情報処理学会論文誌　コンピューティングシステム　Vol. 1　No. 2　144–156 (Aug. 2008)

プロセスレベルの仮想化を用いた大規模分散システムテストベッド

西川賀樹[†1]　大山恵弘[†2]　米澤明憲[†1]

近年，高速なネットワークの普及により，P2P システム等の大規模分散システムが数多く開発されるようになった．しかし，大規模分散システムは相互に通信する多数の計算機を管理する必要があるため，その開発は容易ではない．また，多数の計算機が正常に協調動作することを検証するためのテスト環境構築に大きな手間がかかるという問題がある．本研究では，プロセスレベルの仮想化を用いて 1 台または数台の計算機上に数百から数千の仮想環境を構築し，大規模分散システムのテストを可能とするミドルウェアを提案する．そのミドルウェアは，ネットワークエミュレーション機能，テストの自動実行，デバッグ支援機構，仮想環境の状況を提示する GUI，テスト結果解析機構等の様々な機構により分散システムの開発を支援する．我々は 20 台によるクラスタ上に 6,000 の仮想環境を生成し，その上で Gtk-Gnutella 等の様々な P2P システムを問題なく実行できることを確認した．

Large-scale Distributed System Test Bed by Using Process-level Virtualization

Yoshiki Nishikawa,[†1] Yoshihiro Oyama[†2] and Akinori Yonezawa[†1]

High-speed networks are getting more popular in recent years, and many large-scale distributed systems are developed. However, developing them is not straightforward because they must manage a great number of machines that frequently communicate with each other. In particular, it requires much effort to build a testing environment (test bed) for examining whether many machines cooperate correctly. In this work, we propose middleware using process-level virtualization that enables to test large-scale distributed systems by creating hundreds or thousands of virtual environments on one or several computers. The middleware supports the development of distributed systems with various components including network emulation facility, automatic execution of tests, debugging support, a GUI monitor that shows network states, and a component for analyzing test results. We generated 6,000 virtual environments on a 20-node cluster and confirmed that various P2P systems including Gtk-Gnutella worked well on the virtual environments.

1. はじめに

近年，ネットワークの高速化によって P2P システム等の分散システムが数多く開発されるようになった．従来のクライアント・サーバモデルでは 1 対 1 通信でデータのやりとりが行われる．しかし，分散システムではつねに数多くの計算機が相互に通信することで成り立っており，このような分散システムの開発は容易ではない．なぜなら，多数の計算機が相互に通信し正常に協調動作をすることを検証しなければならないからである．分散システム開発の従来手法としては，実際に多数の計算機を用意して行う手法（PlanetLab[1)]，Netbed[2)] 等）が存在するが，この手法はコストが大きく，手軽に実現できないという問題があった．

本研究では，P2P・ネットゲーム等の大規模分散システムのテストを，プロセスレベルの仮想化を用いて少数の計算機上で行うことを可能とし，様々な機構により開発を支援するミドルウェアを提案する．本研究の提案するミドルウェアでは，1 台または数台の計算機上に数百〜数千の仮想環境を構築でき，分散システムの開発・テスト環境を低コストで実現する．

分散システムのテスト環境を構築する 1 つの有力な手法としては仮想マシンモニタがある[3),4)]．ただしその手法において問題となるのが，仮想化のオーバヘッドのために実現できる仮想環境の数が大きく制限される点である．この問題に対して本研究では，多くの分散システムの開発・テストに必要な OS 資源のみを仮想化することで，より多くの仮想環境を高速に動作させた．実際に，20 台によるクラスタ上で 6,000 の仮想環境を生成し，その上で P2P ファイル共有システムを問題なく実行することができた．

提案するミドルウェアは，ネットワークエミュレーション機能，テストの自動実行，デバッグ支援機構，仮想環境の状況を示す GUI，テスト結果解析機構等の様々な機構により分散システムの開発を支援する．本ミドルウェアは，ユーザから与えられたノードとネットワークに関する仕様，テストのシナリオ（実行手順）を基にテストに合わせて仮想分散実行

†1 東京大学大学院情報理工学系研究科コンピュータ科学専攻
Department of Computer Science, Graduate School of Information Science and Technology, The University of Tokyo
†2 電気通信大学電気通信学部情報工学科
Department of Computer Science, Faculty of Electro-Communications, The University of Electro-Communications

環境を構築し，テストを自動的に実行する．またテスト中は，デバッグ支援機構により仮想環境上で動作する多数のアプリケーションをまとめて管理・制御し，デバッグを行うことができる．テストの結果は，グラフやアニメーション，テキストを用いて表示し，通常困難な分散実行における各ノード・システム全体の状況の把握をより分かりやすく行えるようにする．これらの機構により，開発者がシステムの動作の正当性，パラメータの変化による実行の変化を検証することを可能にする．

本ミドルウェアは Linux 上に実装されており，x86・x86-64 アーキテクチャにおいて動作する．本ミドルウェアは約 10,300 行の C 言語のコードとグラフ・アニメーション表示機能のための約 700 行の Java のコードで記述されている．仮想環境はプロセストレース用 API である ptrace システムコールを用いて構築する．具体的には，アプリケーションが発行するシステムコールを捕捉して，引数に与えられたファイルパスやネットワークアドレスを書き換えること等により，ファイルシステムとネットワークへの操作を仮想化する．そのミドルウェアは，上記の仮想化によって行える開発作業を対象とし，より低い層（デバイスの仮想化等）を必要とするものは対象としない．

本ミドルウェアはユーザレベルで実装されているため，OS やアプリケーションの修正，管理者権限は不要である．このため，ユーザが共用の計算機等でも容易に使用することが可能であり，実機の用意という面でもユーザの負担を軽減し，手軽に頻繁に分散システムのテストを行うことができる．

