバッファオーバーフローを起こさないで配置できる仮想ノード数を求めることができる．次に仮想ノードの配置を決定する目的関数を考える．この目的関数としては，スイッチを通る通信量の最小化をするような配置と各ポートが時刻 t に処理するトラフィック量を均一にする配置が考えられる．スイッチを通る通信量の最小化をするような配置をすることにより，スイッチが処理するトラフィック量が小さくなり，スイッチにかかる負荷を小さくことができる．また，各ポートが時刻 t に処理するトラフィック量を均一にする配置をすることにより，あるポートに負荷が集中するような状況を回避するような配置を行うことが可能になる．以上より，目的関数は以下のものが考えられる．

物理ノード数または仮想ノード数を決定する目的関数

目的関数 7 オーバーフローを起こさない範囲で配置できる仮想ノードの最大数

仮想ノードの配置を決定する目的関数

目的関数 8 スイッチを通る通信量の最小化

目的関数 9 スイッチの各ポートを通る通信量の均一化

ノードの送受信バッファサイズの定量化　ノードでは一度に処理しきれなかったパケットは送信，受信ともにバッファにためておき，処理を行う．ノードで時刻 t にソケット l の送信キューに残っているパケット数を $P_{SRestl,t}$ とする．すると，送信バッファとして必要なサイズ P_{Sbl} は，パケット e のサイズを P_{Sze} とすると，

$$P_{Sbl} = \sum_{e=1}^{P_{SRestl,t}} P_{Sze} P_{SRestl,t,e} \tag{77}$$

となる．

次に，受信の場合を考える．

ノードで時刻 t にソケット l の受信キューに残っているパケット数を $P_{RRestl,t}$ とする．すると，受信バッファとして必要なサイズ P_{Rbl} は，パケット e のサイズを P_{Sze} とすると，

$$P_{Rbl} = \sum_{e=1}^{P_{RRestl,t}} P_{Sze} P_{RRestl,t,e} \tag{78}$$

となる．

OS の送信バッファのサイズを P_{Sbmax}，受信バッファのサイズを P_{Sbmax} とすると，P_{Sb} と P_{Rb} は

$$P_{Sbm} \leq P_{Sbmax} \tag{79}$$

$$P_{Rbm} \leq P_{Rbmax} \tag{80}$$

を満たさなければならない．この条件を満たさない場合，オーバーフローを起こし，正しく送受信が行えない．

スイッチの送受信バッファサイズの定量化　スイッチでもノードと同様に一度に処理しきれなかったパケットは送信，受信ともにバッファにためておき，処理を行う．スイッチで時刻 t にイーサネットポート m の送信キューに残っているパケット数を $P_{SRestm,t}$ とする．すると，送信バッファとして必要なサイズ P_{Sbm} は，パケット e のサイズを P_{Sze} とすると，

$$P_{Sbm} = \sum_{e=1}^{P_{SRestm,t}} P_{Sze} P_{SRestm,t,e} \tag{81}$$

となる．

次に，受信の場合を考える．

スイッチで時刻 t にイーサネットポート m の受信キューに残っているパケット数を $P_{RRestm,t}$ とする．すると，受信バッファとして必要なサイズ P_{Rbm} は，パケット e のサイズを P_{Sze} とすると，

$$P_{Rbm} = \sum_{e=1}^{P_{RRestm,t}} P_{Sze} P_{RRestm,t,e} \tag{82}$$

となる．

よって，スイッチでの送受信バッファのサイズの上限を P_{Sbmax}，受信バッファのサイズを P_{Sbmax} とすると，P_{Sb} と P_{Rb} は

$$P_{Sb} \leq P_{Sbmax} \tag{83}$$

$$P_{Rb} \leq P_{Rbmax} \tag{84}$$

を満たさなければならない．この条件を満たさない場合，オーバーフローを起こし，正しくパケットの送受信が行えない．

3.4.6 トラフィック量に関する目的関数の定式化

以上の式を用いて，設定した目的関数 1 から 3 の定式化を行う．

オーバーフローを起こさない範囲で配置できる仮想ノードの最大値　OS およびスイッチの送受信バッファでオーバーフローを起こさない範囲で物理ノードに配置できる仮想ノードの最大数を求めるための目的関数と制約条件の定式化を行う．

送受信キューでオーバフローを起こさない条件は，式 (79)(80)(83)(84) である．以上より，x_{jm} を仮想ノード j がスイッチのイーサネットポート m に接続していれば 1，していない場合は 0，物理ノード数 p とスイッチのイーサネットポート数 v を定数とし，この条件を定式化すると，

$$maximize \sum_{j=1}^{q} x_{jm} \tag{85}$$

$$subject\,to:$$

$$\sum_{m=1}^{v} x_{jm} = 1 \tag{86}$$

$$x_{jm} \in \{0,1\} \tag{87}$$

$$\sum_{e=1}^{P_{SRestl,t}} P_{Sze} P_{SRestl,t,e} < P_{Sbmaxl} \tag{88}$$

$$\sum_{e=1}^{P_{RRestl,t}} P_{Sze} P_{RRestl,t,e} < P_{Rbmaxl} \tag{89}$$

$$\sum_{e=1}^{P_{SRestm,t}} P_{Sze} P_{SRestm,t,e} < P_{Sbmaxm} \tag{90}$$

$$\sum_{e=1}^{P_{RRestm,t}} P_{Sze} P_{RRestm,t,e} < P_{Rbmaxm} \tag{91}$$

となる．

スイッチを経由する通信量の最小化　スイッチを通る通信量を最小化させるような仮想ノードの配置を行うための目的関数と制約条件の定式化を行う．

スイッチを通る通信量を最小化させるためには，スイッチの各イーサネットポートで受信したパケット数の合計が最小になれば良い．

$$minimize \sum_{m=1}^{v} P_{Rbm} \tag{92}$$

$$subject\,to$$

$$\sum_{e=1}^{P_{RRestm,t}} P_{Sze} P_{RRestm,t,e} < P_{Rb_m ax_m} \tag{93}$$

スイッチの各ポートが処理するパケットサイズの均一化　物理ノードへ仮想ノードの配置を行う場合，時刻 t で必要とされる送信，受信のバッファサイズが式 (83) と式 (84) を満たし，かつ各スイッチのイーサネットポートの時刻 t に使用されている送信，受信のバッファサイズの分散が最小となるような割り当てが必要である．

まず送信バッファについて考えると，イーサネットポート m が時刻 t に使用している送信バッファサイズを $P_{Sbm,t}$，x_{jm} がスイッチのイーサネットポート m に

つながっている物理ノード上の仮想ノードかどうかの判定，loopback デバイスによる仮想マシンに配置できる仮想ノードの最大数を Max_{Guest} とすると，定式化を行うと，

$$\frac{1}{v}\sum_{m=1}^{v}\left(\overline{P_{St,k}} - P_{St,k,m}\right)^2 \tag{94}$$

$$subject\,to:$$

$$P_{Sb} \leq P_{Sbmax} \tag{95}$$

$$\sum_{j=1}^{q} x_{im} P_{Sbt,m} \leq P_{Sbmaxt,m} \tag{96}$$

$$\sum_{j=1}^{q} x_{jm} = 1 \tag{97}$$

$$\sum_{j=1}^{q} x_{jm} = Max_{Guest} \tag{98}$$

$$x_{jm} \in \{0, 1\} \tag{99}$$

となる．

次にスイッチの受信バッファについて考える．イーサネットポート m が時刻 t に使用している受信バッファのサイズを $P_{Rbt,m}$，y_{jm} がパケットの宛先がイーサネットポート m につながっている物理ノード上の仮想ノードなら 1，そうでないなら 0 とし，定式化を行うと，

3.5 目的関数の組み合わせに関する考察

仮想ノードを物理ノードに配置するアルゴリズムを考える前に，メモリ，パケット数とネットワークトラフィックの 3 点で設定した物理ノード数もしくは仮想ノード数を決定する目的関数と，仮想ノードの配置を決定する目的関数をそれぞれ列挙する．

物理ノード数もしくは仮想ノード数を決定する目的関数

目的関数 1 与えられた仮想ノードを配置するために必要な最小の物理ノード数

目的関数 2 与えられた物理ノードに配置可能な最大の仮想ノード数

目的関数 4 パケット破棄を起こさずに物理ノードに配置可能な仮想ノード数

目的関数 7 オーバーフローを起こさずに物理ノードに配置可能な仮想ノード数

仮想ノードの配置を決定する目的関数

目的関数 3 各物理ノードに配置された仮想ノードのメモリの合計値の平滑化

目的関数 5 スイッチが処理するパケット数の最小化

目的関数 6 スイッチの各イーサネットポートで処理するパケット数の平滑化

目的関数 8 スイッチを通る通信量の最小化

目的関数 9 スイッチの各イーサネットポートの通信量の平滑化

仮想ノードを物理ノードに割り当てる資源割り当てアルゴリズムは，物理ノード数もしくは仮想ノード数を決定する目的関数と仮想ノードの配置を決定するアルゴリズムの組み合わせで決まる．

まず始めに物理ノード数と仮想ノード数を決定する．物理ノード数と仮想ノード数は，実験者が両方とも指定するか，どちらか一方が決まっている場合は，物理ノード数もしくは仮想ノードを決定する目的関数を使って決まっていない方を決定する．次に，最適化問題の近似解を求めるアルゴリズムで，配置の目的関数を満たすような仮想ノードの物理ノードへの配置をおこなう．

物理ノード数または仮想ノード数を決定する目的関数の組み合わせは全部で 9 種類存在する．1 つ目は，実験者が物理ノード数と仮想ノード数の両方を設定する場合である．2 つ目は，仮想ノード数が決定されている場合である．このとき，物理ノード数を求める目的関数は 1 のみである．3 つ目は使用する物理ノード数が決まっている場合である．この時，仮想ノード数を求める目的関数は 2,4,7 である．これらを組み合わせは全部で 7 種類存在する．それぞれの目的関数は NP-

表 1 目的関数の組み合わせ (ノード数決定)

目的関数 1 仮想ノード配置に必要な最小の物理ノード数	目的関数 2 物理ノード配置可能な最大の仮想ノード数	目的関数 4 パケット破棄が起こらないで配置可能な仮想ノード数	目的関数 7 トラフィックのオーバーフローが起こらないで配置可能な仮想ノード数	求めるノード数
未使用	未使用	未使用	未使用	なし
使用	未使用	未使用	未使用	物理ノード数
未使用	使用	未使用	未使用	仮想ノード数
未使用	未使用	使用	未使用	仮想ノード数
未使用	未使用	未使用	使用	仮想ノード数
未使用	使用	使用	未使用	仮想ノード数
未使用	使用	未使用	使用	仮想ノード数
未使用	未使用	使用	使用	仮想ノード数
未使用	使用	使用	使用	仮想ノード数

困難な問題である．物理ノード数もしくは仮想ノード数を決定する場合，NP-困難な問題の組み合わせを解く多次元最適化問題となる．すべての組み合わせを表 1 に表す.

配置決定の目的関数は 5 種類ある．それらの組み合わせは全部で 32 種類となる．5 つの目的関数は物理ノード数もしくは仮想ノードを求める時の目的関数同様，すべて NP-困難な問題のため，その組み合わせは多次元最適化問題となるため近似解やヒューリスティックを用いた解として出すことになる．よって，組み合わせのうち，使用する目的関数が多くなれば，配置として最適化されて行くため実験環境として安定しやすくなる．しかし，使用する目的関数が多くなれば，条件を満たす組み合わせは少なくなりかつ条件判定をより多く行うため解が出るまでの時間が長くなる．逆に配置に使用する目的関数の数を減らすと解が出るまでの時間は短くなるが，スイッチが処理するパケットやトラフィック量の増加や

パケット破棄の可能性が発生してしまう．

以上より，仮想ノードを物理ノードに割り当てる資源割り当てアルゴリズムは全部で 288 通りの組み合わせが考えられる．ここで，物理ノード数または仮想ノード数を決定する目的関数の組み合わせと配置決定の目的関数は多次元最適化問題であるので，資源割り当てアルゴリズムも多次元最適化問題となる．そのため，NP-困難問題となり近似解やヒューリスティックによる解をもとめないと現実時間で解が求められない．しかし，メモリの割り当てであれば，OS が消費するメモリ量と OS 上で起動するアプリケーションが消費するメモリ量がわかればメモリ割り当ての最適を出すことは可能であると考えられる．そこで，次章では，このメモリの割り当てに着目して最適なメモリ割り当て手法の提案を行う．

3.6 提案手法

3.6.1 要求事項

研究室レベルでの数台からなるサーバクラスタは実験者による長期間の占有が可能であるが，物理計算機の台数制限から，大規模なインターネットエミュレーションの実施は難しい．一方，StarBED [7]，Emulab [46], DeterLab [6] などの大規模なインターネットエミュレーションを実施するテストベッドは 100 台から 1,000 台規模の物理サーバを保有しているが，複数の利用者による共有資源であり，借用期間や借用ノード数の上限が制限される．テストベッドの借用期間は通常 1 週間から最長でも 2ヶ月程度となっており，その期間内で実験を実施しなければならない．よって，実験の基礎となる実験環境の構築にかける時間は出来る限り短時間でなければならない．

テストベッドでインターネットエミュレーションを用いた実験を実施する場合，実験初期段階では数台から数十台の物理計算機を用いた小規模または中程度の規模で実験を実施して仮想計算機を用いた実験環境の構築に習熟した後に，100 台を超える物理計算機を用いた大規模な実験を実施することが一般的である．規模を拡張する際に，仮想計算機やその上のプロセスに対しての資源割り当ての方法が規模によって変化してしまうと，実験規模ごと資源割り当ての方法を考え，新

たに考案した資源割り当て方法の妥当性の検証が必要になる．よって，資源割り当て方法は規模にかかわらず一定の手法が適用できることが望ましい．

実験を行う際に，仮想計算機に対して実験を施行する上で充分な資源を割り当てないと，仮想計算機上のプロセスが起動せず実験環境の構築が行えないか，資源枯渇により正確な性能検証が実施できなくなる．そのため，各仮想計算機には実験の実施に充分な資源を割り当てる必要がある．

以上を本論文で提案する AS エミュレーションにおける External Border Gateway Protocol (EBGP) ルータの使用メモリ量の算出式に関する要求事項としてまとめる．

要求 1 使用メモリ量の推定にかかる計算量が小さい

要求 2 規模の大小にかかわらず提案する推定式が使用可能である

要求 3 割り当てた資源が不足しない

本論文ではこれらの要求を満たすような AS エミュレーションにおける EBGP ルータへのメモリ使用量の推定手法の提案を行う．

3.7 EBGP ルータのメモリ消費量モデル

EBGP ルータの経路制御に必要となるメモリ，言い換えると BGP ルータの RIB に必要なメモリは，RFC4271 [55] の仕様から，一般的に次の 3 つのメモリ量に分割できる．

1. **Adj-RIBs-In** : 隣接 AS の EBGP ルータから受け取った BGP update メッセージにより広報されてきた経路を保持するために要するメモリ
2. **Adj-RIBs-out** : EBGP ルータが隣接 AS の EBGP ルータに BGP update メッセージで広報する経路情報を保持するために要するメモリ
3. **Loc-RIB** 最適経路情報を保持するために要するメモリ

と定義されている．

よって，ここではある AS i の EBGP ルータの RIB に要するメモリは

1. M_i^{in} : 全隣接 AS の EBGP ルータから受け取った経路を保持するために要するメモリ量

2. M_i^{out} : EBGP ルータが全隣接 AS の EBGP ルータに送信する経路情報を保持するために要するメモリ量

3. M_i^{Loc} : 最適経路情報を保持するために要するメモリ量

と定義できる．

ここで，Adj-RIBs-in および Adj-RIBs-out は隣接 AS との BGP セッションと BGP update メッセージおよび経路情報を交換するために必要なメモリであると読み替えることができる．一方，Loc-RIB は受け取った経路情報から算出された最適経路情報を保持するメモリであるため，その情報はパケット転送に，つまり FIB (Forwarding Information Base) に必要なメモリと読み替えることが出来る．

そこで，

1. M_i^{nb} : AS i の EBGP ルータにて全隣接 AS の EBGP ルータとの BGP セッションを保持するために必要なメモリ量

2. M_i^r : AS i の EBGP ルータにて保持する全経路情報に必要なメモリ量

3. M_i^{fib} : AS i の EBGP ルータにて FIB に必要なメモリ量

を設定すると，Adj-RIBs-in, Adj-RIBs-out, Loc-RIB の定義の読み替えから M_i^r および M_i^{fib} は，

$$\begin{aligned} M_i^r &= M_i^{in} + M_i^{out} \\ M_i^{fib} &= M_i^{Loc} \end{aligned} \tag{100}$$

となる．

一方，BGP 経路制御プログラムは BGP セッションのために TCP セッションを隣接 AS 数分作成するため，直感的に，M_i^{nb} は隣接 AS 数に従う関数として定義

できる．BGP update メッセージの特性上，上流に位置するプロバイダ AS へ送るメッセージ量，対等な関係に位置するプロバイダ AS へ送るメッセージ量および下流に位置するカスタマ AS に向けて送るメッセージ量に違いがあるため，上流に位置する隣接 AS 数 E_i^{up} と下流に位置する隣接 AS 数 E_i^{dn}，対等の関係に位置する隣接 AS 数 E_i^{pr} の三つの変数に従う関数として定義できる．それぞれの隣接関係を持つ AS は排他関係であるため，AS_i が消費するメモリ量 M_i^{nb} は上流に位置する隣接 AS から送られてくる経路により消費するメモリ量 M_i^{up} , 下流に位置する隣接 AS から送られてくる経路により消費するメモリ量 M_i^{dn} と対等な関係に位置する隣接 AS から送られてくる経路により消費するメモリ量 M_i^{pr} を足した式 (101) のようになる．

$$M_i^{nb} = M_i^{dn} + M_i^{up} + M_i^{pr} \tag{101}$$

$$M_i^{up} = f(E_i^{up}) \tag{102}$$

$$M_i^{dn} = g(E_i^{dn}) \tag{103}$$

$$M_i^{pr} = h(E_i^{pr}) \tag{104}$$

ここで，関数 $f(x), g(x), h(x)$, は BGP ルーティングソフトウェアの実装方法や経路フィルタにより定義が変わるが，ソースコードの静的解析や小規模なネットワークトポロジーでの動的解析の計測結果をもとにした回帰分析による立式が可能であるといえる．実際，BGP 経路制御プログラムを含むオープンソースである Quagga [59] や bird [60] には BGP 情報の構造体の中に BGP コネクション（ソケット）のポインタ情報を保持するため，式 (101)，(102)，(103，(104) でのモデル化は妥当である．

3.8 Quagga におけるケーススタディ

本論文では，モデル化手法の妥当性を検証するために，BGP を含むオープンソースのルーティングソフトウェアである Quagga を元にケーススタディを行った．

```
route-map export-my-route-only permit 10
 match as-path MyAS

 route-map export-my-route-only deny 20
```

図 1 Uplink の経路フィルタ

表 2 実験環境のハードウェア仕様

項目	スペック
CPU	AMD Opteron 146HE 2.0GHz
メモリ	8GB
Ethernet	1 GigabftEthernet

3.9 小規模実験による動的解析と回帰式のパラメータ導出

まず，比較的小規模なサーバクラスタ環境を用いて Quagga bgpd のメモリ使用量の検証を行い，3.7 項で定義した式 (101)，(102)，(103)，(104) に対する回帰式とそのパラメータを導出する．