既存研究として数千ノード規模のシミュレーションを行うものはあるが，本研究のように既存のアプリケーションを動作させ検証が可能であるものは我々の知る限り存在しない．

2. 関 連 研 究

仮想化技術による計算機の多重化については，様々な手法がある．まず，Xen [3]，VMware [4] といった仮想マシンモニタを用いると，1 台の計算機上で同時に複数の OS を動作させることができる．仮想マシンモニタは最も汎用性があるが，ハードウェアをエミュレートするオーバヘッドのため，実現できる仮想環境の数が大きく制限される．また，UML [5] 等の仮想 OS もすべての OS 資源の仮想化を行うため，オーバヘッドが大きく実現できる仮想環境の数に限界がある．jail や chroot といったカーネルによる仮想環境構築機能を用いた場合は非常に軽量ではあるが，アプリケーションの実行に必要なファイルを仮想環境内に準備するのに手間がかかる．またネットワークの仮想化が不十分であり，これらの仮想環境構築機能だけでは大規模な仮想分散実行環境を簡便に構築することはできない．単一カーネルにより仮想環境を構築する Linux-Vserver [6]，Zap [7]，Solaris Containers に関しても jail や chroot と同様の問題があり，また Linux-Vserver ではカーネルの修正，Zap においてはカーネルモジュールのロードが必要となる．本研究と同様に，プロセスレベルの仮想化を用いて仮想環境を構築するシステムとして，SoftwarePot [8] がある．SoftwarePot は，ネットワークの仮想化が不十分であり仮想分散実行環境を簡便に構築することができない．

これら既存の仮想化技術を用いて仮想分散実行環境を構築する場合に比べ，本システムを用いた場合の利点は複数実計算機上のすべての仮想環境に独自の仮想 IP アドレスを，本システムの仮想化のみで割り当てることが可能な点である．既存の仮想化技術を用いた場合は，異なる実計算機上で動作する仮想環境間では実 IP アドレスを用いなければ通信できない．しかし，本システムでは割り当てられた仮想 IP アドレスにより，複数実計算機上のすべての仮想環境と通信することが可能である．また本研究で行う仮想化は，分散システムの開発・テストに必要な OS 資源に対してのみ行うため，汎用性という面では仮想マシンモニタや仮想 OS に比べ劣るが，より多くの仮想環境を高速に動作させることができる．

分散システムのテスト環境については，ns-2 [9]，GloMoSim [10] 等のネットワークシミュレータや PlanetLab [1]，Netbed [2]，StarBED [11] 等のテストベッドが利用されている．ネットワークシミュレータは，与えられた構築アルゴリズムやパラメータにより実現されるネットワークの特性・性能をシミュレーションによって高い精度で予測するのに役立つ．しかし，多くのシミュレータでは，シミュレーションにはテスト専用のコードを記述する必要があり，別途実環境で動作するソフトウェアを実装しなければならない．そのため，実環境で動作するソフトウェアがシミュレーションと同じ挙動をする保証はない．Overlay Weaver [12] に関しては，数千ノード規模においてエミュレータ上でアプリケーションを実行可能であり，そのアプリケーションをそのまま実環境で動作させることもできる．しかし，テストするアプリケーションは専用の API を使用して記述しなければならない．また，Overlay Weaver は各アプリケーションに独自の仮想環境を提供するものではない．そのため，ネットワークシミュレータと Overlay Weaver の双方とも既存のアプリケーションをそのまま動作させ検証を行うものではない．テストベッドは誰でも利用することが可能とされているが，利用のためには手続きが必要である場合や，計算リソースの提供が必要等の条件がある場合があり，手軽には利用できないことがある．

本システムはテストベッドでの実験の前段階として，アプリケーションの実装時に開発者が使用可能な限られた実計算機で簡単にテストを行えることを第 1 の目的としている．そのために設定・実行手順の記述を簡潔に行えるようにし，様々な機構によりテスト・開発をサ

ポートする．それにより，テストベッドで行われる正確な実行性能の測定や検証ではなく，プロトタイプ開発段階のアプリケーションの動作検証をより簡単に行えるようにしている．極端な例では，ネットワークに接続されていない 1 台の計算機でも数百規模の仮想分散環境によりテストを行うことができる．また，本システムは管理者権限や OS の修正が必要ないため，既存のテストベッド上でも使用が可能である．それにより，テストベッドでの実験の際に割り当てられたノード上で本システムを使用することで，より大規模なテストを行うこともできる．

3. 提案するミドルウェア

本ミドルウェアの構成図を**図 1** に示す．マスタ計算機において，仮想分散実行環境の入出力を管理するシェル，仮想環境の状況を提示する GUI，テストを自動実行するシナリオマスタ，テスト結果解析機構が動作する．各仮想環境の実行ログは各計算機ごとに記録され，テスト終了後にテスト結果解析機構により，ログの収集・解析が行われる．またデバッグ支援機構についても各計算機ごとに動作し，その操作は入出力管理シェルを通して行われる．本章では，仮想化やこれらの各機構についての詳細を述べる．

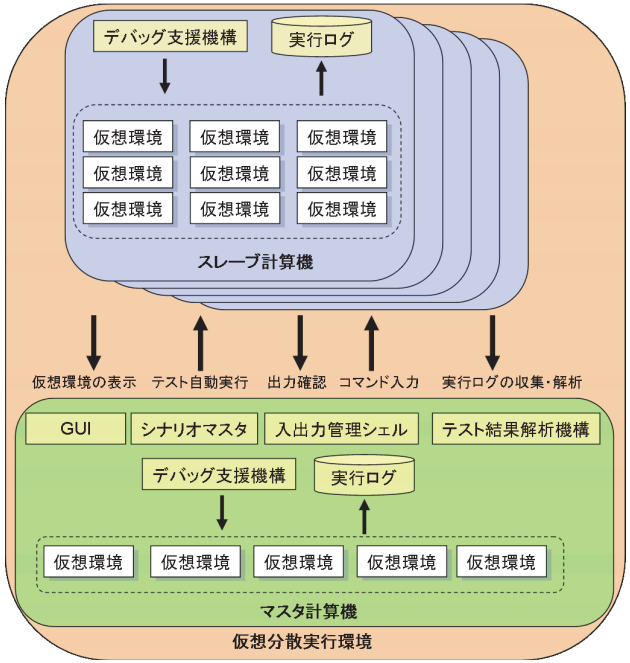

図 1　本ミドルウェアの構成図
Fig. 1　Composition of our middleware.

3.1 仮　想　化

本ミドルウェアでは，ファイルとネットワークの仮想化を行い，実環境から隔離された仮想環境の中でプロセスを実行可能にする．ファイルに関しては，chroot 環境と同様に，実環境のものとは異なるファイル空間を，仮想環境中のプロセスに対して見せる．ネットワークに関しては，各仮想環境に仮想的なアドレスを割り当てる．本ミドルウェアは ptrace システムコールを用いてユーザレベルで実装されている．

3.1.1 実装の詳細

まず本ミドルウェアのスレッドがアプリケーションを実行し ptrace により監視する．以後，この監視を行う本ミドルウェアのスレッドを監視スレッドと呼ぶ．また，1 つのアプリケーションが生成するすべてのプロセスに対して，1 つの監視スレッドが監視を行う．アプリケーションプロセスがシステムコールを発行すると，監視スレッドはそのシステムコールをフックし，アプリケーションプロセスをシステムコール実行前に停止させる．その後，各システムコールに対して実行前後に仮想化処理を行う．処理を行うシステムコールは主に以下の 4 種類である．

- ファイルパスを引数に持つ
  （open，stat，link，rename 等）
- IP アドレス・ポート番号を引数に持つ
  （connect，accept，bind，sendto，recvfrom 等）
- プロセスの生成・新たなプログラムの実行
  （clone，execve，fork）
- ソケット記述子に対して生成・操作を行う
  （socket，read，write，send，recv 等）

ファイルシステムの仮想化は，まず各仮想ノードに作られるファイル群の実体を実ファイルシステム上に置く．監視スレッドは open・stat 等のファイルパスを引数に持つシステムコールをフックし実行前に仮想パスから実パスに，実行後に実パスから仮想パスに変換しアプリケーションが参照できるメモリ空間に書き込む．このメモリ空間は監視スレッドがアプリケーションの実行時に環境変数領域として確保する．このパス変換により，アプリケー

ションは他のアプリケーションのファイル空間を意識せず，仮想的に独自のファイル空間で実行される．

ネットワークの仮想化に関しても同様に，各仮想ノードに仮想 IP アドレス・ポート番号を設定し，実 IP アドレス・ポート番号に対応付ける．監視スレッドは connect・accept・bind 等の通信システムコールの引数を仮想的な IP アドレス・ポート番号から実際の IP アドレス・ポート番号に，またはその逆に書き換えることにより実現する．また Unix ドメイン・ソケットの場合はファイルパスの変換を行う．本ミドルウェアが実際に使用する IP アドレスは 1 計算機につき 1 つであり，複数の仮想 IP アドレスが 1 つの実 IP アドレスに対応付けられる．そのため，仮想 IP アドレスとポート番号の組の各々を異なる実ポート番号にマッピングすることにより，仮想環境における IP アドレスとポート番号の組が実環境における組に 1 対 1 対応するようにしている．

本ミドルウェアでは，複数の実計算機を用いた場合における仮想 IP アドレス・ポート番号と実 IP アドレス・ポート番号のマッピング情報の管理は 2 通りの方法で行う．

第 1 の方法は，各仮想環境が接続を受け入れるために使用する仮想ポート番号と実ポート番号のマッピングをユーザが事前に設定を記述し定義する手法である．それにより，各仮想環境上のアプリケーションは他の仮想環境上で動作するアプリケーションに対して，接続要求を行うことができる．また各仮想環境ごとに使用する実ポート番号の範囲を本ミドルウェアがあらかじめ定義しておき，受け入れ側は使用された実ポート番号からどの仮想環境上で動作するアプリケーションからの接続要求であるか判断することができる．しかし，アプリケーションが bind を行っていないソケット記述子を用いて通信を開始した場合，任意の実ポート番号が使用されるため，監視スレッドはその実ポート番号から仮想 IP アドレスを判断できない．そのため，アプリケーションが bind を行わずに通信を開始しようとした場合，監視スレッドはアプリケーションに意識させずに強制的に bind システムコールを発行させる．それにより，各監視スレッドは各アプリケーションが使用する実ポート番号を管理し，各アプリケーションが動作する仮想環境の仮想 IP アドレスを判断することができる．アプリケーションに bind システムコールを発行させる手法については，まずアプリケーションが bind を行わずに connect システムコールを発行した場合，監視スレッドは各レジスタの値を保存し，適切に引数を書き換え bind システムコールに変換する．bind システムコール実行後，監視スレッドはアプリケーションプロセスが次に実行する命令を保存し，システムコールのソフトウェア割込みを発生させる命令に書き換える．その後，監視スレッドはアプリケーションプロセスの各レジスタ値を復元し実行を再開させる．これにより本来呼ばれるべき connect システムコールが実行される．connect システムコール実行後，保存されている本来の命令に再度書き換えを行う．この方法では，計算機間でマッピング情報の更新を行う必要がなく，高速に仮想ネットワークを構築することができる．

第 2 の方法は，すべての仮想環境が使用する仮想 IP アドレス・ポート番号と実 IP アドレス・ポート番号に対してマッピング情報を作成し，計算機間で同期的にマッピングテーブルを更新する手法である．ユーザが使用する仮想ポート番号と実ポート番号のマッピングを事前に定義できない場合，また複数計算機上で NAT 等のネットワークエミュレーション機能を使用する場合，正確なマッピング情報が必要となるため，この方法で仮想ネットワークを構築する．この方法では，ユーザは制限なしに仮想ポート番号を使用できる．ただし，マッピングテーブルを同期させるための通信を頻繁に行うため，第 1 の方法に比べ性能は低下する．

本ミドルウェアは起動時に，設定において仮想ポート・実ポートの対応を記述されているか，NAT の定義が行われているかを確認し，どちらの方法で仮想ネットワークを構築するか決定する．

プロセスの生成・新たなプログラムの実行を行うシステムコールについては，仮想環境上で動作するアプリケーションが生成するすべてのプロセス・スレッドに対して仮想化を行うための処理を行う．

ソケット記述子に対して生成・操作を行うシステムコールに関しては，前述の仮想ネットワーク構築にともない，各ソケット記述子の管理のため，また仮想環境の状況を提示する GUI やテスト結果解析機構において必要となるログを記録するための処理を行う．

3.1.2 動作確認

我々は実際に Gtk-Gnutella[13]，Mutella[14]，DB Hub[15]，microdc2[16] 等の P2P アプリケーションを使用してテストを行い，すべてのアプリケーションが仮想環境上で正常に動作することを確認した．仮想環境上で動作する Gtk-Gnutella と Mutella をそれぞれ**図 2**，**図 3** に示す．図中の IP アドレスは本ミドルウェアが割り当てた仮想 IP アドレスであり，仮想ネットワークが構築されていることが分かる．本ミドルウェアは非常に軽量であり，200 の仮想環境を構築しても本ミドルウェアが使用するメモリは 10 MB 程度である．使用するアプリケーションにもよるが，平均的スペックの PC 上で 200〜300 の仮想環境を動作させることができる．実際に，20 台によるクラスタ上に 6,000 の仮想環境を構築し，各仮想環境上で DB Hub，microdc2 が正常に動作することを確認した．クラスタの各計算ノードは CPU が x86-64 2.2 GHz のデュアルコアであり，主記憶が 2 GB である．