小規模環境での計測は，以下の 2 つを行った．

1. 隣接 AS との位置関係ごとの隣接 AS 数と bgpd 消費メモリ量の関係
2. bgpd が保持する経路数と bgpd の消費メモリの関係

3.10 実験環境

回帰式のパラメータを導出するための環境として表 2 に示す仕様の物理サーバ上に VMKnoppix 4.2 [61] 起動して検証を実施した．VMKnoppix 上では Xen 3.2 [12] を用いた準仮想化にて仮想化計算機を動作させた．仮想計算機の OS には ttylinux [62] をインストールし，ttylinux 上では BGP daemon として Quagga 0.94 を起動した．Quagga は bgpd と zebra の 2 種類のプロセスが動作するが，それぞ

れの消費メモリ量は proc ファイル・システム内のそれぞれのプロセスの物理メモリの実測値である VmHWM の値を `logger (1)` を用いて syslog サーバに集約し，観測した．

3.11 経路フィルタ

AS エミュレーションでは，1 拠点の BGP ルータで観測された経路情報や複数の BGP ルータで観測された経路情報を元に推定した AS 関係トポロジーがよく用いられる．AS 関係トポロジーで著名なデータセットとして，CAIDA プロジェクト [63] が提供している AS Relationship データセット [64, 65] が存在する．

CAIDA が提供している AS Relationship データセットでは，AS 間の BGP の隣接関係を Peer to Peer (P2P), Provide to Customer (P2C) , Customer to Provider (C2P), Siblings の 4 種類に定義している．P2P リンクは対等の関係にある AS 間での隣接関係で，本論文での Peer に該当する．P2C は Downlink，C2P は Uplink にそれぞれ該当する．Siblings は「同一組織で運用されている別 AS 間の隣接関係」と定義されているが，データセット上ではほとんど出現しないため，本論文では割愛する．

この CAIDA AS Relationship データセットから Quagga bgpd および zebra の設定ファイルを作成するツールセットとして，本論文では AnyBed [66] を用いた．AnyBed では，AS Relationship データセットの隣接関係の値から Quagga bgpd の経路フィルタを生成するツールセットが提供されている．P2P や P2C の関係性の場合は特段経路フィルタを入れず，P2C の場合は図 1 のような経路フィルタが導入される．ローカルの経路のみを広報する経路フィルタを適用する仕組みになっている．この経路フィルタは，特段絞込みを行わない経路フィルタのため，想定される経路フィルタのうち最大の値となる．AnyBed では，IPv4 プライベートアドレスを用いた AS エミュレーションでは/24 をひとつ，CAIDA プロジェクトの `prefix2as` データセットを用いたグローバル IPv4 アドレスを使った AS エミュレーションでは `prefix2as` データセットに登録されている/24 の経路数を AS から Origin として広報する．本論文での実験では AnyBed で出力する経路フィルタと経路広報ルールを用い検証を行った．

3.11.1 計測 1:隣接関係ごとの隣接 AS 数と Quagga の消費メモリ量の関係

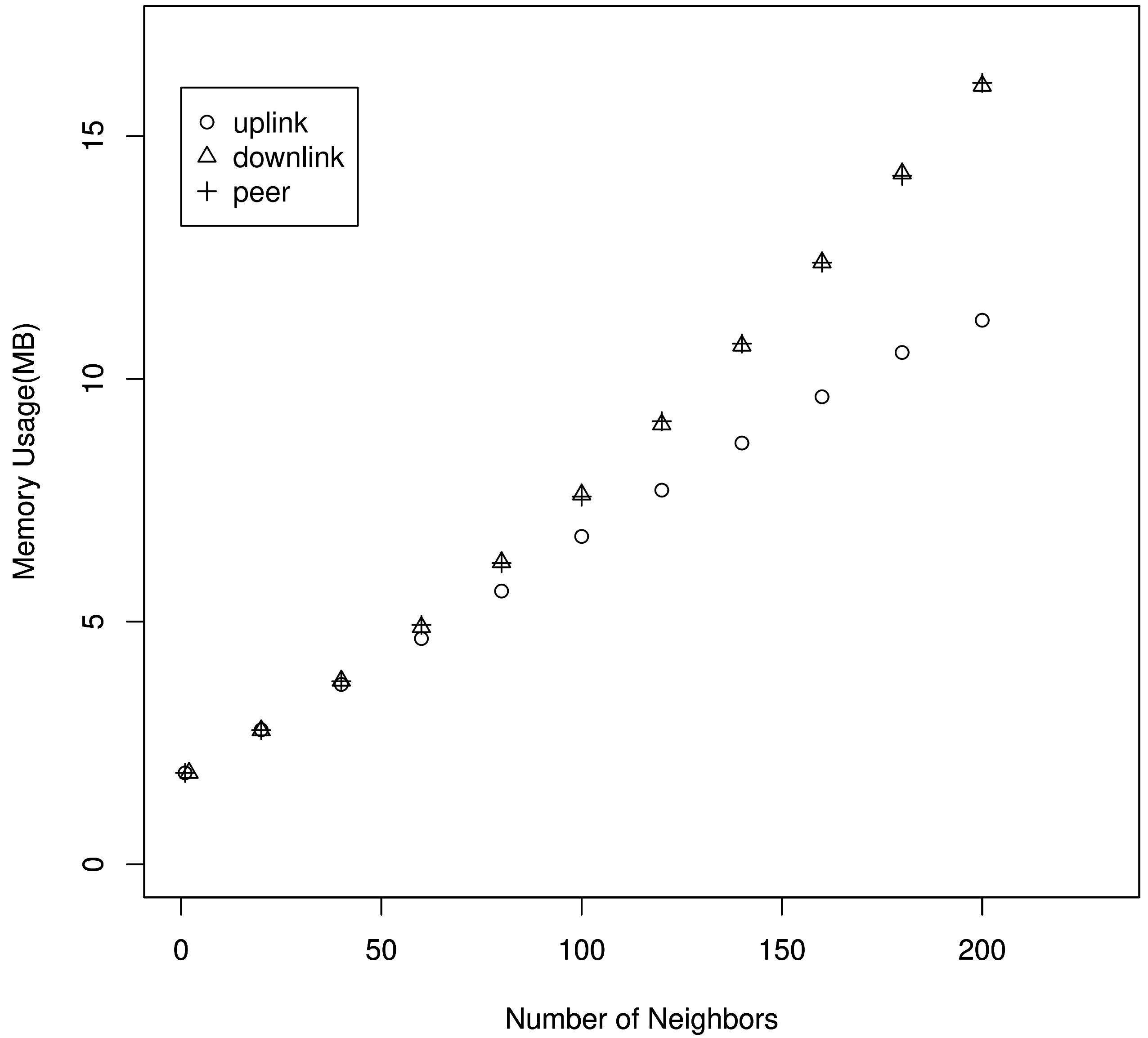

図 2 各隣接関係における bgpd のメモリ消費量

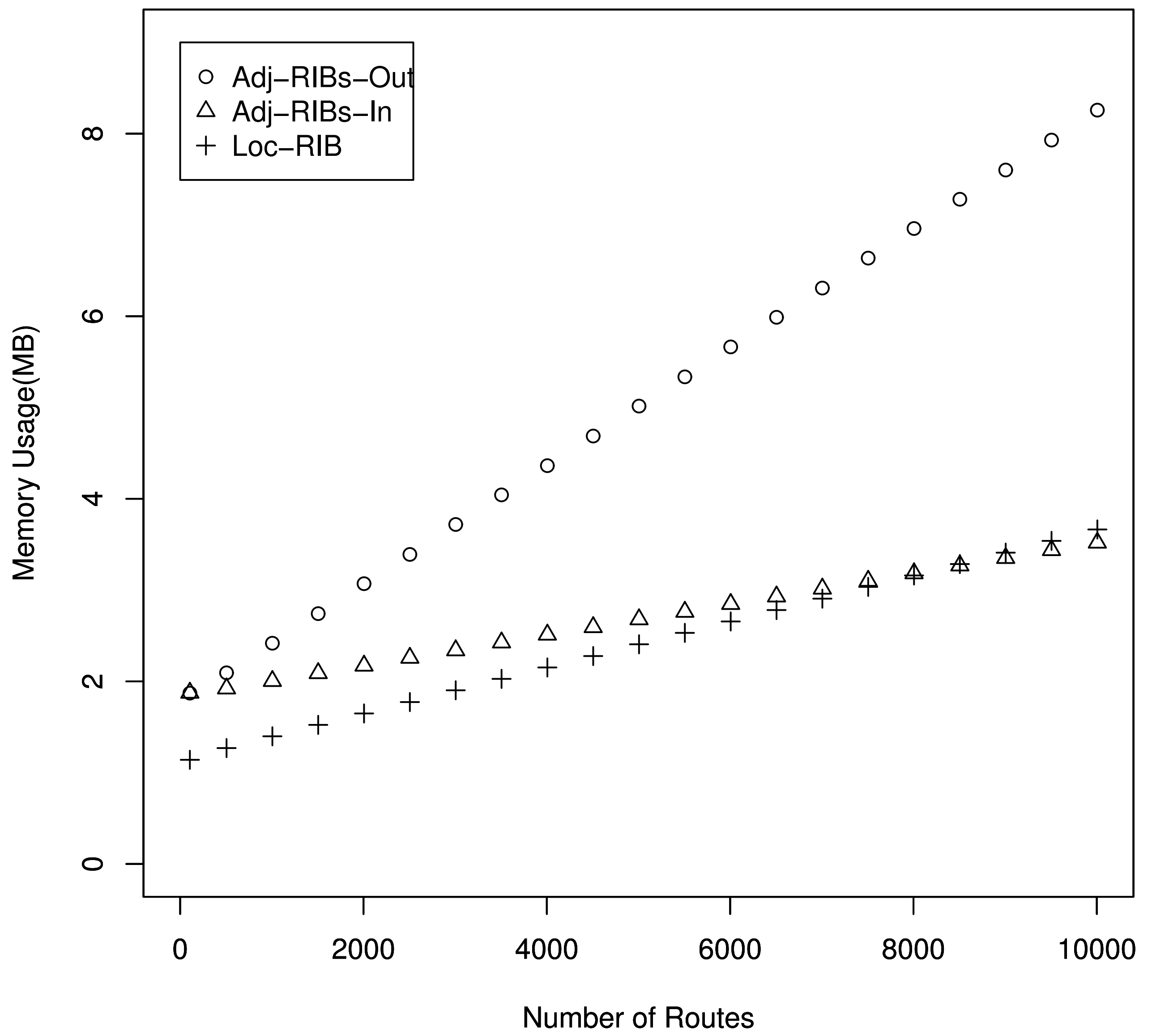

図 3 トポロジ全体の経路数と Quagga bgpd および zebra のメモリ消費量

Uplink，Downlink，Peer，それぞれの隣接AS数とbgpdの消費メモリ量の関係を求めるために，以下の手順でQuagga bgpdの消費メモリ量の計測を行った．

手順1： bgpdが消費するメモリ量を計測するAS i が200台のbgpdと接続するためのbgpdの設定ファイルを用意する．

手順2： AS i の隣接ASとなる仮想計算機を1台起動する．

手順3： すべての隣接ASであるbgpdとのコネクションがEstablishedになりbgpdによる経路交換が終了し，経路が安定状態になった段階で消費したメモリ量を計測する．

手順4： 計測終了後，1台新たにbgpdを起動し手順2を行う．これを200台終わるまで行う．

この計測に用いた表2に示す物理サーバは4台であった．計測した結果を図2に示す．図2は横軸をAS i がBGPセッションを確立しているAS数，縦軸はAS i が消費したメモリ量となっている．式(101)，(102)，(103)，(104)を元に，計測から得られた結果を線形回帰モデルである最小二乗法により得られた近似式を以下に示す．

$$\hat{M}_i^{dn} \simeq 0.14(E_i^{dn})^2 + 45.13(E_i^{dn}) + 1822 \qquad (105)$$

$$\hat{M}_i^{pr} \simeq 0.14(E_i^{pr})^2 + 45.13(E_i^{pr}) + 1822 \qquad (106)$$

$$\hat{M}_i^{up} \simeq 50.2(E_i^{up} + E_i^{pr}) + 1862 \qquad (107)$$

3.11.2 計測 2:Quagga が保持する経路数と Quagga の消費メモリ量の関係

Quagga では bgpd により交換された経路情報は zebra によって管理される．経路数と zebra の消費メモリの関係を求めるために，以下の手順で Quagga の zebra の消費メモリ量の計測を行った．

手順 1： 仮想マシンを 2 台（AS i と AS j）を用意する．

手順 2： AS j では，毎回自身が保持する IPv4 アドレスブロックを 10 個ずつ追加し，AS i に広報する．

手順 3： AS i において AS j から経路を受け取った事により bgpd と zebra が使用したメモリ量を AS i，AS j それぞれにおいて観測する．

手順 4： AS i，AS j の bgpd と zebra が使用したメモリの増加を確認後，手順 2 に戻る．

この計測実験では，AS i 側では経路情報を受信するために必要な M_{ij}^{in} の 1 経路ずつ増加する場合のメモリ増加傾向を計測し，反対に，AS j 側が経路情報を送信するために必要な M_{ij}^{out} の経路数に対する増加傾向を計測することになる．

計測の結果を図 3 に示す．図 3 は横軸が AS i が保持する経路数，縦軸は AS i が消費したメモリ量となっている．

FIB を保持する zebra に必要なメモリ量 $\hat{M}_i^{Loc}$ は全経路情報に対応するため，図 3 の結果を基にした線形回帰分析から

$$\hat{M}_i^{Loc} \simeq 0.26N^r + 1168 \tag{108}$$

となる．

一方で，経路保持に必要となる $\hat{M}_i^r$ は，AS i が保持する隣接関係と Origin として広報する経路数によって変化する．ここで，AS i が Origin として広報する経路数を N_i^r，AS エミュレーション全体での総経路数を N^r とし，計測実験 2 から得

られた結果を線形回帰分析することで得られた，AS j に対する AS i の，$\hat{M}_i^{in}$ は，Peer および Uplink の関係にある隣接 AS からは

$$\hat{M}_{ij}^{in} \simeq 0.17N^r \tag{109}$$

となり，Downlink の関係にある隣接 AS からは

$$\hat{M}_{ij}^{in} \simeq 0.17N_j^r \tag{110}$$

で推定されるメモリが必要になる．

一方，$\hat{M}_i^{out}$ は，経路フィルタによって変動し，Peer および Downlink の場合は全経路を広報するため，

$$\hat{M}_{ij}^{out} \simeq 0.66N^r \tag{111}$$

となり，Uplink の場合は自身が Origin の経路のみを広報するため，

$$\hat{M}_{ij}^{out} \simeq 0.66N_i^r \tag{112}$$

となる．

BGP Update メッセージの交換で必要となるメモリは，Deng らの状態方程式に従うが，最悪の場合を想定すると，Downlink, Peer の関係にある隣接 AS に対して全経路を広報し，Uplink の関係にある隣接 AS に対して自身が Origin の経路を広報する状態と想定できるため，$\hat{M}_i^{out}$ は

$$\hat{M}_i^{out} \leq 0.66N^r(E^{dn} + E^{pr}) + 0.66\sum_{k \in up} N_k^r \tag{113}$$

となる．ここで，$k \in up$ は隣接 AS との関係が Uplink である隣接 AS の集合である．一方，$\hat{M}_i^{in}$ は，Uplink, Peer の関係にある隣接 AS にから全経路を受信し，Downlink の関係にある隣接 AS から経路フィルタをかけられた情報を受け取るため，

$$\hat{M}_i^{in} \leq 0.17N^r(E^{up} + E^{pr}) + 0.17\sum_{l \in dn} N_l^r \tag{114}$$

となる．$l \in dn$ は隣接 AS との関係が Downlink である隣接 AS の集合である．

これにより隣接 AS と経路情報の交換を行うために使用されるメモリ量 M_i^r は式 (100)，式 (113) および式 (114) から作ることができる．ここで，式 (100) は同時にすべての隣接 AS から経路情報を交換することを想定した最悪値であり，実際の環境ではこのような状況になることはまずないため，重み係数 ω を設定し，割り当てるメモリ $\hat{M}_i^r$ を式 (115) と定義する．

$$\begin{aligned}\hat{M}_i^r &= \frac{1}{\omega}(0.66N^r(E^{dn} + E^{pr}) + 0.66\sum_{k \in up} N_k^r \\ &\quad +0.17N^r(E^{up} + E^{pr}) + 0.17\sum_{l \in dn} N_l^r)\end{aligned} \tag{115}$$

以上より，Quagga bgpd に割り当てるメモリ量 M_i^{bgpd} を推定する式を式 (116) と定義する．

$$M_i^{bgpd} = \hat{M}_i^{dn} + \hat{M}_i^{up} + \hat{M}_i^{pr} + \hat{M}_i^{r} \tag{116}$$