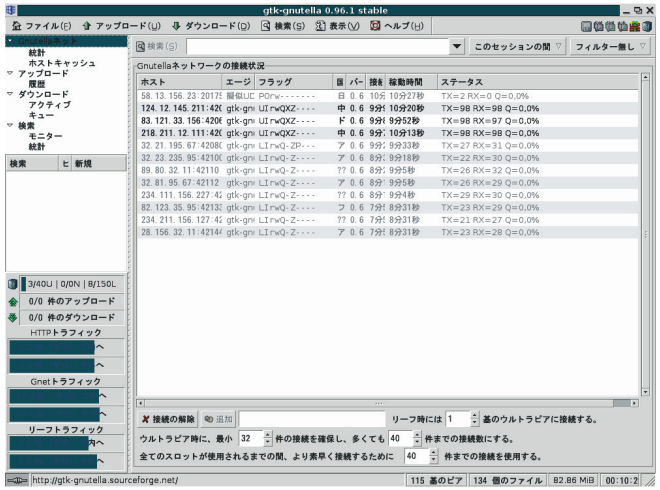

図 2　仮想環境上で動作する Gtk-Gnutella
Fig. 2　Gtk-Gnutella on virtual environment.

3.2　仮想環境の設定

3.2.1　仮想ネットワーク環境

本ミドルウェアでは，ユーザは仮想ネットワーク環境を自由に指定できる．具体的には，以下のような仮想環境の設定が行える．

- ネットワークトポロジと各ノードにおいて動作するアプリケーション
- NAT のエミュレーション
- ノード間の通信遅延
- ノードの動的な参加・脱退

分散システムではノードの動的な参加・脱退が頻繁に発生したり，通信に遅延が発生したりすることを考慮する必要があるため，これらの設定が可能であることは分散システムのテスト・検証では重要である．また，分散システムのネットワーク構築における大きな問題の 1 つが NAT を介した通信である．その動作検証を可能とするため，Cone NAT，Restricted Cone NAT，Port-Restricted Cone NAT，Symmetric NAT の 4 種類の NAT のエミュレーション機能を実装している．

NAT のエミュレーションは実際の NAT と同じ処理をアプリケーション層で行うことにより実現している．NAT ごとにマッピングテーブルを保持し，NAT の内側の仮想ノードが接続要求を行う際に，外側・内側のアドレス・ポートのマッピング，通信先の外部ホストの情報をマッピングテーブルに登録する．そして，他の仮想ノードから NAT の内側の仮想ノードに対する接続要求に対して，マッピングと外部ホストの情報から接続の可否を判断する．マッピングテーブルは計算機間で同期的に更新される．

```
DSTB>info connections#node1,node7
DSTB>
DSTB>
node1_app_out:
>
DSTB>
node7_app_out:
>
DSTB>
node7_app_out:
gnutella network connections:
------------------------------
no.|          address:port | T |  rate i:o  | horizon   | eff. | uptime | c.time
 1)     199.76.123.202:0     U     10:7        21/42M      91%     3m9s     3m9s
 2)   216.126.171.221:6346   U     9:10        25/50M      91%     3m9s     3m9s
 3)     226.83.96.161:0      L     4:4          9/18M     100%    2m50s    2m50s
 4)      89.21.59.121:0      U     6:3         11/22M      67%    2m50s    2m50s
 5)     151.60.55.31:0       L     4:3          9/17.6M   100%    2m48s    2m48s
 6)     210.53.22.102:0      L     4:2          9/17.6M    66%    2m48s    2m48s
 7)    241.30.213.194:0      L     4:3          8/16M      88%    2m48s    2m48s
 8)      97.55.62.120:0      U     6:3         10/20M      64%    2m47s    2m47s
 9)     230.40.60.147:0      L     3:2          8/15.6M    53%    2m46s    2m46s
10)     122.93.151.52:0      L     3:2          8/15.6M    57%    2m46s    2m46s
11)     10.34.204.188:0      L     4:3          9/18M      88%    2m45s    2m45s
12)   113.120.190.100:0      U     5:10        10/20M      92%    2m43s    2m43s
total connections: 12  rate: [   63|53   ]/sec

>
DSTB>
node1_app_out:
gnutella network connections:
------------------------------
no.|          address:port | T |  rate i:o  | horizon   | eff. | uptime | c.time
 1)    16.182.195.41:0       U     7:8         24/48M      91%     3m9s     3m9s
 2)   205.37.148.220:0       L     3:2          9/18M      68%    2m48s    2m48s
 3)    158.33.95.113:0       L     4:3          9/17.6M    94%    2m48s    2m48s
 4)     100.57.81.65:0       L     3:3          8/16M     100%    2m47s    2m47s
 5)      95.70.61.28:0       L     3:2          8/16M      80%    2m46s    2m46s
 6)     189.76.10.56:0       L     4:3          9/17.6M    77%    2m45s    2m45s
 7)   114.203.141.93:0       U     5:5         10/20M      88%    2m43s    2m43s
 8)     98.103.89.57:0       L     4:4          9/17.6M   100%    2m43s    2m43s
 9)     60.49.16.165:0       U     5:4          9/18M      62%    2m42s    2m42s
10)   232.183.148.102:0      U     5:3         10/20M      64%    2m42s    2m42s
total connections: 10  rate: [   37|25   ]/sec

>
DSTB>
```

図 3　仮想環境上で動作する Mutella
Fig. 3　Mutella on virtual environment.

3.2.2　仮想環境の定義

ユーザが与えたい仮想分散実行環境に関する設定を表現するための記述言語を**図 4** に示す．まず `master` か `slave` を定義し，使用する計算機の計算機名と実 IP アドレスを記述する．仮想環境の状況を表す GUI や各仮想環境上のアプリケーションへの入力・出力のイン

```
/*計算機の定義*/
master or slave：計算機名 [実 IP アドレス]{
/*仮想ノードの定義*/
  ノード名 {
    type = 動作するアプリケーション
    IP = [仮想 IP アドレス:仮想ポート:実ポート]
    file = [仮想ファイルパス][実ファイルパス]
    group = 仮想ノードが所属するグループ名
    latency = ノード名 [遅延時間]
  }
}
```

図 4　設定記述言語
Fig. 4　Specifications of the description language.

```
master:compute0[176.11.15.158]{
 for(i;1;200;1){
  node@i{
   type = /usr/local/bin/mutella
   IP = [auto]
   file = [/home/.mutella/][/home/m@i/.mutella]
   file = [/home/mutella/][/home/m@i/mutella]
   group = compute0
   latency = node3[1].node150[1]
  }
 }
}
```

図 5　仮想環境の設定例
Fig. 5　Example of specification.

タフェースは master の定義がされた計算機上で動作する．次に各計算機ごとに仮想ノードを定義する．ノードの定義は type に動作するアプリケーションのパス，IP に仮想ノードが使用する仮想 IP アドレス，アプリケーションが接続を待ち受ける仮想ポート番号と実ポート番号の対応を書く．仮想 IP アドレスは本ミドルウェアが自動で割り振ることも可能であり，その場合は auto と記述する．仮想ポート番号と実ポート番号の対応に関しては前述のとおり，省略も可能である．file は仮想ファイル空間と実ファイル空間の対応を定義する．具体的には，仮想ファイル空間におけるファイルパスと，それに対応する実ファイル空間におけるファイルパスを与える．group は複数の仮想ノードをグループ化するための定義であり，グループごとのコマンドの入力や結果の解析を行うことができる．latency は遅延を発生させるリンクとその遅延時間を定義する．group と latency に関しては，定義するかどうかは任意である．仮想分散環境の設定においてはループ構造を利用して少ない記述で多くを定義することができる．

例として，実際に 1 台に 200 の仮想環境を定義し，Mutella を動作させる場合の設定を**図 5** に示す．まず計算機名を compute0 とし，この計算機に実 IP アドレス 176.11.15.158 が割り当てられていることを示している．そして，仮想ノードをループ構造で定義している．ループは最初に “for（使用する変数；初期値；最終値；ループごとの変数の増加値)” の形式で宣言する．その後，ループする区間を中括弧でくくり，宣言した変数を前に “@” を付けて使用する．この例の場合，変数 i を 1 から 200 まで 1 つずつ増加させながら，node1，node2，…と繰り返し定義している．ループはネストすることも可能である．type は/usr/local/bin/mutella と Mutella のパスを指定し，IP は auto により自動で仮想 IP アドレスを割り当てている．file は変数を使用して仮想ファイルパスと実ファイルパスの対応を記述している．その後，group により計算機 compute0 で生成されるすべての仮想ノードを compute0 にグループ化し，latency で node3 と node150 への通信に 1 秒間の遅延を発生させるように定義している．

3.3　テストの自動化

分散システムのテストでは，各ノード上でアプリケーションを実行し，さらにアプリケーションに対してコマンドを入力しなければならない．たとえば P2P ファイル共有システムの動作検証では，他ノードに対する接続要求，クエリ送信，ダウンロードの開始等を行わせるコマンドを入力する必要がある．各アプリケーションの出力についても，各ノードに対して確認を行わなければならない．そのため，数千規模のノードを用いたテストにおいてはその労力は大きく，入力ミス等も起こりうる．また同じテストを再現することは難しい．

本ミドルウェアではこういった開発者の労力を軽減するため，1 つの仮想端末（kterm 等）から複数の仮想ノードに対して，コマンドの入力や出力の確認を行うことができる．コマンドはアプリケーションに対するコマンドだけでなく，仮想環境の起動・終了や NAT への接続等を行うためのコマンド，後述のデバッグ支援機構やテスト結果解析機構のためのコマンドについても 1 つの仮想端末から入力を行うことができる．また，ユーザが事前にシナリオ（実行手順）を記述することでテストを自動的に行うことができる．テストを自動化することにより，労力をさらに軽減し再現性を持たせることを可能とさせる．

本ミドルウェア起動時に，設定において master が定義された計算機上にコマンドを受け

付けるシェルが起動する．コマンドの入力は “コマンド#実行対象” の形式で行う．実行対象の部分に与えるものは仮想ノード名，仮想環境の設定で定義したグループ名，すべての仮想ノードを意味する文字列 `all` のいずれかである．複数の仮想環境のアプリケーションにコマンドを実行させる場合は “仮想ノード．仮想ノード．…” のように複数の仮想ノードをピリオドで区切り記述する．実行対象にグループを指定した場合，グループに所属するすべての仮想ノードにコマンドを実行させる．実行対象に `all` を指定した場合，すべての仮想ノードにコマンドを実行させる．

テストの自動実行はユーザが事前に定義したシナリオに基づき，シナリオマスタが自動的にテストを実行する．シナリオにおいては，“コマンド#実行対象 [時間]” の形式で記述する．コマンドは前回コマンドが実行された時間から [時間] で指定した時間後に実行される．たとえば，“result#node1[2]” と記述した場合，前回のコマンドが実行された時間から 2 秒後に node1 に対して result コマンドが実行される．シナリオの記述においても前述のループ構造や簡易表記が使用可能であり，簡潔に記述できる．

3.4　仮想環境の状況を提示する GUI

本ミドルウェアが取得したノードの状態やネットワーク・通信状況を基に，アニメーション表示機能により仮想環境の状況の表示を行う．これにより，仮想環境の状況を直感的に理解し，デバッグや実験結果の解析を効率的に行うことができる．実装には TouchGraph [17] ライブラリを用いている．**図 6** に 50 の仮想環境を構築し，P2P アプリケーションを動作させたときの GUI のスナップショットを示す．ノード名を囲む四角形が各仮想ノードを，それらを結ぶ線がリンクを表している．通信はリンクの色が点滅することにより表される．GUI はノードやリンクの生成・削除により，そのトポロジを動的に変化させる．表示はテスト中にリアルタイムに行うことも，テスト終了後に行うこともできる．

数千規模の仮想ノードを用いたテストを行う場合，表示が見にくくなり直感的に状況を理解することが困難となる場合がある．そのため，ユーザがテスト開始からの時間を基準として表示する時間，対象ノード，対象範囲，表示速度を指定することを可能にしている．また ImageMagick [18] を用いて，動画としてアニメーションを保存することも可能である．

3.5　デバッグ支援機構

多くのノードが同時に実行される分散システムのデバッグ作業には大きな手間がかかる．分散システム内の複数のプロセスに対して複数のデバッガを起動し，各デバッガに対して操作を行わなければならないことがあるからである．この問題に対し，本ミドルウェアのデバッグ支援機構は，前述のとおり仮想環境上の複数のノードに対するアプリケーションの入出力同様，1 つの仮想端末からデバッグを行うことを可能にする．