3.12 重みパラメータの選定

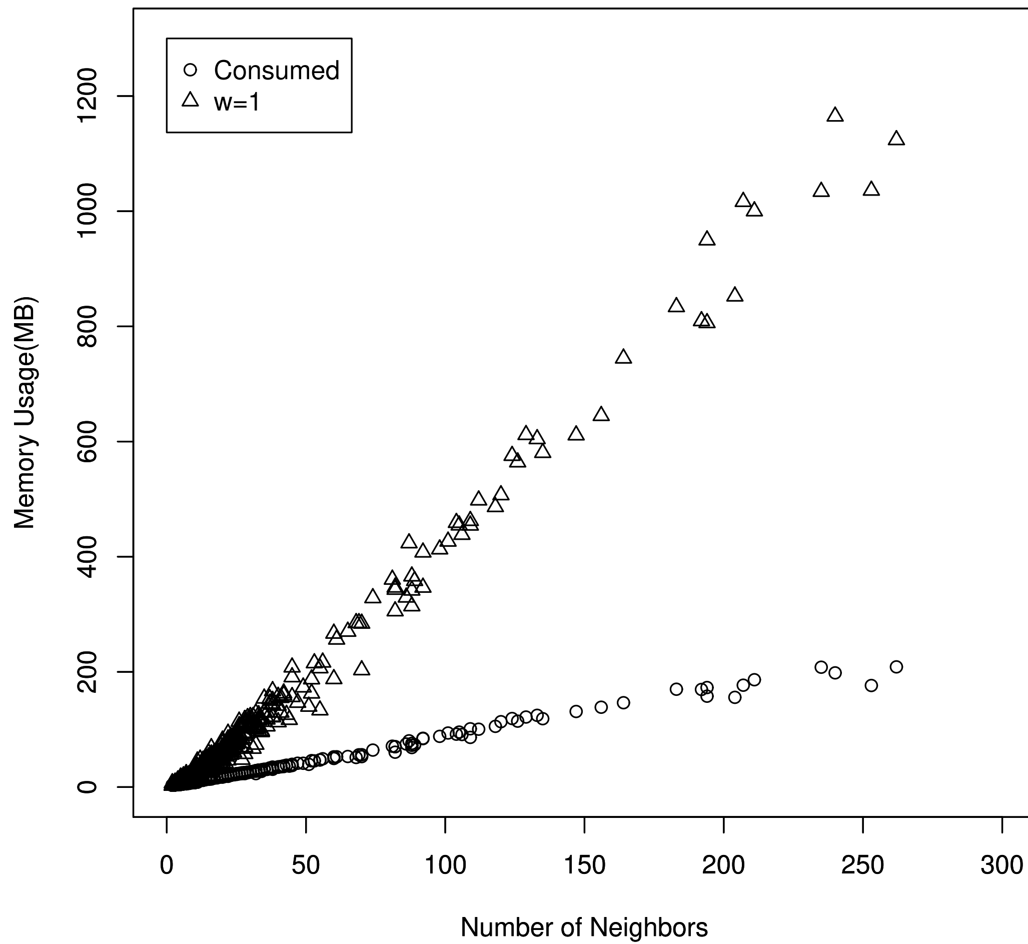

図 4　500AS における w=1 の場合の Quagga の消費メモリ量推定結果と実際に Quagga が使用したメモリ量

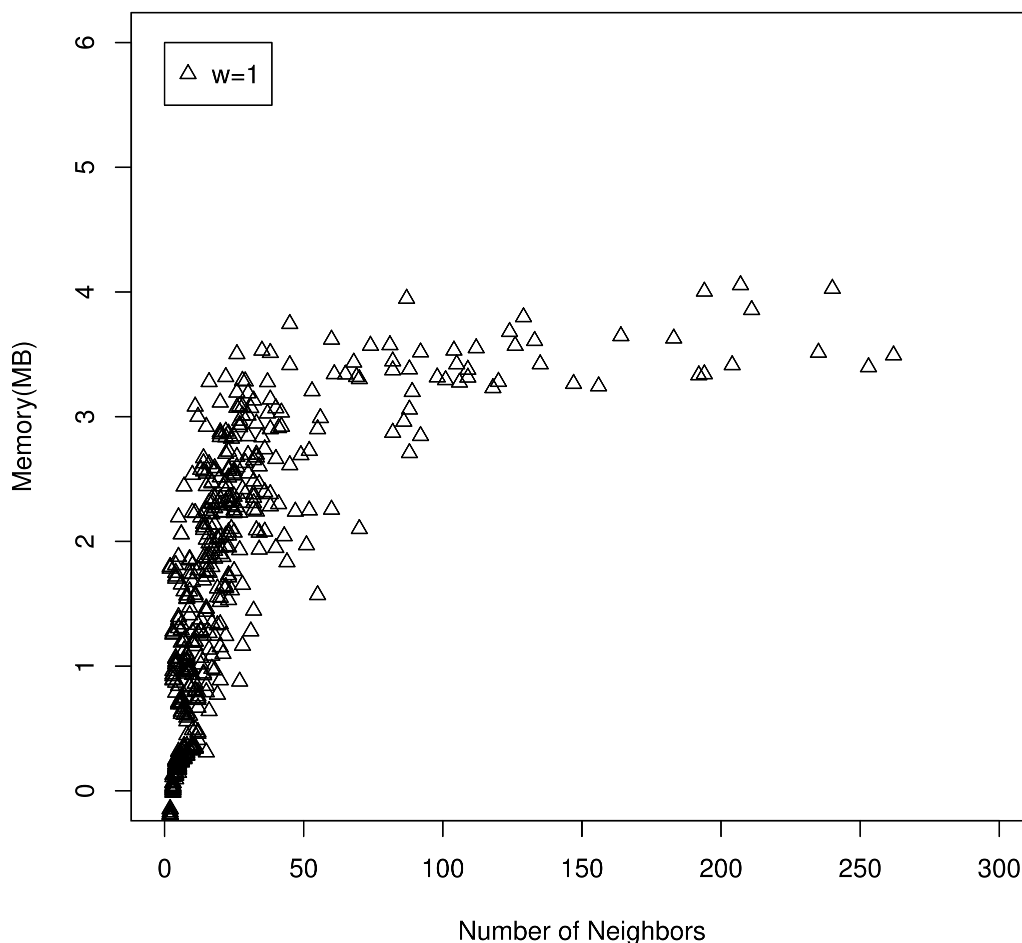

図 5 500AS における隣接 AS 数ごとの Quagga の消費メモリ量の推定結果と実際に Quagga が使用したメモリ量の誤差 ($\omega = 1$)

前節で定義した (116) 式の妥当性と適切な重みパラメータを評価するための実験を行った．実験に用いたトポロジは CAIDA Project [63] が 2009 年 7 月 20 日に収集した，AS Relationship データセットを利用し，文献 [67] で定義されたフィルタリングルールを用いて上位 500AS の AS トポロジーを抽出し，AnyBed [66] を

用いて Quagga bgpd および zebra の設定ファイルを生成した．IPv4 プライベートアドレスを用いた模擬を行ったため，各 AS から広報される Origin の経路は/24 ひとつとなっている．この検証には表 2 の物理サーバを 100 台用いた．

始めに重みパラメータを導入しない場合の割り当てメモリと Quagga bgpd が使用したメモリ量の関係を見る．図 4 に重みパラメータを導入しない場合 ($\omega = 1$) と Quagga bgpd が使用したメモリ量の関係を示す．この図は横軸が AS i と接続している AS 数，縦軸が割り当てまたは使用したメモリ量になっている．隣接 AS 数が多くなると誤差が増えているが，これは 1 隣接 AS あたりの誤差が増えているためで，この誤差が積算され結果合計での誤差が大きくなっている．次に，AS 数ごとの提案近似式による推定結果と実際に Quagga が使用したメモリ量の誤差を図 5 に示す．誤差を以下に定義する．

$$\text{誤差} = \text{推定近似結果} - \text{使用メモリ量の実測値} \tag{117}$$

ここで，重みパラメータ ω を設定する際に考慮すべき点を以下に示す．

- 導出したメモリ割り当て量で割り当て不足がない
- 導出したメモリ割り当て量が過剰な割り当てにならない

とくに，AS エミュレーション環境を構築する際，割り当て量が不足した場合実験環境が構築できなくなってしまう．よって，上記の条件のうち割り当て不足が起きない ω の中で最も割り当てるメモリ量が少なくなるような ω を求める．

本稿では仮想計算機上に BGP 経路制御プログラムを動かし，AS エミュレーション環境を構築することを目指しているが，仮想計算機にメモリを割り当てる際の単位は 1MB 以上であり今回設定する重みパラメータは 500AS で得られたデータの中で誤差が-1MB 以下であれば，仮想計算機に割り当てる量を決定する場合に切り上げることで影響がなくなる．よって，今回は 500AS において負方向の最大誤差が-1MB 以下になる最大の ω を探索する．

500AS の場合，$\omega = 1.4$ とすると割り当てたメモリ量と消費したメモリの誤差で負方向に最大の誤差は-0.74MB となり，$\omega = 1.5$ では誤差は-1.024MB と条件を

満たさなくなる．そこで，500AS のトポロジにおいて導出された $\omega = 1.4$ の規模追従性について次節で確認を行う．

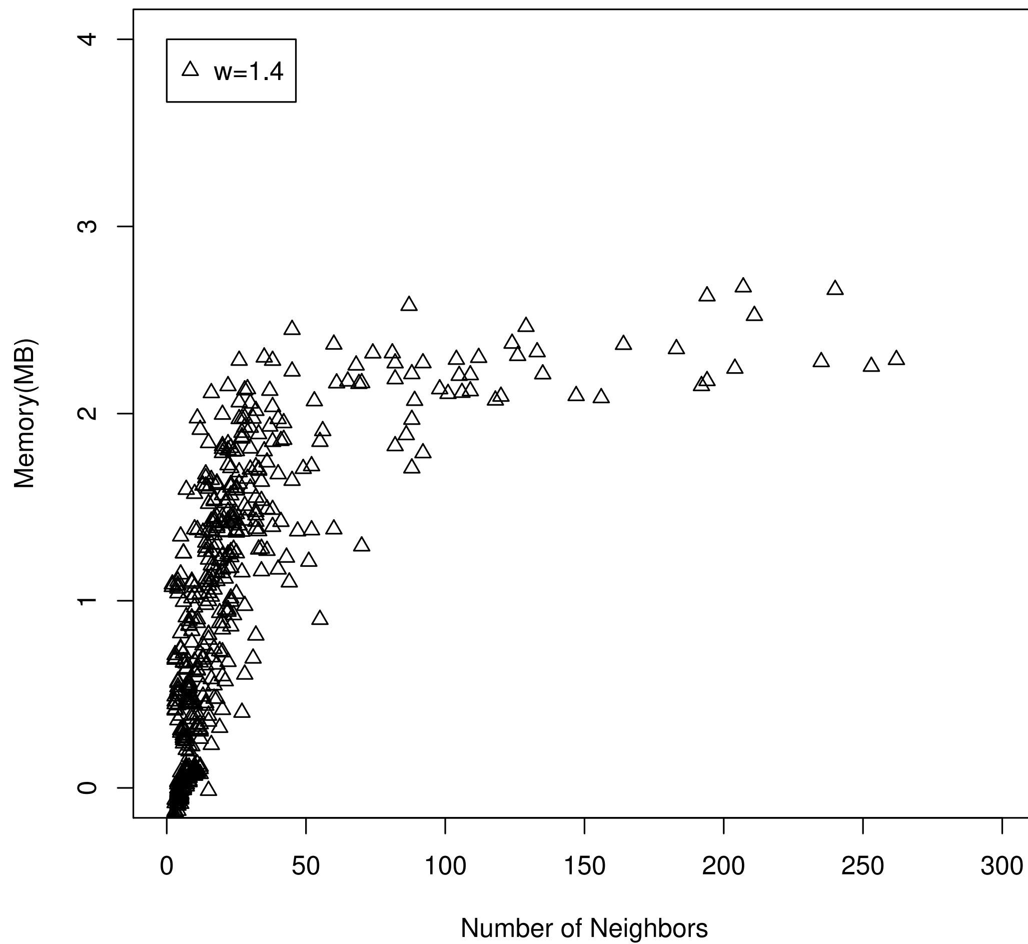

図 6 500AS における 1 隣接 AS あたりの割り当てメモリ量と消費メモリ量の差 ($\omega = 1.4$)

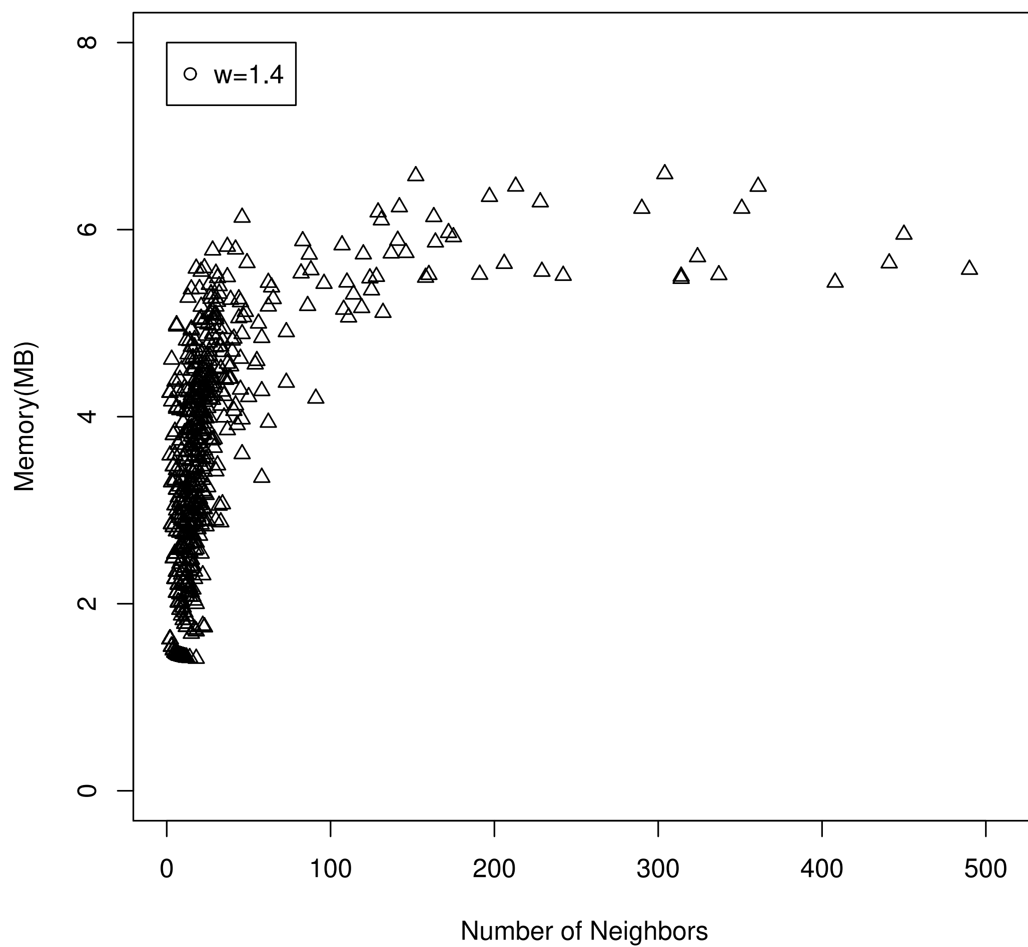

図 7 1000AS における 1 隣接 AS あたりの割り当てメモリ量と消費メモリ量の差 ($\omega = 1.4$)

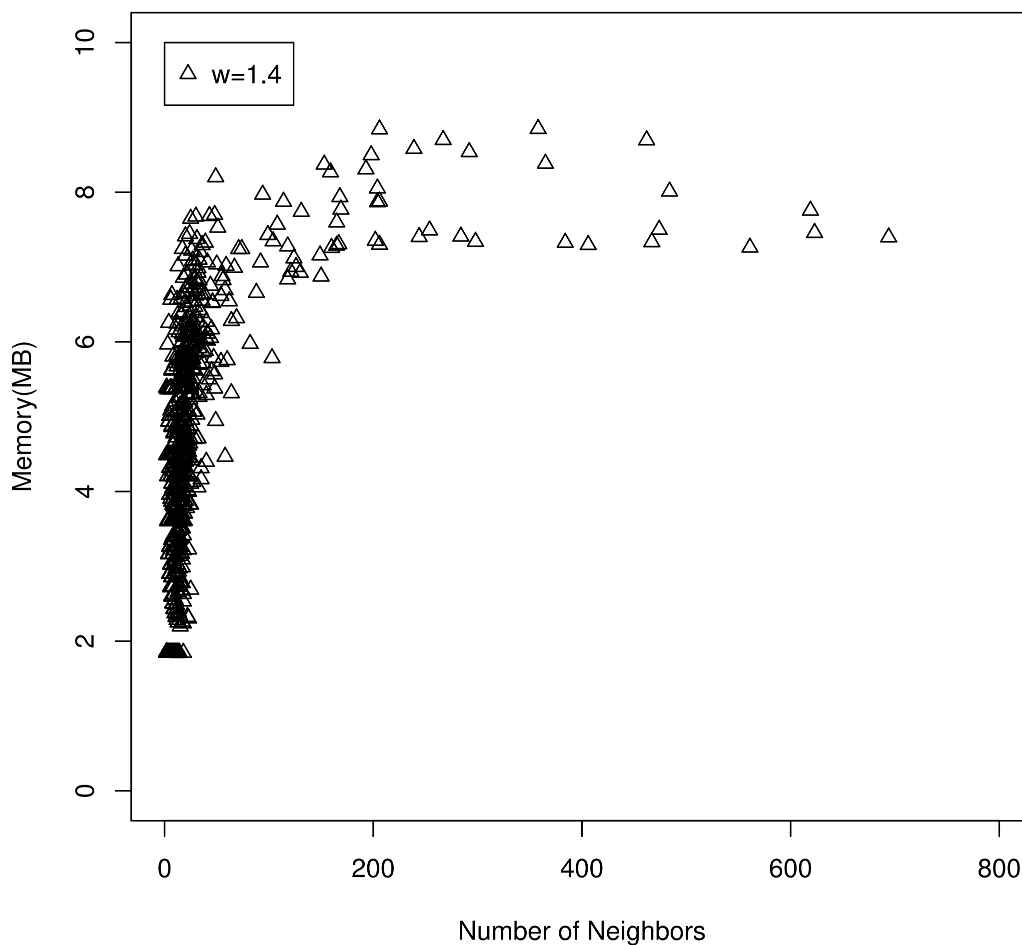

図 8 1500AS における 1 隣接 AS あたりの割り当てメモリ量と消費メモリ量の差 ($\omega = 1.4$)

3.13 選定した重みパラメータの規模追従性の検証

式 (116) に重みパラメータ $\omega = 1.4$ を設定した場合の規模追従性を評価するために，1000AS，1500AS で式 (116) によって得られた値と消費メモリ量の誤差の

調査を行う．実験に用いたトポロジは重みパラメータ $\omega = 1.4$ を導出した環境と同じ条件で，上位 1000AS，1500AS の AS トポロジーを抽出し，AnyBed [66] を用いて Quagga bgpd および zebra の設定ファイルを生成した．

$\omega = 1.4$ とした時の 500AS, 1000AS および 1500AS の AS トポロジでの割り当てメモリ量と使用メモリ量の誤差の分布を図 7，図 8 に表す．$\omega = 1.4$ とした場合，1000AS, 1500AS ともにで 1 隣接 AS あたりの誤差が，正の値になっていることがわかる．また，1500AS 時の誤差は 500AS および 1000AS の場合に比べ増加していることがわかる．このことから，規模が拡大しても割り当て不足になることはないと考えられる．このことから，規模の拡大方向への追従性はあるといえる．

3.14 議論

図 6, 図 7 および図 8 において，隣接する AS が多い AS で割り当てた値と使用したメモリ量の誤差が大きくなっている．この誤差は 1 隣接 AS あたりであるため，全体としてはこの誤差に隣接 AS 数がかけられるため，誤差は大きなものになってしまう．一方，この誤差を減らす場合多数の AS でメモリ割り当て不足が発生してしまう．本稿の手法の主目的が計算量が少なく AS エミュレーションを構築する際に必要なメモリ量を求めることであり，誤差を小さくすることが目的ではないため提案手法を採用した．また，500AS, 1000AS, 1500AS ともに隣接 AS 数が少ない AS での誤差にばらつきがあるのは，この場所に分布する AS は隣接 AS 数が同じであっても E^{up}, E^{dn} と E^{pr} の構成が異なるためである．

重みパラメータの正当性に関しては今回の検証で 500, 1000, 1500 と規模を変更した場合に誤差の最小値が増加傾向にあることから，AS エミュレーション環境を構築する際に各仮想計算機に割り当てるべき値を求める手法としては正しい．今回使用したトポロジは CAIDA によりインターネットから計測した値であり，このデータセットで AS エミュレーション環境を構築するために十分なメモリ量の導出が可能になっていることから目的に合致しているといえる．

計算量に関しては，提案式はトポロジ上に存在する AS 数を n とすると $O(n)$ となり，多項式時間で計算が終了しない RFC4274 や Deng らの手法に比べて明らかに計算量が少なくなっている．Miwa らの手法に比べては計算量の点ではどちら

も計算量は $O(n)$ であるが，Miwa らの手法は割り当てメモリの重みパラメータに規模追従性がない点およびパラメータ数が Miwa らの手法では変数が３つあるが本提案手法では１つしかなく，この点で本手法は Miwa らの手法より優位である．

3.14.1 提案手法の汎化をおこなうために

本研究では，経路制御プロトコルについて仮想計算機を用いた大規模検証環境の構築のために各仮想計算機に必要な資源について事前実験の結果から検討を行った．ネットワークを対象とした検証環境の構築に着目しても割り当ての考慮が必要な資源は５種類あり，これらの資源をそれぞれの仮想計算機に最適配置することは，５次元の最適問題となり現状では多項式時間で最適解を求めることはできない．そこで，本研究では検討した資源の中で，本研究では物理メモリ量についてソフトウェアの限定は行うが，仮想計算機が消費するメモリ量推定のための立式により，安定的に大規模検証環境の構築を行うことができるのではないかと考え，実践した．特に，CPU に関しては，多コア化が進んでおり，またネットワーク帯域に関してもより広帯域化が進んでいる．物理メモリに関しても大容量化は行われているが，ソフトウェアの消費メモリの増大や仮想計算機の多重度の増加により他の資源に比べて律速が早いことが想定される．そこで，本研究ではメモリを対象としたが，より効率的な配置を考えた場合，他の資源に関しても考慮にいれることと最適配置問題の計算量の低減が必要である．