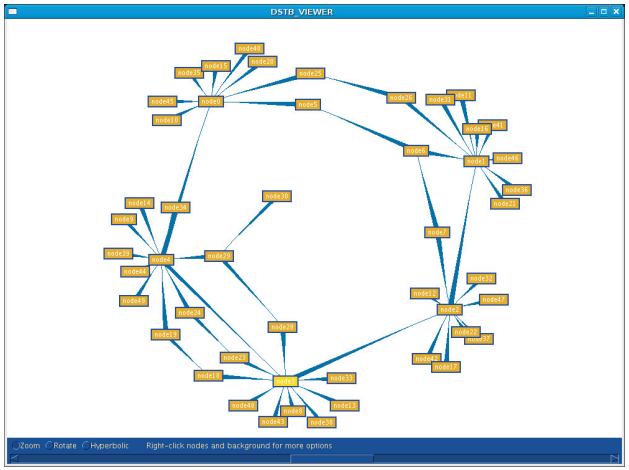

図 6　仮想環境の状況を提示する GUI
Fig. 6　GUI monitor that shows states of virtual environment.

デバッグ支援機構はコマンドラインによるユーザの入力によって，またはメモリアクセスエラー等アプリケーションに異常が発生した場合には自動的に起動される．実装には GDB を用いており起動するとユーザの指定した，もしくは異常の発生したアプリケーションに対応するプロセスに GDB をアタッチさせる．

具体的な使用例を**図 7** に示す．使用例ではまず仮想ノード node1 と仮想ノード node150 に対して gdb_start コマンドによりデバッグ支援機構を起動している．コマンドの入力は，アプリケーションへの入力と同様の形式で “コマンド#実行対象” で指定する．その後，bt:node1.node150 コマンドにより node1 と node150 のスタック情報を表示している．

3.6　テスト結果解析機構

テスト結果解析機構により，ログの解析にかかる開発者の負担を軽減する．テスト中のネットワーク・通信状況やノードの状態のログを記録し，その情報をグラフやアニメーション，テキストを用いて表示を行う．具体的には，テスト中の通信回数，通信エラー回数，接続回数，接続エラー回数について時間の経過にともなう変化とランキングを仮想分散実行環境全体，グループ，各仮想ノードごとに表示する．**図 8**，**図 10**，**図 11** は 6,000 の仮想環境上，**図 9** は 400 の仮想環境上でアプリケーションを動作させたときの表示例である．6,000

```
DSTB⟩gdb_start#node1.node150
DSTB⟩
node150_gdb_out:
Using host libthread_db library "/lib/tls/libthread_db.
so.1".
Attaching to program:  /home/ds/sample, process 8532
Failed to read a valid object file image from memory.
Reading symbols from /lib/tls/libc.so.6...done.
Loaded symbols for /lib/tls/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
0xb7f2495e in write () from /lib/tls/libc.so.6
(gdb)
DSTB⟩
node1_gdb_out:
Using host libthread_db library "/lib/tls/libthread_db.
so.1".
:
:
0xb7f6c8de in read () from /lib/tls/libc.so.6
(gdb)
DSTB⟩bt#node1.node150
DSTB⟩
node1_gdb_out:
#0 0xb7f6c8de in read () from /lib/tls/libc.so.6
#1 0x080485a8 in main () at sample.c:21
(gdb)
DSTB⟩
node150_gdb_out:
#0 0xb7f2495e in write () from /lib/tls/libc.so.6
#1 0x08048588 in main () at sample.c:20
(gdb)
```

図 7　デバッグ支援機構使用例
Fig. 7　Example of utilizing the debugging support mechanism.

の仮想環境を構築したときの例では，設定により計算機ごとに compute0〜compute19 まで 20 のグループを定義し，各計算機で動作する仮想ノードをグループ化している．

図 8 は仮想分散実行環境全体のテスト開始から終了までの時間における 30 秒ごとの通信量を表している．表示する時間帯，通信量の表示を何秒ごとにするかは変更可能である．このグラフからピーク時には 30 秒間で 60,000 回以上の通信が発生していることが分かる．

図 9 は仮想ノード node5 において，テスト開始から終了までの接続数の変化を示してい

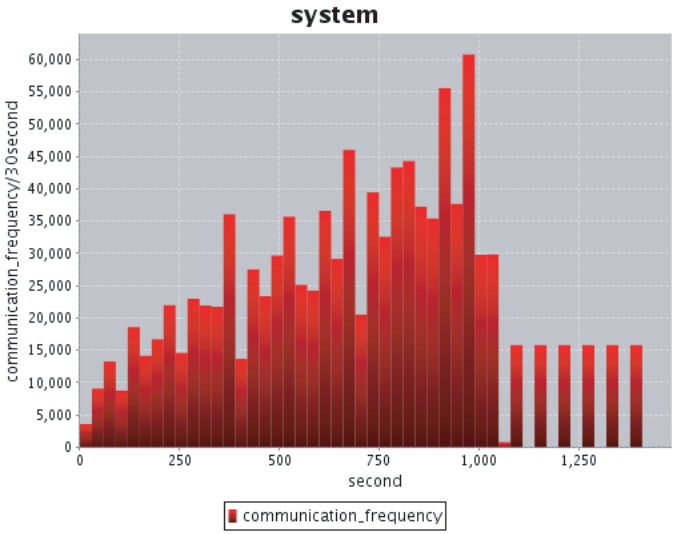

図 8　時間経過にともなう通信量の変化
Fig. 8　Change of communication frequency with passage of time.