3.15 まとめ

本論文では，仮想計算機環境上で AS エミュレーションを行う際に，各仮想計算機に割り当てるメモリ量の推定を行う手法の提案を行った．本手法では計算量と精度の両立を目指し，資源割り当てアルゴリズムを提案し評価を行った．提案手法により，比較的小規模な環境での実験データを元に規模やトポロジの変化に対応する仮想計算機へ割り当てるメモリ量の推定を行うことができることが明らかになった．また本手法は先行研究に比べると，Deng らや RFC4274 の手法に比

べ明らかに計算量が少なく，Miwa らの手法に比べ精度の良い推定が可能であることを示した．

第 II 部

大規模検証環境構築のための検証施設間の資源および機能の連携

4. 検証施設の資源および機能の連携

4.1 背景

検証施設は世界各国で構築されている．従来はそれぞれ個々で運用されてきたが，近年より多くの資源の利用や IoT のように無線と有線の混在や機器の多様化により，ここの検証施設では検証が困難な事例がでてきた．そこで，検証施設を連携することによりこれらの事例を検証する研究が行われきた．

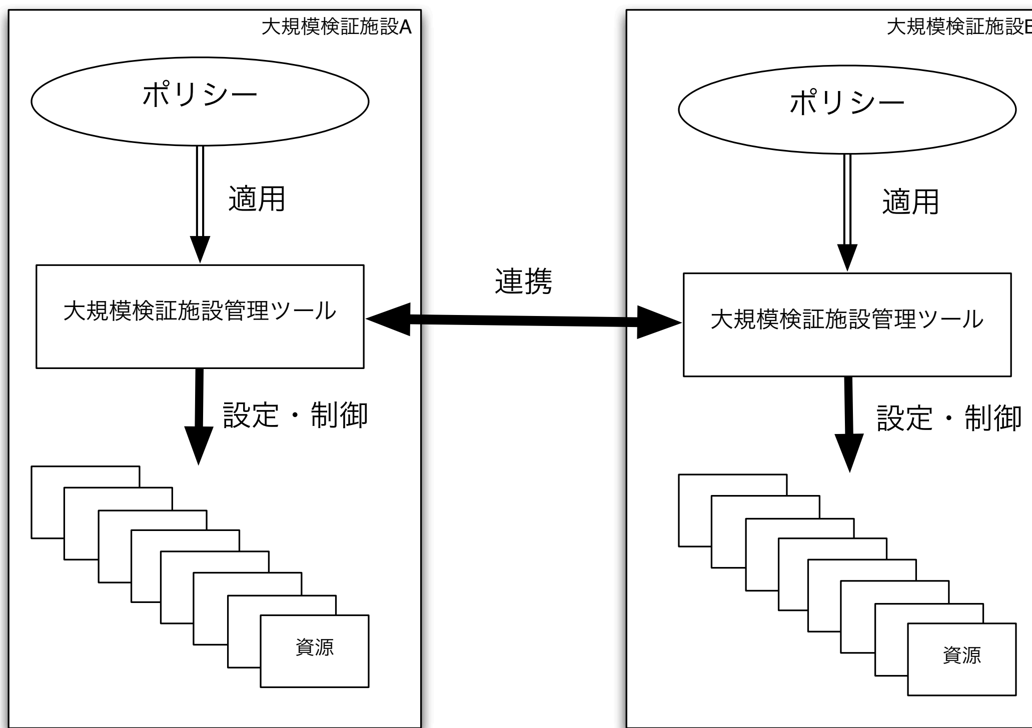

図 9 テストベッド連携の概念

4.2 検証施設の連携における現状と問題点

ネットワークに関する技術の検証を目的とした検証施設は図 9 のように概念的に資源，検証施設管理ツールと運用ポリシーの 3 つの要素から構成されている．検証施設では，検証者は検証施設管理ツールを用いて検証施設が有する資源に対して設定を行う．多くの場合，検証施設は複数の検証者が同時に単一の資源に対して設定の投入や制御を行うことが想定される．そのため，検証者が資源に対して操作できる権限の管理が必要になってくる．また，検証施設ごとに主目的としている検証が存在している．例えば，DeterLab [6] では，ネットワークセキュリティに関する技術やソフトウェアの検証を主目的としており，インターネットなどの外部への接続性に関しては厳しいポリシーを持っている．このように，検証

施設により，同じ検証施設管理ツールを使っているとしても，異なるポリシーで運用されることがある．

検証施設管理ツールは，機能として大きく分けて 3 つに分けることができる．

- 検証施設の資源に対して設定の投入や制御を行う機能
- 資源に対しての検証者の権限の確認のための認証機構
- 検証施設の資源の状態の管理や，検証者への貸し出し状況を管理するデータベース機能

検証管理ツールは大きく分けてこの 3 種類の機能で構成され，検証施設を連携する際はこれら 3 つの機能の連携が必要になる．

検証施設は，有する資源に対して設定の投入や制御を行う機能を組み合わせて，検証施設固有の機能を提供している．例えば，DeterLab の SEER [68] や StarBED の QOMB [69] である．これらは，検証施設管理ツールと一部足りない機能に関して実装することにより，従来の検証施設管理ツールのみでは行うことができなかった検証を可能としている．検証施設を連携する際は，これらの機能を合わせて連携する必要がある．

そこで，本研究では検証施設の連携を資源と機能の 2 つの点から連携することを考える．資源の連携に関しては，ネットワーク的な資源の接続による連携と，検証施設管理ツール同士の接続による資源への環境構築の連携の 2 つにわけられる．本研究では，この連携手法として Collaborative Testbed Federator (CTF) の提案を行う．

機能は検証施設ごとに異なっている．また，検証施設管理ツールが有する機能や設定可能なパラメータが異なるため，他の検証施設の機能を単純に自身の検証施設に移植することは困難である．この問題を解決する手法として本研究では Ditto subsystem の提案を行う．

5. 検証施設連携時の機能の連携

5.1 概要

検証施設は理論の実証や成果物の動作の検証を目的に構築されている検証施設にはネットワークに関する技術やソフトウェアの検証を目的とした StarBED, Emulab や DeterLab などの Emulab-based 検証施設, JGN-X, GENI, FIRE や ORBIT がある．検証施設では実験者が実験環境に使用する PC やネットワークスイッチの初期化や設定の投入を行う作業を補助するために検証施設管理ツールを開発している検証施設管理ツールには StarBED の SprigOS や Emulab や DeterLab などの Emulab-based 検証施設の Emulab-tools, GENI の omni tools や ORBIT の OMF や OML がある．これらの検証施設管理ツールはそれぞれの検証施設が行うことを目的としている実験の内容に従って持っている機能が異なっている

新しいツールの開発や導入により検証施設で行うことができる実験の範囲を広げる研究が行われているしかし，これらの研究はパッケージ化されていないものも多く，検証施設が利用者に貸し出す機能になっていないこともある．よって，このような機能を利用者が容易に実装，または利用者に提供可能とするための仕組みの提供が必要である．

そこで，自身の検証施設で提供されていない機能を実現することを可能とする Ditto subsystem の提案を行う．Ditto subsystem はその検証施設で運用されている検証施設管理ツールを利用しつつ，機能の追加および提供を可能にすることを目的とする．本章では Ditto subsystem のケーススタディとして StarBED 上にトポロジ記述言語である TopDL を用いて実験環境構築を行う．また，TopDL では記述可能であるが，SpringOS ではパッケージ化されておらずユーザに提供されていない機能としてリンクの遅延や帯域幅などの設定を Net-Constellation を Ditto subsystem に機能として加えることにより，Ditto subsystem が機能の追加が可能であることを検証する．

5.2 背景

ネットワークに関するソフトウェアの検証を目的としている StarBED，Emulab と DeterLab はそれぞれ異なるトポロジ記述言語を開発している．StarBED はノードとレイヤ２のトポロジの記述を行うことができる K 言語 [70] を開発している．Emulab のノードとレイヤ２およびレイヤ３のトポロジの記述が可能である NSE [71] を Emulab 用に拡張している [72]．この違いに加え，nse や TopDL ではネットワークに対して遅延や帯域幅の設定を行うことができるが，K 言語ではこれらのパラメータの設定を行うことができない．よって，同種の実験の駆動が目的の検証施設でも，各検証施設で実験シナリオの記述や設定可能なパラメータが異なっており，容易に他の検証施設の環境構築記述を使用することはできない．StarBED で他の検証施設の実験シナリオを駆動する方法として２つの方法が考えられる．

ひとつめのアプローチは検証施設に他の検証施設管理ツールをインストールするアプローチである．この手法は，容易に他の検証施設管理ツールを使用した実験を自身が使用する検証施設で行うことが可能になる反面，検証施設の資源を制御する点で幾つかの問題点がある．PC サーバやネットワークスイッチは検証施設において，他の利用者と共有して使うことになるため，一般的には通常の PC サーバやネットワークスイッチの運用とは異なるアクセス制限が必要になる．従来の検証施設での管理ツールを用いた資源の管理手法では，割り当てた利用者の ID とパスワードをもとに管理ツールで設定を投入する機器に対して権限があるかどうかを確認し，管理ツールが設定を代理して行うことが多い．このアクセス制御とアクセス制御のための機能は検証施設が保有している資源が異なる場合うまく機能しない場合がある．よって，他の検証施設管理ツールを自身の検証施設に導入することは困難である．

２つ目のアプローチは他の検証施設が使用している検証環境の設定ファイルや実験のシナリオ記述を自身の検証施設の管理ツールが実行可能な形式に変換または翻訳する方法である．このアプローチは既存の検証施設管理ツールに修正を加える必要はない反面，自身の検証施設管理ツールがサポートしていない機器や機能に入力するパラメータの投入ができなくなってしまう可能性がある．

他の検証施設が使用している検証環境の設定ファイルや検証のシナリオ記述を解析し，自身の検証施設の管理ツールの制御システムの提案を行う．この手法は，既存の検証施設の機器や機能との互換性を確保した上で他の検証施設が使用している検証環境の設定ファイルや検証シナリオ記述を実行する上で，利点がある．一方，検証環境の設定ファイルや検証シナリオのなかに自身の検証施設管理ツールが設定できないパラメータがある場合，一部の検証環境の構築ができない可能性がある．もし，このような機能が SpringOS の機能の組み合わせで実現可能であるならば，SpringOS を制御して該当機能を実現するための追加機能を加えることにより，他の検証施設の実験シナリオの駆動に不足がなくなる可能性がある．そこで本章では，StarBED に他の検証施設の実験シナリオの入力し SpringOS の手続きに変換するシステムを導入し，かつ SpringOS が有していない機能に関して，StarBED で実現可能であれば追加機能として導入可能な実験シナリオ駆動システムを考える．

5.3 Ditto Subsytem の提案

本章では，StarBED と DeterLab の２つの検証施設を対象として，提案システムの設計を行う．それぞれの検証施設は，StarBED は SpringOS, DeterLab は Emulab tools と呼ばれる検証施設管理ツールを有している．StarBED は，ネットワーク技術やセキュリティに関する検証を目的として構築した．一方，Emulab-based の検証施設である Emulab や DeterLab でも StarBED と同種の検証が行われている．StarBED と Emulab-based の検証施設は同種の検証を目的として構築されているが，検証施設の実験シナリオの記述，検証施設の資源を制御するツールと実験流れに関してのツールの補助の考え方の違いがある．

5.3.1 StarBED と DeterLab の差異

本節では，StarBED で TopDL で記述された検証環境の設定ファイルを用いて検証環境を構築する方法を考える．はじめに，StarBED と DeterLab の差異について

考える．StarBED および DeterLab はネットワークに関するソフトウェア，技術，プロトコルの検証を目的とした検証施設である．これらの検証施設は，異なる実験シナリオ記述言語を使用しており，利用者に提供されている検証施設管理ツールも異なっている．

ここで，両検証施設における検証施設の利用手順を以下に示す．

1. 利用者は検証のシナリオおよび検証環境のトポロジを作成する．
2. 利用者は検証に用いる資源を自身が作成したシナリオ及び検証環境のトポロジに従って検証施設から借り受ける．
3. 利用者は検証環境を自身が作成した検証環境トポロジに従ってネットワークスイッチや PC サーバに対して設定を行う．

実験シナリオの作成は StarBED は K 言語またはシェルスクリプトで記述し，DeterLab では Emulab-tools 用に拡張された NSE で記述する．資源の予約に関しては，StarBED は事前に期間を指定し PC の台数と VLAN ID 数を申請するが，Emulab では実験シナリオを実験のキューに入れておけばシナリオに記述された資源の確保が行われる．

5.3.2 資源予約と割り当て

Ditto subsystem には複数の利用者が利用することを想定すると，Ditto subsystem で実行される実験シナリオごとに資源の割り当てを記録したデータベース (DB) が必要になる．StarBED も DB を持っているが，今回は運用を変更しないことを想定しているため，StarBED のデータベースを使用することはできない．そこで，Ditto subsystem でも DB を持つことで，StarBED から Ditto subsystem で借りているノードと Ditto subsystem に投入されている実験シナリオで使用している資源のマップを撮る必要がある．

5.3.3 Ditto subsystem の要求

本論文では，StarBED で使われている検証施設管理ツールや実験シナリオ駆動システムに変更なしに Emulab NSE の実験シナリオを駆動可能とするシステムを考える．また，StarBED の資源に対しての操作は SpringOS で行うため，Emulab-NSE の実験シナリオを SpringOS の機能として実行可能な形式に変換しなければならない．よって，StarBED に Emulab NSE が入力可能で入力された Emulab NSE は SpringOS の機能として実行可能な形式に変換するコンポーネントが必要になる．

SpringOS の各機能は個別に制御インターフェースを持っており，SpringOS の機能として実行可能な形式に変換された情報をもとに各 SpringOS の機能を制御するためのコントローラが必要になる．さらに，Emulab-NSE で記述可能なパラメータで SpringOS では実現するための機能がないものがある．この機能に対して機能を追加可能とするための Function Adaptor が必要である．

本システムが必要とするコンポーネントは以下の 4 つである．

- Translator
- Testbed Controller
- Resource Mapper
- Function Adaptor

これら 4 つのコンポーネントは既存の StarBED で動いている SpringOS や実験シナリオ駆動システムに対して改変や影響をあたえるものではない．よって，本システムは SpringOS の subsystem とみなすことができる．このように，既存のシステムに改変や影響を与えずに他の検証施設の実験シナリオを駆動可能とするシステムを本論文では Ditto subsystem と名付ける．

Ditto subsystem の処理は 3 つにわけられる．

工程 1

実験環境のトポロジファイルから SpringOS で設定する部分と Function Adapter で設定する部分に分類し，それぞれの実行のためのコマンドを作成する処理である．

工程 **2**

検証施設が運用している資源に関するデータベースから実験環境の構築に十分な資源が利用できるか問い合わ
せ，割り当てた資源とトポロジファイルの対応付を管理する処理である．

工程 **3**

SpringOS と Function adapter を実行し割り当てられた資源に設定を投入し実験環境を構築する処理である．

以下に Ditto subsystem の処理手順を示す．

1. Ditto subsystem に対して設定ファイルを入力する．
2. Ditto subsystem はデータベースに対して設定ファイルに記述された検証環境を構築するのに十分な資源の確保が可能か確認を行う．
3. Ditto subsystem は設定ファイルに記述された検証環境を構築するための SpringOS に対する命令系を作成する．
4. Ditto subsystem は設定ファイルに記述された検証環境を構築するために必要な命令系の中で，SpringOS では対応できないものに関して Function Adapter 用の命令系を作成する．
5. Ditto subsystem で作成された命令系は Testbed Controller および Function Adapter に渡され，それぞれで命令系が実行される．

5.4 TopDL 記述の SpringOS の設定手続きへの変換

TopDL は DeterLab Federation Architecture (DFA) で使用するために，DeterLab によって開発された実験環境記述言語である．[14] DeterLab は，実験環境のシナリオや設定記述は主に OTcl で記述されており，DFA は OTcl の記述を TopDL に変換する機能を提供している．TopDL は XML 記述を採用しており，要素として主に実験に使用するノードのパラメータ記述のための Computer クラスやノード

間の接続関係や，ノード間のリンクのパラメータを記述するための Network クラスで構成されている．TopDL のネットワーククラスは，リンクを識別するための名前の他にリンクの帯域幅や遅延，パケットの通過率の設定をパラメータとして与えることが可能である．TopDL のコンピュータクラスは，実験環境を構成するノードの名前の他に，実験環境構成時に PC にインストールするオペレーティングシステムのディスクイメージファイル名やオペレーティングシステムにインストールするソフトウェアの情報やノードのネットワークインターフェースが所属するネットワーク名をパラメータとして与えることが可能である．

DFA では，TopDL で同じネットワーク名を持つネットワークインターフェースに同じ VLAN ID を与えることで，検証施設上でネットワークの分離を行っている．我々は，TopDL から同一の名前をもつネットワークインターフェースの情報を集め，SpringOS の機能群を使用して検証者が想定するネットワークトポロジと同一のネットワークトポロジを StarBED 上に表現している．

SpringOS では，ノード間のネットワークに対して遅延や帯域幅やパケットの通過率を設定するための機能は提供されていない．そのため，TopDL で設定可能なパラメータを StarBED の実験環境で提供可能とするため，Function Adapter として Net Constellation [73] と呼ばれる機能を開発した．NC は SpringOS の機能と SpringOS で提供していない機能で構成されており，SpringOS で提供されている機能に関しては Testbed Controller 経由で設定し，SpringOS が提供していない機能に関しては直接 PC サーバやオペレーティングシステムに設定を投入することで，リンクの遅延や帯域幅，パケットの通過率の設定を可能とした．Translator は NC が設定を行うパラメータについて手続きを作成し，NC に渡す．

5.5 Testbed Controller による検証施設管理ツールの制御

SpringOS の機能とパラメータに変換された Emulab NSE の実験シナリオをもとに StarBED 上で実験シナリオを駆動する．SpringOS のそれぞれの機能はプロトコルインターフェースを有している．Translator が作成した環境構築のための手続きを実行するために，複数の SpringOS の機能を実行する．例えば，PC サーバに OS をインストールする場合，PC サーバのブートローダや電源，ディスクイメー

ジを HDD に書き込む機能を実行する．

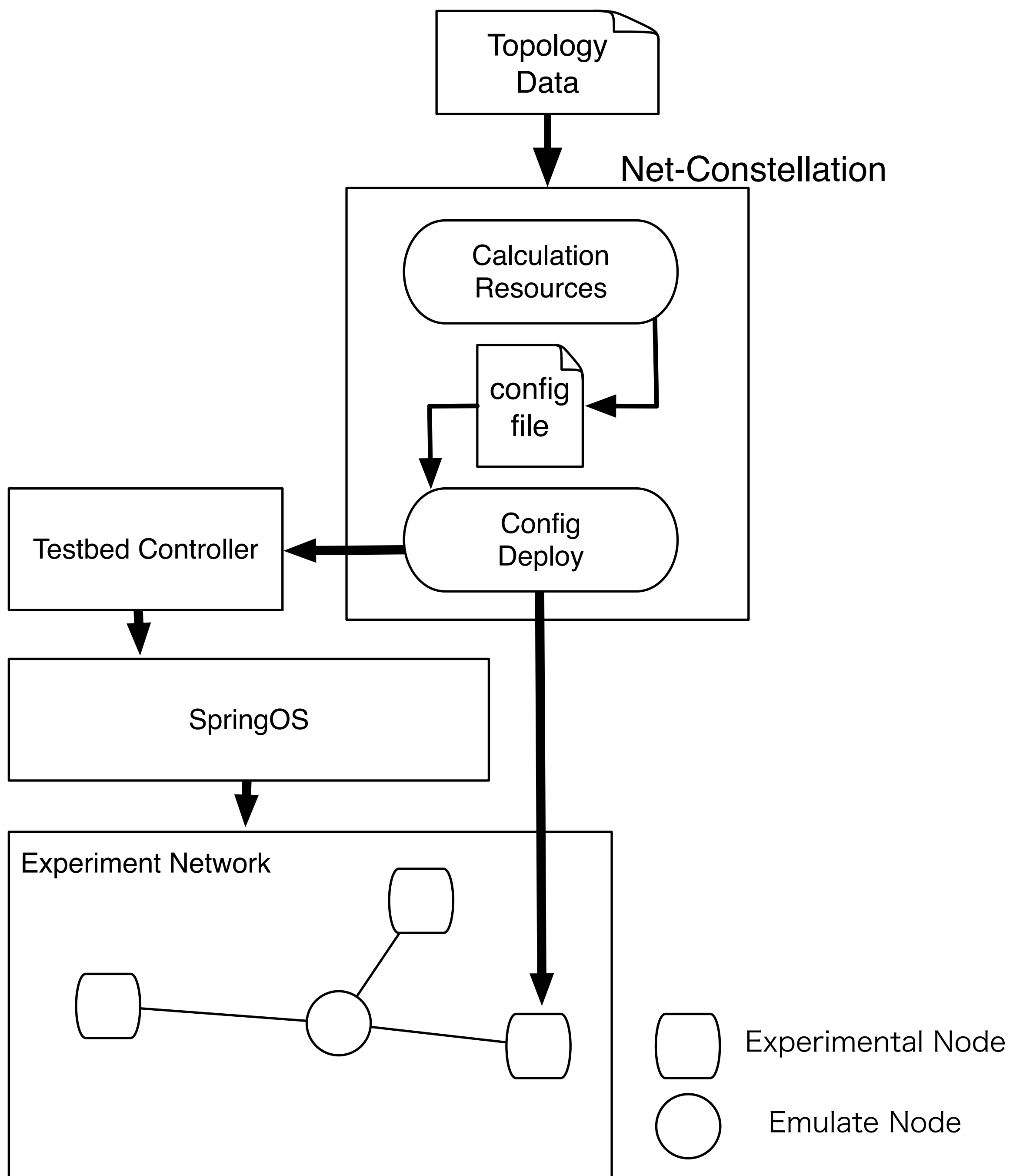

図 11 Fllowchart of Net Constellation

5.6 SpringOS が対応していないパラメータ設定を行うための機能の追加

SpringOS は他の検証施設で設定可能なパラメータを設定するための機能を有していない．例えば，検証環境のトポロジの中で，エッジのパラメータに当たるネットワークの帯域幅や，遅延，パケットのドロップ確率である．これらのパラメータを設定可能にするために，Net Constellation [73] と呼ばれる機能を Ditto subsystem に追加した．Net Constellation(NC) の概要を図 11 に示す．NC は検証トラフィックを流すネットワークに対して netem [74] を用いて遅延，帯域幅とパケットのロス率の設定を行うことができるソフトウェアである．

NC は遅延や帯域幅を experimental ノードの間のネットワークで模倣するために以下の 3 つの処理を行っている．

工程 1

experimental ノードの間に遅延や帯域幅を模倣するために，Ditto subsystem で確保した emulate ノードとは別のノードを確保

工程 2

emulate ノードは experimental ノードの間に配置されるように L2 の設定を Testbed Controller を用いて行う

工程 3

入力された遅延や帯域幅を模倣するために，emulate ノードで netem が実行する

NC は以下の手順で実行される．

1. NC は帯域幅や遅延などの設定すべきネットワークパラメータの値を Translator から受け取る．

2. NC は StarBED の PC サーバを確保し，エミュレートノードとして Resouce Mapper に登録する．

3. NC は設定するパラメータからエミュレートノードに必要なメモリ量を計算する．

4. NC は検証施設コントローラにレイヤ 2 の設定の変更のためのコマンド群を渡し，ネットワークスイッチの設定変更を行う．

5. NC はエミュレートノードにリモートログインし netem の設定を行う．

5.6.1 Resource Mapper

Resource Mapper は，事前に StarBED の資源から利用者に割り当てられていない PC を確保する．確保された PC サーバは StarBED のデータベースは利用中となり，Ditto subsystem のデータベースでは貸出可能な資源と定義される．Resource Mapper は Testbed Controller からの入力された実験シナリオに従った資源の要求があると，割り当て可能であれば割り当てたうえで，DB で貸し出した資源を割り当て可能資源から外す．要求に従った資源の確保ができない場合は，Testbed Controller にエラーを返す．

5.7 議論

5.7.1 Translator

Translator が TopDL を SpringOS の手続きに変更した際に幾つかのセマンティクスの変更が行われている．これは，TopDL が DeterLab Federation Architecture で使用することを前提に設計されているため，幾つかの属性が検証施設連携を前提に設計されているためである．例えば，Emulab-base の検証施設では OS のデフォルトイメージを設定する項目があり，TopDL でも OS イメージを設定する項目が存在する．一方 StarBED ではデフォルトイメージは用意されておらず，この部分は Translator による変換時に我々が用意したディスクイメージ情報に書き換えられる．

5.7.2 Testbed Controller

SpringOS の機能の実行にはいくつかの順番がある．SpringOS の機能を用いて PC に OS をインストールする際に boot に関する設定を事前に入れる必要がある．一方，OS のインストール後は OS の boot 情報を再度変更する必要があり，OS のインストール前後で boot パラメータの変更が必要になる．ネットワークスイッチの設定に関しても，ポートに指定された VLAN ID を設定する前に現在の VLAN ID の設定を一度 clear した上で設定を投入する必要がある．このように，Translator で変換されたコマンドに対して，暗黙的な処理を Controller で行わなければならない．

5.7.3 Resource Mapper

今回のケーススタディでは，PC と VLAN ID 情報のみを Resource Mapper では管理している．検証施設では，PC と VLAN ID 以外にも実験環境または実験シナリオの実行という観点から管理が必要な情報があると考えられる．それぞれ管理が必要な情報が異なってくる可能性があるため，どのようにこれらの情報を統一的に管理するか今後検討する必要があると思われる．

5.7.4 Function Adapter

本研究では，帯域幅と遅延を StarBED 上で再現するために NC を用いた．エミュレーションのシナリオとしては，NC を用いることにより Emulab-base 検証施設と同等のパラメータ設定が StarBED で可能になった一方，DeterLab が独自に開発している SEER などの機能の設定は現状行うことができない．これらを実行可能とするためには，それぞれの機能を Function Adapter として実装しなければならない．

5.7.5 Ditto subsystem

Ditto subsystem は検証施設に機能の追加というかたちで付加価値を与えることが目的である．本論文でのケーススタディから，従来の検証施設管理ツールが持っ

ていない機能を従来のツールに手を入れないで実現することができることを示した．このケーススタディを通して今後の検討事項について議論を行う．

本論文では Ditto subsystem のケーススタディとして TopDL の記述で SpringOS 上に実験環境を構築するために必要な機能を検討し，translator と Function Adapter を作成した．しかし，ORBIT や GENI, Fire ではことなる実験環境記述を持っており，現状の Ditto の仕組みではそれぞれの検証施設ごとに Translator と Function Adapter の作成が必要になってしまう．抽象的に実験環境を記述する記法があれば，それぞれに対応する Translator を作成する必要がなくなり，Dittosubsystem の実装コストを減らすことができる．抽象的な記法を考えるときに必要になってくることはネットワークに関連する実験が持ちうるパラメータの整理が必要であると考えられるため，この点を今後検討する必要がある．

また，今回のケーススタディでは実験環境を構築することに主眼が置かれている．一方，実験の中には実験環境のトポロジに現れないが実験の実行に必要なノードが発生する場合がある．例えば，ネットワークに外乱を発生させるためのパケットジェネレータやソフトウェアにおけるストレッサーがこの事例にあたる．このような機能を Ditto subsystem に加えることは可能であると思われるが，実験シナリオに記述があった場合に Testbed 側で機能にどのように変換するかについては考察が必要である．

5.7.6 Ditto subsystem の仮想化環境への適用

今回のケーススタディでは，仮想化を用いない実験環境を対象とした．これは，TopDL および SpringOS がどちらもベアメタルの PC サーバとネットワークスイッチを対象としているからである．そこで，Ditto subsystem で仮想化を用いた実験環境の構築を考える．

はじめに，検証施設管理ツールに対しての考察を行う．仮想化されたノードやネットワークが記述された場合に現状の検証施設管理ツールに仮想化に対応した Function Adapter を導入する方法がある．２つ目の方法として，検証施設管理ツールを仮想化に対応したものに変更する方法がある．本論文では，StarBED で使われている SpringOS を修正することなく Ditto subsystem を実現することを想定し

たが，Ditto subsystem が他の検証施設管理ツールにも適用が可能であるかについての議論は今後行わなければならない．

次に，実験環境記述に対しての考察を行う．今回使用した TopDL はベアメタルの資源についての記述のみを対象としている．よって，TopDL に仮想化された資源に関する記述を可能とする拡張を行う方法が考えられる．また，TopDL ではなく他の仮想化された資源を記述可能な記述式を採用することで Ditto subsystem で仮想化に対応する方法が考えられる．この場合，他の記述を用いる場合に Ditto subsystem に制約が発生するかどうかについて今後検討を行わなければならない．

5.7.7 異なる記述形式を組み合わせた実験環境記述

異なる種類の実験記述を組み合わせることを考える．はじめに，複数の実験環境記述により構築された実験環境があった場合を考える．異なる記述により構築された実験環境は，実験のシナリオにより組み合わされて新しい実験として行うことできると考えられる．他の考えとして，レイヤ２やレイヤ３の環境が記述されたネットワークトポロジが展開された実験環境に対して，ノードの役割が記述された環境記述を組み合わせた実験の実行も考えられる．これらの場合，複数の実験環境記述を組み合わせることにあらたな実験の創出が行われるが，記述を組み合わせるときにどのように実験環境が組み合わされるか検討しなければならない．

5.7.8 Ditto subsystem の他の検証施設管理ツールおよび検証環境構築記述への適用について

ここでは，他の検証施設管理ツールおよび検証環境構築記述への適用について述べる．例として検証環境構築記述に関しては ProtoGeni の Rspec や Planetlab の構築記述について適用の可能性について述べる．また，検証環境構築ツールに関しては，Emulab の Emulab tools や Geni の Omnitool に関して適用の可能性について述べる．

5.8 まとめ

本節では Ditto subsystem を提案したことにより，従来の検証施設管理ツールを変更することなく，新たなツールでの検証施設の機能拡張を可能にできることの一例を示した．このケーススタディでは，DeterLab が採用している TopDL の記述を用いて，StarBED 上に検証環境の構築を可能とした．さらに，NC を Function Adapter の例とすることで，SpringOS の機能だけでは設定できないないパラメータであるネットワークの遅延や帯域幅を StarBED の機器を用いて構築可能とした．このときに，SpringOS のコードや運用には変更を加えずに実施した．よって，Ditto subsystem は現状の検証施設管理ツールや運用を変更または修正なしに他の検証施設と機能を揃えることを可能とする subsystem であると示すことができた．

6. 検証施設連携時の資源の連携

6.1 概要

ネットワークに関するソフトウェアや技術, プロトコルの検証を目的として様々な検証施設が構築されている. これら検証施設は実施する検証の目的によって最適化されて構築されている. 各検証施設は, 目的としている検証をサポートするために適した運用形態や運用ツールを開発している. そのため, それぞれの検証施設で提供している資源や機能は標準化されておらず, また資源の設定や使用方法も異なるため, 資源を複数の検証施設から借り受けて検証環境の構築を行うと, それぞれの検証施設のやり方にしたがって設定や接続を行わければならない. この問題を解決するために, 検証環境の連携のためのアーキテクチャが提案されているが, 従来のアーキテクチャの多くは同一の管理ツールで管理された検証施設間の連携に主眼が置かれており, 異なる管理ツールを使用している検証施設間にまたがった検証環境を構築するためにはいくつかの問題点がある. 本研究では, 異なる管理ツールを使用している StarBED と DeterLab を対象として検証施設にまたがった検証環境の構築手法の提案を行う.

6.2 背景

6.3 検証施設の資源管理手法の現状と連携を行う上での問題点

現状要求されている検証事例とそれぞれの検証施設での検証の可能性について述べる. 近年の検証事例とテストベッドの乱立状況とテストベッド連携の事例を議論する. ここで新規にテストベッドを構築するのではなく既存のテストベッド同士の連携により実現可能な事例について述べ, テストベッド連携の必要性を述べる. テストベッドの連携について, 現状はそれぞれ独自に進化している管理ソフトウェアがあり, 互換性がなく容易に連携ができない点を述べた上で, この問題を解決する提案手法の説明を行う.

各検証施設は, 異なる運用形態や運用ツールで管理されている. そのため, 検証施設連携を行う方法として, 既存の運用形態を維持しつつ検証施設連携を行う

方法と，検証施設連携用の運用形態や運用ツールに移行する二つの方法が考えられる．前者は既存の運用形態を維持するため検証施設のオペレータにとって容易に検証施設連携の運用を行うことができる反面，検証施設連携を行う検証施設の運用形態によっては検証施設連携自体が困難になってしまう場合が考えられる．後者は検証施設連携を行うのに適した運用形態に検証施設の運用方法を変更することになるため，検証施設連携を行うことは容易である．しかし，検証施設の独自性が失われてしまう可能性がある．本論文では，検証施設連携を行う方法として，既存の運用形態を維持しつつ検証施設連携を行うことを目的とする．

次に既存の運用形態を維持しつつ検証施設連携を行う上で必要な機能について以下で述べる．検証施設を使用するときにはじめに行うことは実験を行う上で必要な資源を確保することである．連携した環境では複数の検証施設の資源を確保しなければならない．よって，検証施設連携アーキテクチャでは利用者の資源確保を助ける機能をもつ必要がある．検証施設連携では実験に使用する資源が複数の検証施設上に存在する．検証施設ではそれぞれ独自のユーザ認証システムをとっている．検証施設のユーザ認証システムは資源の貸出や資源のアクセス制御にも用いられているため，これを変更することは検証施設の管理の大きな部分を変更する事になってしまう可能性がある．そのため，ユーザアカウントの差異を吸収する機能が必要になる．最後に，検証施設連携の環境では実験に用いるトラフィックが流れるネットワークを接続しなければならない．前に述べたが，検証施設では資源はインターネットから隔離されているためインターネットを通した接続には特別な方法が必要になる．よって，検証施設連携アーキテクチャでは検証施設同士の実験トラフィックが流れるネットワークの接続方法の提供が必要になる．

これらの検証施設連携アーキテクチャに必要な機能を提供する上では，次のトレードオフが存在する．トレードオフは，ユーザインターフェースの自由度，検証施設側のシステム構築形態の自由度，検証施設側の管理形態の自由度，検証施設固有の機能の利用と運用および開発にかかるコストである．

本論文では検証施設のシステム構築形態や管理形態を改変せずに検証施設連携を実現するアーキテクチャとして，Collaborative Testbed Federation (CTF) を提案する．CTF はユーザが実験シナリオを入力する利用者側の翻訳機と入力されたシ

ナリオを解釈し検証施設にシナリオに基づいた環境を構築する検証施設側の翻訳機の２つの翻訳機から構成されている．これにより，ユーザが使用するユーザインターフェースを選択可能となり，また検証施設の管理形態やシステム構築形態を変更することなく検証施設連携を行うことが可能となる．さらに，本論文ではCTF を用いて StarBED と DeterLab の間で検証施設連携を行い，その結果構築した環境上で行った実験について述べる．

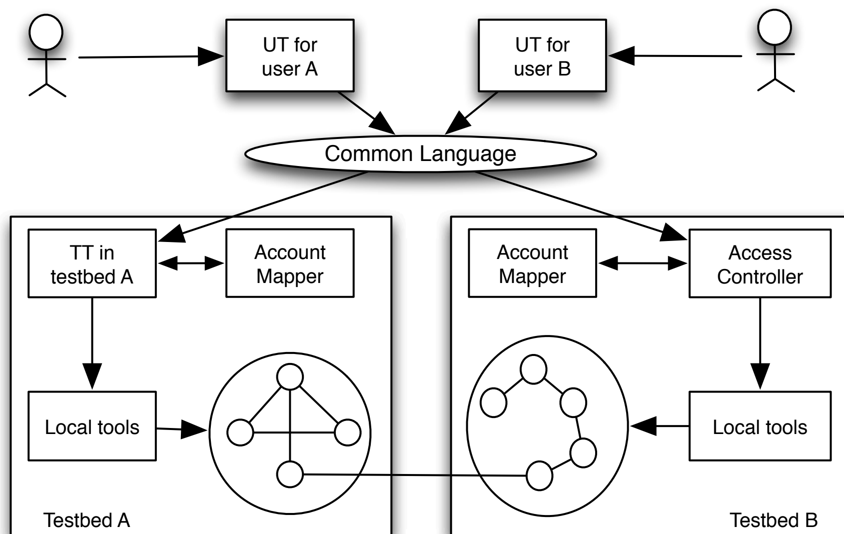

図 12 CTF architecture

6.4 CTF の設計

検証施設には運用形態により，システムによる検証施設連携が困難であるため，オペレーションにより解決しなければならない部分が存在する．そのため，はじめにあらかじめオペレーションにより解決しなければならない部分を知るために，検証施設の違いをまとめる．

運用方針や運用方法について，StarBED と DeterLab では以下の違いがある．一点目は資源の予約方法である．資源の予約はそれぞれ，StarBED のように事前に使用する資源と期間を決める方法と DeterLab のように実験を行う時に使用可能な

資源を借りる方法である．二点目は，ユーザアカウントである．アカウントは例として，DeterLab ではユーザ個人がそれぞれユーザ ID をもち，ユーザ ID の認証にはユーザパスワードを用いている．資源の借用や資源へのアクセスはユーザ ID とプロジェクト ID を関連付けることでアクセス制御を行なっている．一方 StarBED ではユーザ ID，パスワードとプロジェクト ID で管理されているが，ユーザ ID は各プロジェクトでひとつしかなく，すべての実験者が共通のユーザ ID，パスワードとプロジェクト ID を共有している．三点目は資源の利用制限である．資源の利用制限の例としては，DeterLab では OS はあらかじめ用意されたイメージを利用するのに対し，StarBED ではユーザが好きな OS を使用することが可能である．また，ネットワークに関しては DeterLab では VLAN ID はユーザには隠されており，ユーザが入力した実験トポロジをもとにシステムが自動的にネットワークの分離を行う．一方 StarBED では VLAN の ID 空間を切り分けてユーザに提供しており，ユーザは割り当てられた ID 空間を利用してネットワークの分離を行う．これらは，検証施設間の本質的な違いであり，これらの違いを吸収する機能を設計する必要がある．

この中で，資源の予約については StarBED では自動的な資源確保を運用方針として許可しておらず，必ず研究員との調整が必要になるためシステム的な解決は困難である．そのため，今回の CTF ではこの部分についてはオペレーションによる解決が必要になる．資源の予約以外の二つについてはシステム的な解決を行う．