る．これに関しても表示する時間帯の指定が可能である．この例では，テスト開始直後に接続を行っているが，すべての仮想環境上のアプリケーションがいっせいに接続を開始している影響で切断されてしまっている．その後，接続数を増やし最大で 4 つの接続を維持している．

図 10 はグループ compute3 に属する仮想ノードの接続数のランキングを上位 10 まで表示している．表示する順位の範囲は変更可能である．この例において，3 つの仮想ノードの接続数が突出しているのは使用したアプリケーションがサーバを用いるハイブリッドタイプの P2P アプリケーションであり，この 3 つの仮想ノードでサーバが動作しているからである．

図 11 はグループ単位での通信回数のランキングを表示している．この例では，すべての計算機において同じ数の仮想環境を構築しているので，グループごとの通信回数に大きな差がないことが分かる．

このように，グラフ化を行うことでどの時間帯に接続や通信，もしくはそのエラーが頻発しているのか，またどの仮想ノードやグループに集中しているのかが容易に理解できる．こ

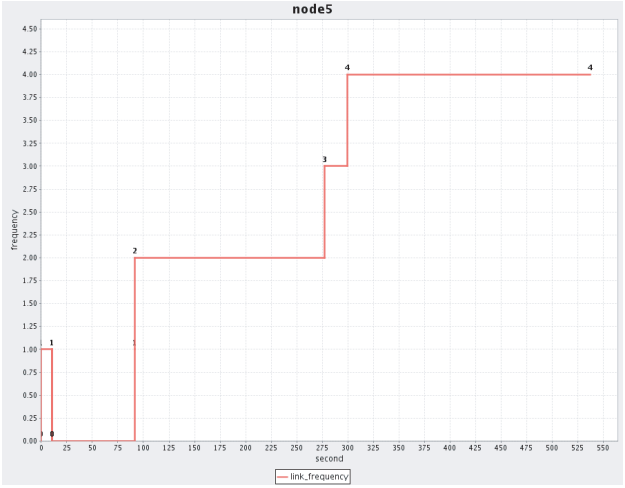

図 9　時間経過にともなう接続数の遷移
Fig. 9　Transition for the number of connections with passage of time.

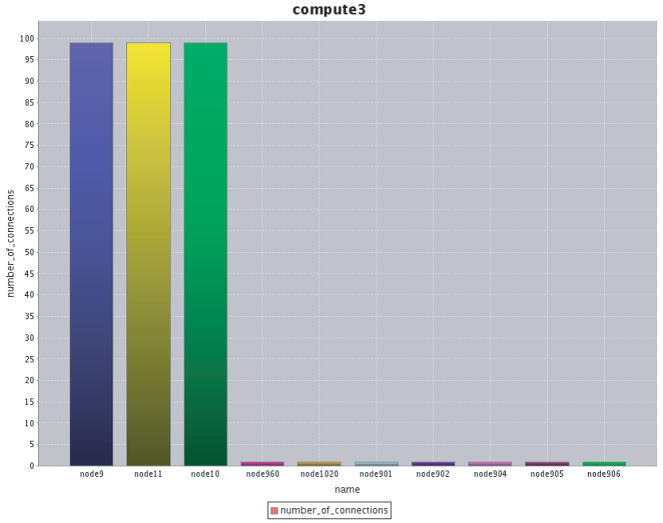

図 10　グループ内の接続数ランキング
Fig. 10　Ranking for the number of connections within the group.

れにより，通信や接続の偏りを容易に理解でき，異常な動作の発見，またネットワーク形成のアルゴリズムの正当性，パラメータの値によるネットワーク形成の変化を効率的に検証できる．グラフによる解析により，気になるところがあれば前述の仮想環境の状況を提示する GUI により，アニメーションにより視覚的に確認することもできる．また，さらに詳しい情報が必要な場合，テキスト形式で通信・接続が発生した正確な時間や通信における相手ノード，回数，また致命的エラーで終了したノード等の詳細な情報を提示することもできる．すべてのグラフやテキスト形式による情報はファイルとして保存可能である．

4. 議　　論

4.1 システムコールフックの手法

本研究では仮想環境構築のため ptrace によりシステムコールをフックしているが，システムコールをフックする他の手法としては，カーネルの拡張もしくはカーネルモジュールを用いることでカーネルレベルでフックする手法[19),20)] と動的リンクライブラリ（DLL）を用いることによりユーザレベルでフックする手法[21)] がある．カーネルレベルでのシステムコールのフックは，コンテキストスイッチが発生しないため高速である．しかし，管理者権限が必要となるうえに，カーネルに修正を加えたりモジュールをロードしたりすることは，ユーザのシステム導入の障壁を高くしてしまい，手軽にテストを可能にするという本研究の目的に反する．DLL を用いる手法では，フックライブラリを環境変数 LD_PRELOAD によってプリロードすることにより，仮想化が必要なシステムコールをオーバライドする．DLL を用いる手法はカーネルレベルでの手法と同様にコンテキストスイッチが発生しないため，ptrace を用いる手法に比べ高速である．しかし，静的リンクされたアプリケーションや，int 0x80 や sysenter を実行するアセンブリコードによりシステムコールを直接呼び出すアプリケーションでは，システムコールをフックすることはできない．

4.2 他のテストベッドとの検証内容の比較

テストベッドは一般的にネットワークエミュレーションテストベッドとオーバレイネットワークテストベッドの 2 つに分けられる．ネットワークエミュレーションテストベッドは，クローズドな環境にユーザの要求に沿ったホスト・ネットワーク環境を人工的に構築するものであり，Netbed [2)]，ModelNet [22)]，StarBED [11)]，AnyBed [23)] 等がある．ネットワークエミュレーションテストベッドでは，ユーザは様々なホスト・ネットワーク環境で実験を行

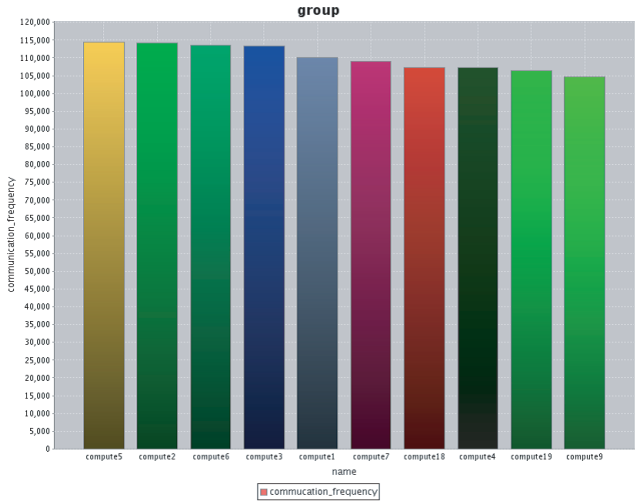

図 11　グループ単位の通信回数ランキング
Fig. 11　Ranking of communication frequency with groups.