図 12 に CTF の概要を示す．CTF では，検証施設アーキテクチャ，ユーザインターフェースの差異を吸収するために，二種類の翻訳機を用いる．翻訳機は，利用者側の翻訳機である User side Translator (UT) と検証施設側の翻訳機 Testbed side Translator (TT) になる．UT はユーザから入力された実験記述を TT が理解可能な言語への翻訳を行う．TT では UT が翻訳した言語を入力として，実験ノードおよびネットワークの設定を行う．TT は入力に対して，TT が管理する検証施設では対応していない機能の場合は記述を無視して設定を行う．

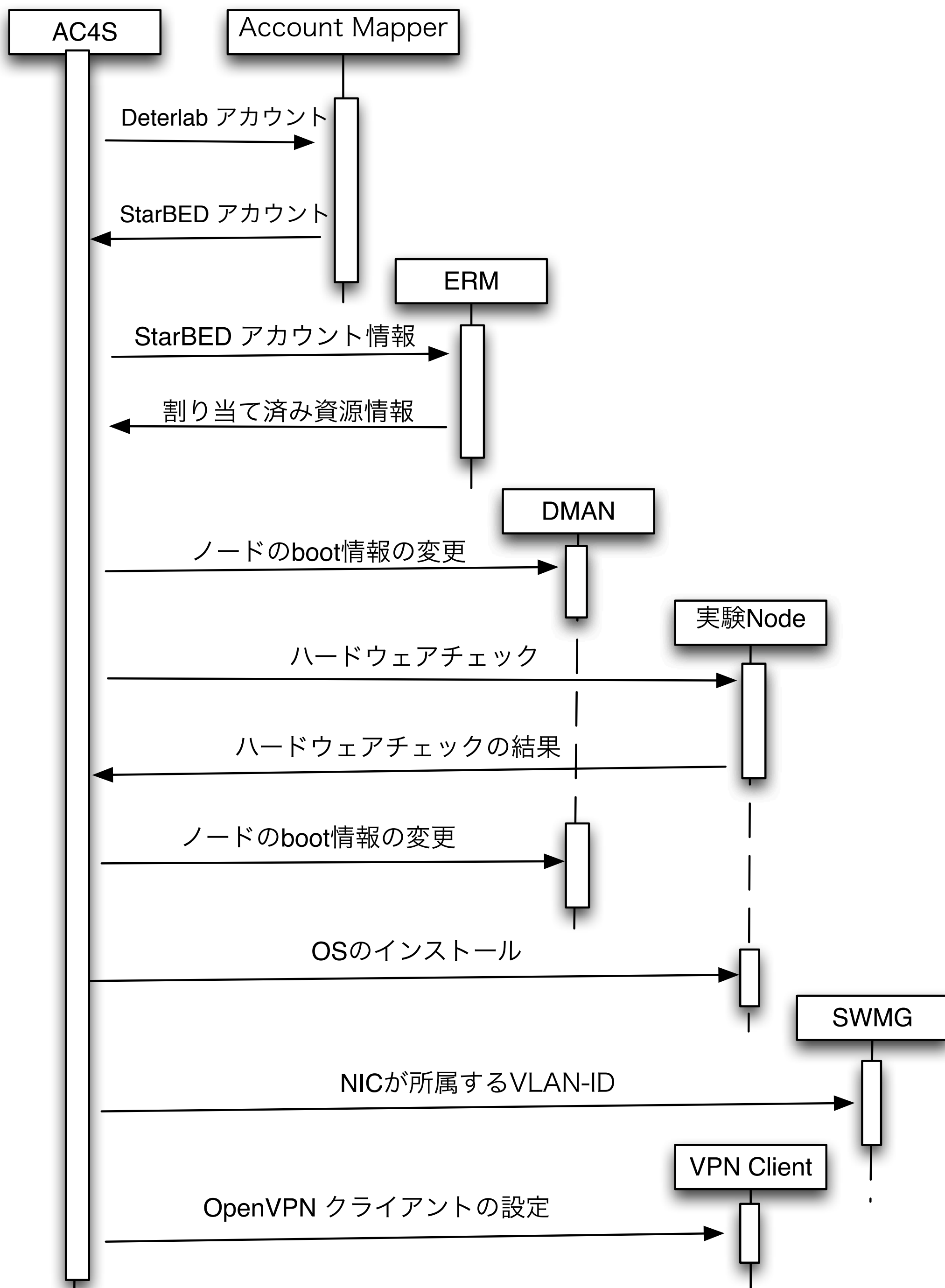

図 13 TT4S の処理の流れ

6.5 TT4S の設計

本節では，CTF の StarBED 側の翻訳機である Testbed side Translator for StarBED (TT4S) の設計を行う．TT4S は CTF の中で StarBED で用いる検証施設側の翻訳機にあたる．

TT4S を設計するに当たり，いくつかの設計上の制限を述べる．一つ目は，実験者は各検証施設が保持している資源や検証施設固有機能を熟知していることである．これは，CTF では実験者がユーザインターフェースの選択や改変することを前提としているためである．二つ目は，実験者があらかじめ検証施設の資源を予約しておくことである．これは，前節で述べたように，資源予約についてはシステムでの解決が困難であるからである．よって，本論文ではあらかじめ資源が使用出来る状態に対して，資源の設定を行うアーキテクチャの提案までを行う．三つ目は，各検証施設が提供している固有機能は，その固有機能を保持する検証施設の資源でのみ利用可能とすることである．他の検証施設で別の検証施設固有機能を有効にする場合，固有機能によっては検証施設の資源の機能制限や運用制限に触れる可能性が考えられるためである．これらの制限を設定した上で，TT4S の設計を行う．図 13 は TT4S の処理の流れを表している．それぞれの機能の詳細を以下で述べる．

6.5.1 アカウントマップ機能

はじめにアカウントの差を吸収する機能であるアカウントマッパーについて述べる．6.4 で述べたように，StarBED と DeterLab ではユーザの認証方法が異なっている．よって，TT4S では StarBED と DeterLab のアカウントの対応を保持させる．これにより，UT で DeterLab のアカウントを用いて StarBED の資源の確保や実験環境の構築が可能になる．

6.5.2 資源の確保と動作確認機能

次に実験に用いる資源の確保の方法について述べる．StarBED ではあらかじめ実験に用いる資源を確保し，使用期間内になることで資源が使用可能になる．ま

た，StarBED では資源を貸し出すときに資源がハードウェアに問題がないか確認しないため，実験者には実験に使う資源量より多めの資源を予約することを推奨している．よって，あらかじめ実験に使う資源よりも多めの資源が確保されていることを前提にする．

TT4S では実験者側の翻訳機である UT(今回は DFA の EC である) から実験に用いるノード数が送られてくる．送られてきたノード数のノードをあらかじめ確保されているノードから選び出すが，使用するノードを選び出す前にハードウェアが問題を抱えていないかどうかの確認を行う．StarBED では OS の起動をネットワークブートである pxeboot で行う．そこで，ノードの起動確認としてノードを実験に割り当てる前に pxeboot で knoppix を起動し，knoppix が起動するかおよびすべてのネットワークインターフェースが knoppix から確認できるかの確認を行う．

6.5.3 StarBED 内の実験環境構築

使用するノードが確定したあと，TT4S は実験シナリオに基づいてノードおよびスイッチの設定を行う．OS のインストールについては，現在 TT4S でインストール可能な OS は Debian，Knoppix または tftpboot が可能な OS となっている．Debian はハードディスクにインストールしハードディスクから起動するが，knoppix と tftpboot に関してはネットワークブートになっている．スイッチの設定については StarBED の管理ツールである SpringOS の機能の一部である SWMG を使用している．SWMG にユーザ情報，VLAN-ID とノード名を渡すことで VLAN をポート VLAN で設定することができる．TT4S はアカウントマップ機能で変換された StarBED のアカウント，TT4S が実験に割り当てたノード名と TT4S が解釈した実験シナリオに書かれたネットワークのトポロジをもとに SWMG を用いてネットワークの分離を行う．

6.5.4 実験ネットワークの接続

最後に，実験トラフィックが流れるネットワークの接続の方法の説明を行う．実験では broadcast パケットが流れるようにレイヤ２でのトンネリングが要求される．そのため，CTF では OpenVPN を用いたレイヤ２の仮想プライベートネットワークを構築する．OpenVPN で接続するためには，実験に使用するノードの一部がグローバル IP アドレスを持ち，インターネットに直接に接続していなければならない．DeterLab は数台のグローバル IP アドレスを持った portal と呼ばれるノードが利用可能である．StarBED でも実験ノードにグローバル IP アドレスを設定することも可能であるが，設定にはオペレータによる作業が発生してしまう．ただし，HTTP プロキシを経由した接続に関しては StarBED の運用方針で許可されている．よって，TT4S では実験に用いるノードとは別に実験トラフィックが流れるネットワークを DeterLab と接続するために，OpenVPN のクライアントとして設定する．

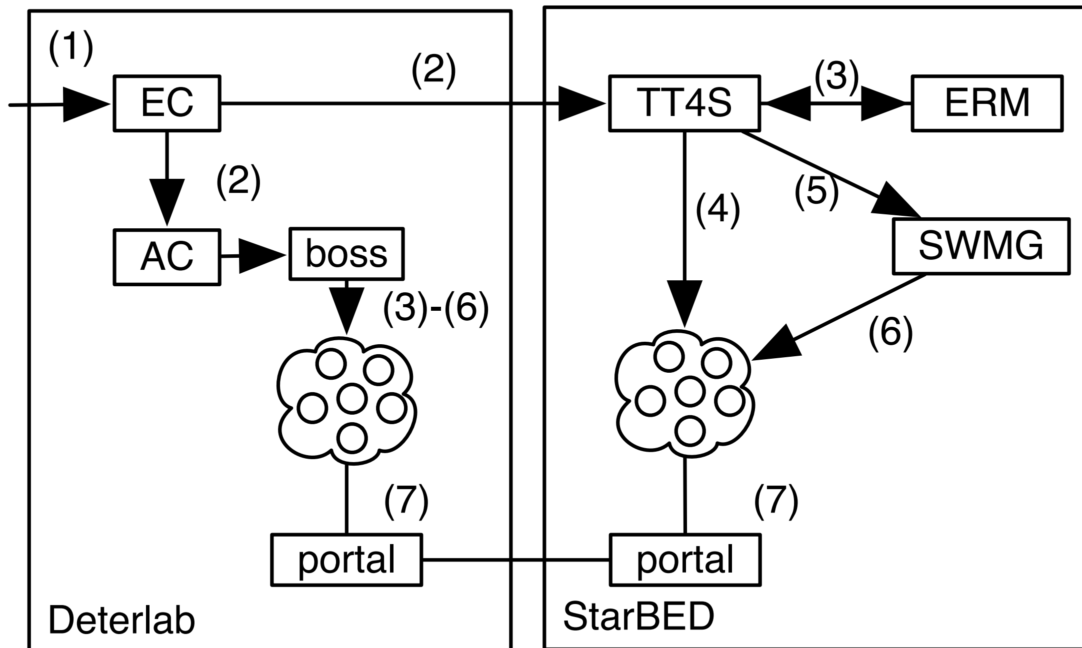

図 14 StarBED および DeterLab での環境構築の流れ

6.6 StarBED と DeterLab の連携例

今回提案した CTF のユースケースとして，StarBED と DeterLab 間の検証施設連携を行った結果について述べる．今回のプロトタイプ実装では，CTF の中でユーザ側の翻訳機である UT として DFA の EC を利用している．また，UT と TT 間で翻訳する言語として DFA で開発された TopDL を用いた．今回は，DFA の EC を利用するため，実験シナリオを入力する際に用いられるユーザアカウントは DeterLab で使用しているアカウントとなる．よって，DeterLab 側の AC ではアカウントマッパーの必要がないため，DeterLab の AC は DFA で設計された AC をそのまま利用することが出来る．最後に TT4S の実装を行い，DFA の EC と通信可能な状態にすることで検証施設連携環境を構築することが可能な環境が整う．

6.6.1 EC で指定可能な設定情報

UT に入力する情報は大きく分けてノードに関する情報とネットワークに関する情報になっている．ノードに関する情報は，ノードにインストールする OS の種類，ノードを保持している検証施設名とノードの種類になっている．ノードの種類とは各検証施設で定められたノードのグループの名前となっている．ネットワークに関する情報は，ネットワークの名前，ネットワークに接続しているノード，帯域と遅延の情報となっている．この中で，StarBED では帯域の制限や意図的な遅延を発生させることは通常行うことができない．そのため，TT4S では帯域と遅延の情報については，UT から数値の指定があったとしてもそれを無視している．

6.6.2 StarBED および DeterLab での実験環境構築手順

まずはじめ，ユーザは自分が使用する実験シナリオを EC に入力する．UT は入力された実験シナリオを TopDL の記述に翻訳を行う．TopDL に翻訳された実験シナリオはユーザが使用する検証施設の DeterLab の TT および TT4S に転送される．

TT4S では 13 の手順で実験環境構築が行われる．DeterLab 側の AC では，実験シナリオで指定された環境の情報を Boss サーバに渡す．Boss サーバは実験シナリオに記述された環境を構築するために十分な資源が確保できるかどうかを判断し，確保できる場合は実験環境の構築を行う．Boss は指定された資源を確保できない場合，もしくは実験シナリオに指定されたパラメータが設定不可能な場合は TT は UT にエラーを返し，実験環境の設定を終了する．

6.6.3 検証施設間ネットワークの構築

今回の実装では，StarBED と DeterLab 上に構築された実験環境の接続の自動化は行なっていない．今回は DeterLab が提供している portal を OpenVPN のサーバ，StarBED のノードの一台を OpenVPN クライアントとし，StarBED のノードから DeterLab の portal に接続することで実験トラフィックが流れるネットワークをレイヤ 2 VPN で接続している．DeterLab の portal は OpenVPN サーバとして自動的に起動し StarBED の OpenVPN クライアントからの接続を待ち受けている．しかし，TT4S が DeterLab の portal の情報を取得する事ができないため，StarBED 側の OpenVPN クライアントの設定を行うことができない．portal のグローバル IP アドレスを取得できない理由は，TT4S は DeterLab の EC とは接続しているが，DeterLab の AC とは接続することができないためである．そのため，今回は自動的に設定可能な部分がすべて出来上がったあとで，MyDETERlab で portal のグローバル IP アドレスを確認し，手動で StarBED の OpenVPN のクライアントの設定を行い portal に接続を行なっている．

6.7 議論

CTF は検証施設が各自の管理形態や運用方針を変えずに検証施設連携を実現するために制限やオペレーションでの解決を試みている．

表 3 StarBED と DeterLab のノード間のスループット

	D to S		S to D	
Average[Mbits/sec]	2.21	2.39	3.68	3.86
Variance	0.06	0.03	0.04	0.05
Standard Deviation	0.25	0.18	0.21	0.22
Maximum[Mbits/sec]	2.44	2.61	3.86	4.10
Minimum[Mbits/sec]	1.59	2.09	3.28	3.46

表 4 StarBED と DeterLab 上のノード間のジッター

	D to S		S to D	
Average[ms]	4.69	4.69	6.95	6.95
Variance	0.28	0.28	14.30	14.31
Standard Deviation	0.53	0.53	3.78	3.78

6.7.1 資源確保の方法について

一つ目の制限は FTA は資源の確保の方法についてである．先に述べたように各検証施設で資源の確保の方法や貸し出し形態は異なっている．StarBED のように，あらかじめ資源量と期間が決まるような方法は，実験計画を立てる際に詳細な実験計画を立てることや大規模な実験の計画を立てることが容易である．一方，ノードの予約を予め行わなければならないため，急遽必要となった実験を行う場合は数日間またなければならない可能性がある．DeterLab のように，資源に空きがある場合に実験シナリオに応じた資源をユーザに割り当てる方法では，急遽必要となった実験に関して資源に空きがある場合に限り実験が可能である．ただし，実験の期間が明確に定められないため，資源の回収のポリシを明確に定義しない場合，資源が開放されないという事態が想定される．また，この方法の場合大規模な実験を行うのに難があるといわれている [75]．今回は StarBED であらかじめノードの確保をしておき，ノードの利用期間内で DeterLab の資源を確保することで検証施設連携を行った．この方法によりそれぞれのノードの確保の方法

を変えることなく検証施設連携が可能になった．DeterLab と StarBED 以外では，Emulab-base の検証施設，Planetlab や Onelab では DeterLab のような資源の確保方法をとっているため，この方法であれば DeterLab と StarBED 以外の検証施設とも検証施設連携が可能であると考えられる．

6.7.2 資源のコントロールについて

CTF は，資源の操作を行うインターフェースは持たずに，各検証施設で提供する操作方法をユーザに利用することを想定している．StarBED では ssh 等によるリモートログインを行うための踏み台サーバが提供されている．さらに，StarBED では実験者がノードへ OS をインストールすることを許可しているため，Remote KVM を実験者に提供している．一方，DeterLab ではユーザがサーバに対して OS をインストールすることを許可しておらず，予め用意されている OS イメージをインストールした状態でユーザにサーバを貸し出す．そのため，DeterLab では remote KVM の機能はユーザに提供されておらず，ssh を用いた遠隔ログインでサーバの管理を行う．StarBED，DeterLab の両検証施設ともにノードを管理するためのネットワークと実験トラフィックを流すためのネットワークは隔離されている．ssh によるリモートログインのみで良い場合は，実験者が入力した実験環境のシナリオに 1 つ管理用のネットワークを加えて環境構築することで，1 つの踏み台からすべてのノードにアクセすることは可能である．しかし，この方法では実験トラフィックを流すネットワークに対して管理用トラフィックを流してしまうことになるため，一部実験用トラフィックと管理トラフィックが混在するネットワークが存在する可能性が発生してしまう．

6.7.3 検証施設間の実験ネットワークの接続について

検証施設間の接続に関しては現在手作業で行なっている．これは，FTA では TT からの情報を UT で受けとることを想定しておらず，検証施設で行った設定を TT や他の検証施設の TT で把握できないため，VPN で接続する場合の接続先の情報が得られないためである．

CTF を用いて構築した StarBED と DeterLab の VPN サーバ，クライアントの間で帯域計測を行った．計測は 10 回行い，帯域計測の結果を表 3，ジッターの計測結果を表 4 に示す．その結果，日本とアメリカでインターネットを利用した VPN では 2Mbps から最大でも 4Mbps しか帯域がない．また，ジッターも大きいため実験内容によっては実験の信頼性が損なわれる可能性がある．ただし，狭帯域でもよくジッターが大きくても問題ない実験や，実験環境内部にインターネットで起きるような外乱を発生させる実験であれば，実験規模の拡大やさまざまな資源を利用した検証施設連携環境が構築できる．また，検証施設間に安定した広帯域なネットワークを構築するためには，DeterLab のように専用線を検証施設間に構築することや JGN-X [11] や Internet2 [10] のようなインフラストラクチャを提供する検証施設を利用することが考えられる．

6.7.4 CTF の他の検証施設との連携の可能性について

本研究では，はじめに検証施設について概念的に整理を行い，その上で連携手法について提案を行った．多くのテストベッドに関しては，検証施設管理ツール，運用ポリシーと資源で構成されているため，提案手法を用いた連携は可能であると考えられる．しかし，いくつかの条件では連携ができないことが考えられる．一つ目は，物理的に資源同士が接続できない場合である．物理的に接続できない場合とは，例えば検証施設の資源が外部のネットワークに接続することが物理的もしくは運用ポリシー的にできない場合や一方の検証施設の資源は無線での接続を想定している一方で，他方は有線の接続しかそうていしていない場合である．これらの場合は提案手法では検証施設の連携を行うことができない．二つ目は，検証施設の資源への設定投入において人間による操作が入る場合である．提案手法では，検証者が記述したシナリオを中間言語にした上で，それぞれの検証施設管理ツールが解釈可能な形式に変換している．提案手法では，この一連の流れにおいて，人間が介入することを想定しておらず，人間への設定依頼を行うことはできない．そのため，一連の設定が終了後，人間による設定が必要な部分に関しては，設定が未実行となり環境の構築が失敗したとみなされてしまう．三つ目は，運用ポリシーの衝突による場合である．運用ポリシーは検証施設ごとに異なって

いる．そのため，機器への設定可能なパラメータは同一の機器を使っている場合でも異なってしまう．そのため，一方の運用ポリシーにより行うことができない設定やそもそも外部検証施設との連携を許可しないポリシーを有する検証施設の場合は，連携を行うことができない．

6.8 まとめ

本論文では，StarBED と DeterLab 間の異種検証施設連携の実例について述べた．今回は，実際に DeterLab で運用されており，Emulab-base の検証施設を対象とした検証施設連携アーキテクチャである DFA を StarBED に適用することで StarBED と DeterLab の検証施設連携を行うことを目指した．しかし，StarBED と DeterLab はそれぞれ異なる運用方針を持つ検証施設であり，このような検証施設で連携を行う際には資源に対するアクセス制御，検証施設で許可されている資源への操作や検証施設の資源の外部への接続についての制限の問題があり，これらを解決する必要がある．これらを解決する方法として本論文では StarBED の運用方針に合わせた検証施設連携を行うものとして TT4S を設計した．我々は，TT4S を用いて StarBED と DeterLab の両方の資源を用いて実験環境を構築し，構築した環境で複数の実験を行うことで TT4S の有効性を確認した．

7. 議論

本論文では，情報工学におけるネットワークに関するプロトコルやネットワークにより情報のやり取りを行うソフトウェアや分散処理を行うソフトウェアの検証環境の中で，実際のソフトウェアや機器を使用して検証を行うエミュレーションにおいて，大規模な検証環境を構築する手法について 2 つのスケールアップとスケールアウトの観点で手法の提案を行った．本章ではシミュレーションとエミュレーションを組み合わせた検証手法についてと他分野への応用の可能性について述べる．

7.1 仮想計算機を用いた検証環境構築時の忠実度

本研究では有限の計算機資源の中で検証に用いるノードを効率よく増やすためのアルゴリズムの提案を行った．本手法では仮想計算機を用いることによって，単一の計算機資源に対して検証ノードを多重化して動作させることにより，計算機資源の量による検証規模の制限の緩和に取り組んだ．

仮想化では PC のハードウェア資源を複数の計算機から使用可能にする技術であるため，PC 上で動作するオペレーティングシステム (OS) とハードウェアの間にソフトウェアが動作することになる．また，複数の OS からハードウェアにアクセスが起きるため，仮想化ソフトウェアはこのアクセスのスケジューリングを行うことになる．そのため，仮想化なしに PC に OS を導入した場合と挙動が異なる可能性が高い．そのため，検証を行っているソフトウェアが展開される環境とハードウェアの性能面だけではなく，OS とハードウェアの間の動作面においても異なる．よって，検証環境と展開環境の類似度もしくは忠実度を図る尺度の検討が必要になる．

OS が認識する時刻に関しても PC 上で動作する仮想計算機の仕事量が多くなるとずれが大きくなることがわかっている．大規模な検証環境での検証においては，検証ノードが多くなるため検証を行うものがすべてのノードでおきていることを検証時に把握することは困難である．そのため，大規模な検証環境での検証では，それぞれのノードからログとしてノードの状態や状況の変化を記録してお

き，検証後に解析することが一般的である．このときに，おきた現象を時系列に把握することになるが，上記のような時刻のずれが発生するとログの順序が時系列に整列せずに正確な解析が行えなくなってしまう．本研究では，この問題に対しては単一のログ収集サーバを用意し，それぞれのノードはログ収集サーバに向けてログを送信し，時刻はログ収集サーバで打刻することにより，ノードの時刻のずれが検証に影響を及ぼさないようにした．しかし，単一のログサーバでは同時に受信可能なログの量に限界があるため，より大規模な環境の検証を想定するとログサーバを分散させる必要がある．この時，分散ログ収集サーバの時刻動機をどうするか，もしくは検証ノード自体で時刻のずれが生じない技術の開発が必要である．

大規模検証環境における検証ノードの制御に関して最大で 10000 台規模の検証環境にて検証ノードの制御を行った [76]．ここでは，模倣インターネットの構築を目的として実際にインターネット上で観測された情報からインターネットの経路情報を生成し，それを検証施設上で Border Gatewat Procol(BGP) の経路交換を行うソフトウェアである Quagga を用いて StarBED 上に BGP 網を構築した．模倣インターネットは，インターネットでの動作を想定したソフトウェアに対して，検証時により実環境であるインターネットのネットワーク特性に近いネットワークでの検証を目的として行わてれいた．本手法により，ソフトウェアが動作するネットワークは，実インターネットに対してより忠実さを持ったネットワークの検証が可能になった．一方，インターネットの規模は単一の検証施設が有する計算機資源上に構築することはできず，観測データから作成したトポロジから一部を切り出して環境の構築を行っていた．他には，仮想計算機により単一の計算機に対して複数の検証ノードを多重起動することで，計算機資源の量の制限を緩和した検証環境の構築をおこなってきた．しかしながら，検証施設に構築するネットワークを実インターネットに近づけるために規模を大きくしていくとさまざまなところで資源の抽象化が行われることになる．よって，現状では検証規模を上げることによりネットワークに関しては忠実さの向上が見込める一方で，検証環境側では資源の抽象化による忠実さの減少が発生してしまう．

本研究では，資源の抽象化による検証環境の忠実さの減少と規模拡大によるソ

フトウェアの動作ネットワークの忠実さの向上のトレードオフに関する課題については未解決の問題となっている．

7.2 複数の大規模検証施設にまたがった検証環境の監視および観測

本研究では，大規模検証施設にまたがった検証環境の構築手法の提案を行った．本研究では，資源の設定と制御に関して連携する手法について提案を行った．しかし，先に述べたように大規模な検証環境での検証を行う場合は，各検証ノードのログの収集が重要になってくる．さらに，検証に障害が発生した場合，原因を探る上で，検証ノードの間のネットワークで何がおきているか調査することも重要になってくる．大規模検証施設では，運用のポリシーのため取得可能なログの種類や取得方法がそれぞれの施設で異なってくる．そのため，検証者は複数の施設にまたがった環境での検証を行う場合，それぞれの施設で収集可能なログの種類や取得方法に対応してログを収集しなければならない．

大規模検証環境を連携し，複数の施設にまたがった検証環境を構築する場合，施設間の資源をネットワーク的に接続する方法として，インターネットを通す場合と，施設間を繋ぐ専用線を通す場合がある．本研究で対象とした StarBED と Deterlab では専用線を有しておらず，Virtual Private Network(VPN) を用いてそれぞれの大規模検証施設の資源の接続を行った．一方，例えば StarBED と Jose [77] は JGN-X で接続されており，インターネットを経由せずに検証ノード間で通信を行うことができる．インターネットを通して通信を行う場合，通信帯域がこの部分だけ他の通信路に比べて著しく細くなってしまう．そのため，この通信路において，検証に使用する通信以外を流すことが難しく，統一的な監視，観測ができない．そのため，検証者はそれぞれの検証施設で個々に監視および観測のための機能を設定しなければならない．その一方で，専用線で施設間が接続している場合，この通信路は十分に広帯域なことが多く，上記のような検討を行う必要はなく，どちらか一方の施設で監視および観測を行うことができる機能を設定すれば良くなる．

ここでは，我々が行ったデモで得られた知見について述べる [76]．この時，ログ収集として 2 種類の手法が考えられる．

- それぞれの検証ノードにログを保存し，検証終了後ログを集める．
- ログ収集サーバを用意し，検証ノードはログを収集サーバに転送する．

これらの手法ではそれぞれ，利点と欠点が存在する．それぞれの検証サーバにログを保存する場合，通信路に検証とは関係ないパケットが流れなくなる．一方で，仮想計算機を用いる場合，同時に検証ノードがディスクに対して書き込みを行うことになるため，ディスクの I/O が飽和してしまう．ログ収集サーバを用意する場合は通信路に検証に関係ないパケットが流れてしまう一方，ログ収集サーバに十分なハードウェアを用意すれば検証ノードに保存するより多くのログを処理することができる．さらに，すべての計算機から検証終了後ログを収集するコストを省くことができる．本デモでは，ログ収集サーバを用意する手法を取ったが，ネットワークの帯域を埋めるような通信を伴う検証や検証ノードの間のネットワーク機器の性能の調査がある場合は検証ノードにログを保存する方法を撮るべきである．

7.3 複数の大規模検証施設の資源利用時の資源制御

それぞれの大規模検証施設では，PC への接続方法に対してそれぞれの施設で異なる方法を採用している．例えば，StarBED では，PC サーバが有する遠隔接続ソフトウェア (Dell の iDrac や Cisco の CIMC など) での接続を許可している．一方，Emulab や Deterlab では，遠隔接続ソフトウェアでの接続は許可されておらず，SSH によるリモート接続のみが許可されている．そのため，利用者はそれぞれの大規模検証施設が求める方法により自身が設定した資源に対して接続しなければならない．また，多くの大規模検証施設で許可されている SSH による接続でも，現状は資源を抽象的に表現する手法がなく，それぞれの資源がどの施設のどの PC なのか利用者で把握して接続しなければならない．大規模な検証環境の場合，すべての検証ノードがどこにあるのか正確に把握するのは困難であるため，検証者が認識しやすい抽象的な名前での接続方法や，一覧性に優れた描画による検証ノードの把握のための機能が必要であると考えられる．

ここでは，我々が行ったデモで得られた知見について述べる [76]．このデモでは，我々は検証ノードに対して以下の 2 つの制御方法を試みている．

- すべての検証ノードが起動したことを確認した後，それぞれのノードに遠隔で Quagga の起動のコマンドを発行する．
- 検証ノードの起動時に Quagga が起動するように設定を投入する．

これらの手法ではそれぞれ，利点と欠点が存在する．すべての検証ノード起動後に遠隔でコマンドを発行する方法では，検証ノードがすべて起動したことを確認した後にコマンドを実行するため，障害発生時に検証対象のソフトウェアの問題なのか，構築した環境の問題なのかの切り分けを容易に行うことができる．一方，10000 台の検証ノードすべてにコマンドを発行し終わるには時間を有する．さらに，仮想計算機による多重化を行っているため，同時に 1 台の計算機で稼働している仮想計算機にコマンドを発行してしまうと，オペレーティングシステム起動時の処理が同時に起こってしまい，CPU が高負荷状態になってしまい検証環境の開始に多くの時間がかかってしまう．

検証ノードの起動時に Quagga が起動するように設定を投入する場合は，検証ノード起動時に検証が開始されるため検証者にとって環境の監視および観測の手間を減らすことができる．一方，OS の起動時に動作させることで障害発生時に問題の切り分けが検証対象のソフトウェアなのか，検証環境なのか困難になる．

本検証から得られた知見として，動作検証などでは問題の切り分けが容易なすべての検証ノードが起動したことを確認した後遠隔コマンドの発行による検証の実施を行い，性能検証などで検証を繰り返すような場合は検証ノードの起動時にソフトウェアを起動する手法が適していると考えられる．

7.4 他の用途への応用の可能性

本研究は，固定数の計算機資源に対して仮想計算機を用いて計算機資源の量の制限を緩和した検証環境の構築手法と，大規模検証環境の連携による計算機資源を増加させた検証環境の構築の大きく 2 つの手法について提案を行った．本節では，これらの技術の他の用途への応用の可能性について述べる．

7.4.1 クラウドコンピューティング

クラウドコンピューティングにおいては，Google Cloud Platform や Amazon AWS, さくらクラウドなどがある．クラウドコンピューティングは，計算機のどの部分を貸し出すかにより Infrastructure as a Service(IaaS), Platform as a Service(PaaS) と Software as a Service(SaaS) に別れる．この中で IaaS では計算機資源や計算機資源同士を接続するネットワークを利用者に貸し出す形態であり，IaaS の貸し出し形式は大規模検証施設の資源の貸し出し方に近い．IaaS の中には計算機資源が仮想計算機として利用者に貸し出される場合もあるが，一方，PC サーバ本体を貸し出す場合もある．事業者に取っては，利用者による資源の設定時もしくは観測の結果から，より効率よく仮想計算機を配置することで手持ちの資源に対してより多くの利用者に資源を貸し出すことが可能になると考えられる．一方利用者の観点からは，本研究の成果では，検証環境に展開する前に検証環境を構築するためにどの程度のスペックを持った計算機を何台用意すればよいか事前に計算可能になる．その為，クラウドコンピューティングを借りるさいに，検証環境構築に必要最小限な計算機資源を借りればよく，初期のクラウドコンピューティングの導入にかかるコストを減少させることができる．

7.4.2 制御システムを有する施設での検証への応用

IoT という，さまざまな機器がネットワークに繋がるようになった昨今，検証においても通常の PC サーバだけでは実現できない場合が多くなってきた．これらの機器は，これまでの大規模検証施設で主に採用されてきた x86 や x64 などのアーキテクチャのプロセッサとは異なるアーキテクチャのプロセッサで動作することが想定されており，従来の大規模検証施設の資源では検証を行うことができなくなってきた．さらに，これらの機器の一部は演算結果をもとに接続されたアクチュエータの操作が行われる．このようなものも，従来の大規模検証施設の資源では検証ができない一因となっている．

重要インフラや工場などで使用される機器も近年，ネットワークを使用しお互いの機器の情報のやり取りや監視のための情報の収集が行われている．これらの分野で使用される機器も一部や従来の大規模検証施設が有する資源と同様のアー

キテクチャを持つコンピュータが使われている一方，多くの機器は IoT の機器と同様に従来の大規模検証施設が有する資源では検証ができない．このため，これら重要インフラや工場で使われている機器を検証するための検証施設や設備が近年，構築されてきた [78–81].

これらの施設では，主に制御に関する機器の検証に重点をおいているため，重要インフラや工場の制御で使用される機器の数や種類は豊富である一方，一部ではあるが使用されている PC で提供されているサービスに関しては検証を行うに十分な資源を有してない場合が多い．本研究では，検証規模の拡大を目的として大規模検証施設の連携の手法について提案を行った．本研究で扱った連携は資源の設定と資源の制御および資源の接続である．しかし，資源の設定と制御は主に PC サーバとネットワークスイッチを対象として行っており，これに制御機器を対象とした資源の設定と制御手法を取り入れることで，重要インフラや工場を対象とした検証施設と大規模検証施設を連携した重要インフラや工場の制御システム，業務システムを組み合わせた検証環境の構築を行うことが可能となる．

8. 結論

本研究では，情報科学のネットワーク分野における大規模なプロトコルやソフトウェアの検証のための手法についてモデル化を行うことで検証を行うシミュレーションと実際のソフトウェアや機器を用いて検証を行うエミュレーションに大別思で整理を行った．その上で，本研究ではエミュレーションにて大規模な検証を行う部分に着目し，その上でより大規模な検証環境を構築する手法について第一部にて検証環境のスケールアップの手法として，固定の資源に対して効率よく検証に用いるネットワークトポロジを大規模検証施設の資源上に構築する手法について述べた．第二部では，スケールアウトの手法として，複数の大規模検証施設を跨いだ検証環境構築手法として，要求事項を資源の増加のための手法と機能の均一化のための手法の 2 つにわけ，それぞれについて仕組みの提案を行った．

8.1 本研究の成果

近年の技術の高度化やインターネットの広帯域化により実現した PC の低価格化，クラウドコンピューティングの登場などにより研究者にとって大規模な検証環境構築のために必要なコストは以前よりは小さくなっている．その一方で，ネットワークに関するプロトコルやソフトウェアの研究や開発を行う者にとって，大規模な検証環境構築のために必要なコストの低下以上に爆発的に増大するインターネットの規模の検証環境を構築することはまだ困難である．また，以前に比べて企業などのネットワークや工場，センサーデバイス等の展開環境もインターネットほどではないが増加の一途を辿っている．この場合に，検証時により実展開環境に近い規模の検証環境での検証を検証者は求める一方，現在の小さくなったコストでも検証ごとに環境を揃えることは容易ではない．

本研究ではこうした問題を解決するべく，実際のソフトウェアや機器を用いて検証を行うエミュレーションでの検証時に少ない資源の中に検証環境を構築する手法と複数の大規模検証環境にまたがった検証環境を構築する手法について提案を行った．少ない計算機資源の中に検証環境を構築する手法では，検証に用いるノードとして，仮想計算機技術による 1 つの資源に複数の検証ノードを多重化す

る手法を用い，かつ予め検証に用いるソフトウェアが要求するメモリ量を静的解析と動態解析の 2 種により事前推定を行い，それぞれの仮想計算機に割り当てるべきメモリ量の決定するアルゴリズムの提案を行った．さらに，割り当てた仮想計算機を複数の計算機資源に対してどのように割り当てるかを事前計算するアルゴリズムの提案も行っている．

これまでは，大規模検証環境構築のための技術として大規模検証施設の連携を用いた複数の大規模検証施設にまたがった検証環境の構築手法について述べた．しかし，従来の手法では大規模検証施設が使用している管理ソフトウェアの種類が同一であること，大規模検証施設が資源に対して設定可能なパラメータが揃っていることが条件であり，検証者にとって制限が大きな手法であった．よって，本研究ではこの制限に対して異種管理ソフトウェアで管理された大規模検証環境間で連携を行うフレームワークの提案と，大規模検証環境が有する機能を均一化する手法について提案を行った．それぞれの手法では，現状の大規模検証施設の運用の変更やソフトウェアの改変にかかるコストを発生させないために，従来の管理ソフトウェアや運用ポリシーには変更を加えずに異種ソフトウェアで管理された大規模検証環境の資源連携と大規模検証施設の機能の均一化が可能であることを示した．

検証環境の大規模化に関しては序論にて，規模に比例して検証ノードの制御，監視および観測が困難になる点，大規模な検証環境構築の手法によっては忠実さが損なわれる点について述べた．本研究では，それぞれの手法から得られた知見を述べることで今後の検証環境の大規模化に関する道筋を示した．

8.2 本研究分野の課題と展望

本研究では，エミュレーションによる検証において，大規模な検証環境を構築する手法として，スケールアップによる検証環境の大規模化とスケールアウトによる検証環境の大規模化の 2 つの方法についてそれぞれ手法の提案を行った．一方で，近年は NS Version3 のように，シミュレーションとエミュレーションを組み合わせることで検証を行う手法についても提案がなされている．さらに，近年は機器を人間が持ち，移動しながら使用することを想定したソフトウェアやセン

サーによる情報収集がなされており，ソフトウェアに入力されるパラメータは以前に比べて複雑化している．

本研究では，エミュレーションに絞って検証環境の大規模化を行う手法について提案を行ったが，NS version3 のようにエミュレーションの検証にシミュレーションによる演算を混在させる事により大規模な検証環境を構築する手法や，ソフトウェアに入力されるパラメータをシミュレーションでモデル化し，エミュレーションによる実ソフトウェアの実行結果をシミュレーションに反映させるような検証におけるシミュレーションとエミュレーションのループ構造をもつような検証を想定する場合，本研究テーマをシミュレーションとエミュレーションを混在させ大規模な検証環境を構築する手法のための拡張が必要になる．

また，大規模な検証環境を構築する際に大きな問題となる，検証環境の監視および観測については本研究ではふれていない．しかしながら，規模が大きくなるにつれて構築した検証環境が検証を行う者にとって想定したどおりのものになっているか，また，検証中にどのノードでなにがおきているのかを正確に把握することは現状では検証の規模にしたがって困難になっていく．そのため，本提案手法の機能の一部として検証環境の監視および観測を行う必要がある．

これらの機能を加えていくことで，検証を行う者にとって資源や機能の制限による検証の縮退をなくすことが可能になり，検証者にとってより理想的な検証環境の提供が可能になると確信している．

謝辞

本研究を行うにあたり，長期間に渡りご指導賜りました門林雄基教授に深く感謝いたします．本論文の審査員をお引き受け頂きました，岡田実教授，安本慶一教授および，北陸先端科学技術大学院大学篠田陽一教授には，本論文を完成させるにあたり多くの助言をいただきました．深く感謝いたします.

国立研究開発法人情報通信研究機構総合テストベッド研究開発推進センターテストベッド研究開発運用室河合栄治室長，宮地利幸副室長，三輪信介主任研究員には本論文に関して議論やご指導，ご意見を賜りまして，深く感謝いたします.現在所属しております，北陸 StarBED 技術センターの職員のみなさまのサポートのお陰で本論文を仕上げることができました.

奈良先端科学技術大学院大学には，山口研，門林研で長く所属させていただきました．研究室在籍の中でお世話になりました，櫨山寛章特任准教授 (在籍時)，大阪大学奥田剛准教授，宮本大輔特任准教授．樫原茂助教，Doudou Fall 助教，ネットワーク統合運用研究室小林和真教授には研究面でさまざまなご指導，ご意見を賜りました．深く感謝いたします.

また，とも研究活動を行った奈良先端科学技術大学院大学インターネット工学研究室およびサイバーレジリエンス構成学研究室の先輩・後輩諸氏に感謝します.

本論文は家族の援助無しには完成しませんでした．両親と義理の両親がときに厳しくときに優しく励ましてくれたことが励みになりました．妻祐佳とペットのころ丸が私の精神的な支えとなってくれたことに感謝をし，謝辞とさせていただきます.

最後に，故山口英教授に本論文をお見せすることができなかったことが残念ではありますが，ここに先生のご指導なしでは本研究をなし得ることはできなかったことを記します.

参考文献

[1] Riverbed Technology. OPNET. https://www.riverbed.com/jp/products/steelcentral/opnet.html.

[2] Srinivasan Keshav. *REAL: A network simulator*. University of California Berkeley, Calif, USA, 1988.

[3] Steven McCanne, Sally Floyd, and Kevin Fall. ns version 1-lbnl network simulator.

[4] George F Riley and Thomas R Henderson. The ns–3 network simulator. *Modeling and tools for network simulation*, pages 15–34, 2010.

[5] The Univercity of Utah. Other Emulab Testbeds.

[6] The University of Utah and Information Sciences Institute. DETER Network Security Testbed. http://www.isi.deterlab.net/index.php3.

[7] StarBED Project. StarBED – A Large Scale Network Experiment Environment. http://www.starbed.org/.

[8] The Trustees of Princeton University. PlanetLab – An open platform for developing, deploying, and accessing planetary-scale services. http://www.planet-lab.org/.

[9] UPMC Paris Universitas. OneLab – Future Internet Testbeds. http://www.onelab.eu/.

[10] Internet2. http://www.internet2.edu/.

[11] National Institute of Information and Communications Technolgoy (NICT). Next Generation Network Testbed JGN-X. http://www.jgn.nict.go.jp/english/index.html.

[12] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proc. of SOSP '03*, pages 164–177, New York, NY, USA, 2003. ACM.

[13] KVM contributors. Kernel Virtual Machine. `http://www.linux-kvm.org/`.

[14] T. Faber and J. Wroclawski. A federated experiment environment for emulab-based testbeds. In *Proc. of TridentCom 2009*, pages 1–10. IEEE, 2009.

[15] Panlab consortium. Panlab – Pan European Laboratory Infrastructure Implementation. `http://www.panlab.net/`.

[16] FIRESTATION consortium. Future Internet Research and Experimentation – FIRE. `http://www.ict-fire.eu/`.

[17] L. Peterson, S. Sevinc, S. Baker, T. Mack, R. Moran, and F. Ahmed. PlanetLab Implementation of the Slice-Based Facility Architecture, Draft Version 0.05, June 2009. `http://www.cs.princeton.edu/~llp/geniwrapper.pdf`.

[18] The Trustees of Princeton University. PlanetLab Central API Documentation. `https://www.planet-lab.org/doc/plc_api`.

[19] The network simulator – ns-2. `https://www.isi.edu/nsnam/ns/`.

[20] OMNet++. `https://omnetpp.org`.

[21] QualNet. `http://web.scalable-networks.com/qualnet-network-simulator-software`.

[22] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. Glomosim: a library for parallel simulation of large-scale wireless networks. In *ACM SIGSIM Simulation Digest*, volume 28, pages 154–161. IEEE Computer Society, 1998.

[23] TOP500 The list. `https://www.top500.org/lists/2017/11/`.

[24] 国立研究開発法人海洋研究開発機構. http://www.jamstec.go.jp/j/.

[25] 理化学研究所計算科学研究機構. http://www.aics.riken.jp/jp/k/.

[26] SETI@home. http://setiathome.ssl.berkeley.edu.

[27] The Worldwide LHC Computing Grid project. http://wlcg.web.cern.ch.

[28] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A resource management architecture for metacomputing systems. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82. Springer, 1998.

[29] Open Grid Schedure. http://gridscheduler.sourceforge.net.

[30] Son of Grid Engine project. https://arc.liv.ac.uk/trac/SGE.

[31] Ian Foster, Carl Kesselman, Craig Lee, Bob Lindell, Klara Nahrstedt, and Alain Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Quality of Service, 1999. IWQoS'99. 1999 Seventh International Workshop on*, pages 27–36. IEEE, 1999.

[32] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster, and Steven Tuecke. Condor-g: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[33] Rajkumar Buyya and Manzur Murshed. Gridsim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and computation: practice and experience*, 14(13-15):1175–1220, 2002.

[34] puppet. https://puppet.com/#langprompt.

[35] Red Hat Ansible. https://www.ansible.com.

[36] Chef. https://www.chef.io.

[37] CFEngine. https://cfengine.com.

[38] Glenn Judd and Peter Steenkiste. Repeatable and realistic wireless experimentation through physical emulation. *ACM SIGCOMM Computer Communication Review*, 34(1):63–68, 2004.

[39] Xuxian Jiang and Dongyan Xu. vbet: a vm-based emulation testbed. In *Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, pages 95–104. ACM, 2003.

[40] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[41] S. Miwa, M. Suzuki, H. Hazeyama, S. Uda, T. Miyachi, Y. Kadobayashi, and Y. Shinoda. Experiences in emulating 10K AS topology with massive VM multiplexing. In *Proc. of VISA 2009*, August 2009.

[42] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. *ACM SIGOPS Operating Systems Review*, 36(SI):271–284, 2002.

[43] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review*, 36(SI):255–270, 2002.

[44] M. Suzuki, H. Hazeyama, D. Miyamoto, S. Miwa, and Y. Kadobayashi. Expediting Experiments across Testbeds with AnyBed: A Testbed-Independent Topology Configuration System and Its Tool Set. *IEICE Trans. on Info. and Sys.*, E92-D(10):1877–1887, October 2009.

[45] T. Miyachi, K. Chinen, and Y. Shinoda. StarBED and SpringOS: Large-scale general purpose network testbed and supporting software. In *Proc. of TridentCom 2006*, page 30, 2006.

[46] The University of Utah. Emulab - Network Emulation Testbed. http://www.emulab.net/.

[47] Geoffrey Werner-Allen, Patrick Swieskowski, and Matt Welsh. Motelab: A wireless sensor network testbed. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, page 68. IEEE Press, 2005.

[48] Emre Ertin, Anish Arora, Rajiv Ramnath, Vinayak Naik, Sandip Bapat, Vinod Kulathumani, Mukundan Sridharan, Hongwei Zhang, Hui Cao, and Mikhail Nesterenko. Kansei: a testbed for sensing at scale. In *Proceedings of the 5th international conference on Information processing in sensor networks*, pages 399–406. ACM, 2006.

[49] Manjunath Doddavenkatappa, Mun Choon Chan, and Akkihebbal L Ananda. Indriya: A low-cost, 3d wireless sensor network testbed. In *TridentCom*, volume 90, pages 302–316. Springer, 2011.

[50] Vlado Handziski, Andreas Köpke, Andreas Willig, and Adam Wolisz. Twist: a scalable and reconfigurable testbed for wireless indoor experiments with sensor networks. In *Proceedings of the 2nd international workshop on Multi-hop ad hoc networks: from theory to reality*, pages 63–70. ACM, 2006.

[51] OpenStack. https://www.openstack.org.

[52] Juju. https://www.ubuntu.com/cloud/juju.

[53] docker. https://www.docker.com.

[54] L. Peterson, A. Bavier, M.E. Fiuczynski, and S. Muir. Experiences building planetlab. In *Proc. of OSDI 2006*, pages 351–366. USENIX Association, 2006.

[55] Yakov Rekhter, Tony, and Shares Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), January 2006.

[56] R. Coltun, D. Ferguson, J. Moy, and A. Lindem. Ospf for ipv6, July 2008. RFC5340.

[57] R. Atkinson and M. Fanto. Ripv2 cryptographic authentication, February 2007. RFC4822.

[58] C. Hopps. Routing ipv6 with is-is, October 2008. RFC5308.

[59] Quagga Software Routing Suite. http://www.quagga.net/.

[60] CZ.NIC z.s.p.o. The bird internet routing daemon. http://bird.network.cz/.

[61] Advanced Industorial Science and Technology. VMKNOPPIX: Collection of Virtual Machine. http://unit.aist.go.jp/itri/knoppix/vmknoppix/.

[62] Pascal Schmidt. ttylinux Homepage. http://www.minimalinux.org/ttylinux/showpage.php?pid=1.

[63] CAIDA Project. http://www.caida.org/.

[64] Xenofontas Dimitropoulos, Dmitri Krioukov, Marina Fomenkov, Bradley Huffaker, Young Hyun, kc claffy, and George Riley. AS relationships: Inference and validation. *ACM SIGCOMM CCR*, 37(1):29–40, 2007.

[65] CAIDA Project. The CAIDA AS Relationships Dataset. http://www.caida.org/data/active/as-relationships/.

[66] M. Suzuki. AnyBed. http://sourceforge.net/projects/anybed/.

[67] H. Hazeyama, M. Suzuki, S. Miwa, D. Miyamoto, and Y Kadobayashi. Outfitting an Inter-AS Topology to a Network Emulation TestBed for Realistic Performance Tests of DDoS Countermeasures. In *Proc. of CSET'08*, August 2008.

[68] S. Schwab, B. Wilson, C. Ko, and A. Hussain. SEER: a security experimentation EnviRonment for DETER. In *Proc. of DETER 2007*, Aug. 2007.

[69] Razvan Beuran, Lan Tien Nguyen, Toshiyuki Miyachi, Junya Nakata, Ken-ichi Chinen, Yasuo Tan, and Yoichi Shinoda. Qomb: A wireless network emulation testbed. In *Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE*, pages 1–6. IEEE, 2009.

[70] Ken-ichi Chinen, Toshiyuki Miyachi, and Yoichi Shinoda. A rendezvous in network experiment-case study of kuroyuri. In *Testbeds and Research Infrastructures for the Development of Networks and Communities, 2006. TRIDENTCOM 2006. 2nd International Conference on*, pages 8–pp. IEEE, 2006.

[71] Kevin Fall. Network emulation in the vint/ns simulator. In *IEEE International Symposium on Computers and Communications*, pages 244–250. IEEE, 1999.

[72] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale Virtualization in the Emulab Network Testbed. In *Proc. of USENIX ATC 2008*, June 2008.

[73] 明石邦夫, 井上朋也, and 篠田陽一. インターネットの特性を考慮した実験ネットワーク構築フレームワークの設計と実装 (情報ネットワーク). 電子情報通信学会技術研究報告= *IEICE technical report:* 信学技報, 113(140):7–12, 2013.

[74] netem. `https://wiki.linuxfoundation.org/networking/netem`.

[75] Fabien Hermenier and Robert Ricci. How to build a better testbed: Lessons from a decade of network experiments on emulab. In *Proc. of Tridentcom 2012*, 2012.

[76] H. Hazeyama, S. Ohta, S. Miwa, T. Miyachi, M. Suzuki, S. Uda, K. Chinen, M. Enomoto, Y. Kadobayashi, and Y. Shinoda. Emulating over 10K AS topology with massive VM multiplexing. In *ACM SIGCOMM 2009, Demo Session*, August 2009.

[77] 国立研究開発法人情報通信研究機構. Jose(japan-wide orchestrated smart/sensor environment). `https://www.nict.go.jp/nrh/nwgn/jose.html`.

[78] 技術研究組合制御システム・セキュリティセンター. `http://www.css-center.or.jp`.

[79] Masatoshi Enomoto, Shu Hosokawa, and Kazutaka Matsuzaki. A Testbed for Enhancing Control System Security. In *Proceeding of the World Engineering Conference and Convention 2015*, November 2015.

[80] Idaho national laboratory. `http://www.inl.gov/`.

[81] European network for cyber security. `https://www.encs.eu`.

付録

A. 業績リスト

論文誌

1–1. 榎本真俊, 櫨山寛章, 奥田剛, 山口英. “仮想計算機による AS エミュレーション環境構築のための回帰分析を用いた消費メモリ量推定手法の提案”, 日本ソフトウェア科学会「ネットワーク技術」特集, 2015 年 8 月.

1–2. Shingo Yasuda, Kunio Akashi, Masatoshi Enomoto, Shinsuke Miwa, and Yoichi Shinoda. “Technical requirements of experiment support software for huge-scale network environment”, *International Journal of Computers and Communications*, No. id.123, 2012.

国際会議（口頭発表）

2–1. Masatoshi Enomoto, Tomoya Inoue, Kunio Akashi, Shisuke Miwa, Toshiyuki Miyachi, Daisuke Miyamoto. “Ditto subsystem : A case study for building an experiment environment in StarBED using TopDL”, *In proceedings of European Conference on Electrical Engineering & Computer Science*, November 2017.

2–2. Masatoshi Enomoto, Shu Hosokawa, and Kazutaka Matsuzaki. “A Testbed for Enhancing Control System Security”, *In Proceeding of the World Engineering Conference and Convention 2015*, November 2015.

2–3 Tomonori Ikuse, Shinya Kanda, Masato Jingu, Kyohei Moriyama, Masatoshi Enomoto, Hiroaki Hazeyama, and Takeshi Okuda. “FU-JIN : A Cloud Computing Environment Visualization System for Specifying Points of Failure.”, *In 2nd IEEE International Conference on Cloud Computing Technology and Science*,December 2010.

2–4 Hosokawa Shu, Enomoto Masatoshi, Matsumoto Kohei and Takahashi Makoto. "A prototype of cyber incident diagnosis mechanism for cyber attacks early recognition support system", *In Proceeding of Asian Control Conference 2015*, 2015.

2–5 Hiroaki Hazeyama, Satoshi Ohta, Shinsuke Miwa, Toshiyuki Miyachi, Mio Suzuki, Satoshi Uda, Ken ichi Chinen, Masatoshi Enomoto, Youki Kadobayashi, and Youichi Shinoda. "Emulating over 10K AS topology with massive VM multiplexing", *In ACM SIGCOMM 2009, Demo Session*, August 2009.

国内ワークショップ発表論文（査読なし）

3–1. 榎本真俊, 櫨山寛章, 小林和真, 山口英. "テストベッド連携環境を用いたネットワークセキュリティ実験の試行. ", 電子情報通信学会 技術研究報告, vol. 114, no. 489, ICSS2014-67, pp. 25-29, 2015 年 3 月.

3–2. 尾花悦正, 榎本真俊, 櫨山寛章, 門林雄基. "協調型侵入検知システム研究における実装負荷軽減のための実装テンプレートの提案", 電子情報通信学会 技術研究報告, volf. 112, no. 489, IA2012-90, pp. 43-48, 2013 年 3 月.

3–3. 榎本真俊, 櫨山寛章, 三輪信介, 奥田剛, 山口英. "異種テストベッド間のテストベッド連携手法の提案", 第 *20* 回 インターネットと運用技術研究発表会 ,Vol.2013-IOT-20 No.26, 2013 年 3 月.

3–4. 齋藤利文, 榎本真俊, 櫨山寛章, 門林雄基. "LISP における二段階マップテーブルを用いた DDoS 攻撃緩和方式の実装と評価", 電子情報通信学会 技術研究報告, vol. 112, no. 489, IA2012-89, pp. 37-42, 2013 年 3 月.

3–5. 齋藤利文, 榎本真俊, 櫨山寛章, 門林雄基. "囮サーバーで攻撃を局所に封じ込めることによる DDoS 攻撃緩和手法の提案.", 電子情報通信学会 技術研究報告, vol. 112, no. 302, IA2012-58, pp. 35-40, 2012 年 11 月.

3–6. 尾花悦正, 榎本真俊, 櫨山寛章, 門林雄基. “ピアツーピアによる協調型侵入検知システムの一検討”, *電子情報通信学会 技術研究報告*, vol. 112, no. 302, IA2012-56, pp. 23-28, 2012 年 11 月.

3–7. 幾世知範, 榎本真俊, 櫨山寛章, 門林雄基, 山口英. “動的依存性グラフを用いた障害原因解析の計算コスト削減に関する一考察”, *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会 (2011-OS-119)*, 2011 年 11 月.

3–8. 神田慎也, 榎本真俊, 櫨山寛章, 門林雄基, 山口英. “実験環境でのプロファイル作成のためのデータ収集手法の基礎調査”, *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会 (2011-OS-119)*, 2011 年 11 月.

3–9. 榎本真俊, 櫨山寛章, 三輪信介, 門林雄基, 山口英. “BGP ネットワークエミュレーションにおける仮想計算機を用いた大規模実験時の効率的なメモリ割り当て手法の提案”, *情報処理学会 マルチメディア，分散，協調とモバイル (DICOMO2010) シンポジウム*, pp. 296-303, 2010 年 7 月.

3–10. 太田悟史, 宮地利幸, 三輪信介, 櫨山寛章, 榎本真俊, 宮本大輔. “ライブトラフィックを用いた模倣インターネットの特性に関する一考察”, *情報処理学会 マルチメディア，分散，協調とモバイル (DICOMO2010) シンポジウム*, pp. 314-323, 2010 年 7 月.

3–11. 鈴木未央, 櫨山寛章, 榎本真俊, 三輪信介, 門林雄基. “ネットワークエミュレーションテストベッドを用いた実 OSPF トポロジ模倣システム.”, *インターネットコンファレンス 2009*, pp. 15-23, 2009 年 10 月.

3–12. 鈴木未央, 櫨山寛章, 榎本真俊, 三輪信介, 門林雄基. “模倣インターネット - 実 OSPF ネットワークの模倣 -”, *インターネットコンファレンス 2009 デモンストレーション展示*, p. 121, 2009 年 10 月.

3–13. 櫨山寛章, 榎本真俊, 鈴木未央. “ネットワークエミュレーションテストベッドを用いたオーバーレイネットワークアプリケーションの実験手法に関す

る一考察", 電子情報通信学会技術研究報告 インターネットアーキテクチャ , 第 109 巻, pp. 37-42, 2009 年 9 月.

その他の業績

4–1. 櫨山寛章, 榎本真俊. "Interop Cloud Computing にて総合 4 位", 2009 年 6 月.