うことが可能であるため，デバッグや性能向上のテストに有用である．しかし，現実世界のトポロジや通信の特性等のネットワーク状態は様々な要因によって複雑に変化するため，こういった人工的なネットワーク環境における実験だけでは不十分なこともある．

それに対しオーバレイネットワークテストベッドは，実環境であるインターネット上にテスト環境を構築するものであり，PlanetLab[1)] 等がある．オーバレイネットワークテストベッドでは，インターネット上にテスト環境が構築されることから現実世界のネットワーク状態を反映することができる．そのため，より実際の運用時に近い環境でアプリケーションをテストすることが可能である．しかし，実環境であるインターネットに接続されているので，テストするアプリケーションが実ネットワークに影響を及ぼす可能性がある．

現実のネットワーク状態を反映させることができるネットワークエミュレーションテストベッドとして Flexlab[24)] がある．Flexlab は Netbed を提供する Emulab によって開発されており，Netbed をベースにしている．Flexlab は PlanetLab 上の環境においてネットワークデータを収集し，それに基づくパラメータをテストベッドに反映することにより，より現実世界に近いネットワーク環境を構築する．それにより，ネットワークエミュレーションテストベッドとオーバレイネットワークテストベッドの両方の利点を実現する．

本システムでは開発者がテスト環境を構築する実環境により，現実世界のネットワーク状態を反映したテスト環境とネットワークエミュレーションによる人工的なテスト環境の両方のテスト環境において実験を行うことができる．開発者が実際のインターネット上に本システムによってテスト環境を構築し，各計算機の仮想環境を他の計算機の仮想環境に接続させた場合は，インターネットを通して通信が行われ現実世界のネットワーク状態を反映することができる．また，クローズドなネットワーク環境においてテスト環境を構築した場合は，本システムのネットワークエミュレーション機能により人工的なネットワーク環境でテストを行うことができる．既存のネットワークエミュレーションテストベッドは，本システムよりも正確なネットワークエミュレーションを実現できる．よって，本システムでのみ検証可能な事象は存在しない．本システムの意義は，カーネルの修正や実験用ネットワーク機器の導入等の面倒な作業を行うことなく様々な台数の計算機上に仮想分散環境を簡単に構築できる点にある．

4.3　スケジューリングの影響

本システムでは 1 台の実計算機上で多数のアプリケーションを動作させるため，スケジューリングによっては同時刻に行われるべき並列な通信が起こらなかったり，通信のタイミングのずれが生じたりする可能性がある．しかし，実環境においてもネットワークの遅延・ジッタが発生し通信のずれが生じる可能性があり，時刻についても計算機間で時刻のずれが生じている場合もある．すなわち，実環境上での実行においてもスケジューリングの影響で実験結果が変わる可能性がある．一般に，実験結果の再現性はオーバヘッドや実装の簡潔さとトレードオフの関係にある．本システムでは，通信やスケジューリングのタイミングを制御することはせず，タイミングに影響されない動作のテストのみを適用対象としている．

5. 評 価 実 験

仮想分散環境上でのアプリケーションの実行時に，本ミドルウェアによって生じるオーバヘッドの測定と本ミドルウェアのスケーラビリティを評価するため 2 つの実験を行った．

5.1　Apache・ApacheBench による評価

Apache とそのベンチマーク・ツールである ApacheBench を用いてオーバヘッドの測定を行った．実験は，Intel Pentium M 1.73 GHz・主記憶 1.2 GB，Intel Core Duo 2 GHz・主記憶 2 GB の 2 台の計算機を用いて行った．ネットワークインタフェースは 2 台とも Gigabit イーサネットである．

まず実環境において，1 台の計算機に Apache を動作させ，もう一方の計算機上で

表 1　ApacheBench による性能測定の結果
Table 1　Performance results with ApacheBench.

	実環境	仮想分散環境
処理に要した時間（秒）	3.83	17.99
リクエスト処理数/秒	2,611.67	558.81

ApacheBench を実行し，同時接続数を 100 として 10,000 リクエストを処理するまでの性能を測定する．リクエストするファイルのサイズは約 30 KB である．次に，Apache・ApacheBench を仮想分散環境上で動作させ同様に測定を行う．

実験の結果を**表 1** に示す．すべてのリクエストについて実環境，仮想分散環境共に成功し問題なく終了した．この結果から，本ミドルウェアにより 4.7 倍程度のオーバヘッドが発生していることが分かる．本ミドルウェアによるオーバヘッドはファイルパスを引数に持つシステムコールとネットワークアドレスを引数に持つシステムコールの実行頻度に大きく影響される．この実験では，リクエストのたびに接続・ファイル要求が行われるため，短い時間にそれらのシステムコールが非常に頻繁に発生し，オーバヘッドが大きくなってしまっている．TCP による通信の場合，通信路の構築時のみネットワークアドレスを引数に持つシステムコールが発生する．P2P システムでは，通常ある程度の時間接続は維持され通信が行われるので，オーバヘッドはより小さいものになると考えている．

5.2　スケーラビリティの評価

本ミドルウェアのスケーラビリティを評価するため，ハイブリッド P2P アプリケーションの動作を模倣したテストプログラムを用いて実験を行った．実際の P2P アプリケーションでは，1 つの実計算機上で同時に複数実行することができない．また実行時間の計測が困難なためテストプログラムを用いている．実験には，5 台の Dual Core AMD Opteron 2.2 GHz・主記憶 2 GB の計算機を用いた．ネットワークインタフェースはすべて Gigabit イーサネットである．

テストプログラムは，まず実計算機ごとに 1 つのサーバを起動し各サーバ間で接続を行う．次に実行数に応じた数のピアが起動し，異なる実計算機で動作するサーバに接続を行う．すべてのサーバには同数のピアが接続される．5 台の実計算機で 1,000 個テストプログラムを実行する場合であれば，各サーバごとに 199 個のピアが接続される．その後，1 つのピアがサーバにメッセージを送信し，各サーバを経由してすべてのピアにそのメッセージを転送するという処理を 10,000 回行う．最後に 1 つのピアがメッセージを返し，リクエストを送信したピアと直接接続し 10 MB のファイルを転送する．すべてのメッセージサイズは 100 バイトである．このテストプログラムを 5 台の計算機で，10 個，100 個，500 個，1,000 個を仮想環境と実環境で実行し，その実行時間を計測した．ただし，仮想環境で動作させるプログラムをそのまま実環境で複数同時に動作させることはできないため，バインドや接続する IP アドレス・ポートの変更等を行っている．

表 2　テストプログラムの実行時間（秒）
Table 2　Execution time of the test program.

実行数	10	100	500	1,000
実環境	0.27	1.96	6.77	13.10
仮想分散環境	1.56	31.66	573.32	2,441.06

実験の結果を**表 2** に示す．この結果から，各実行数におけるオーバヘッドは，10 個で 5.8 倍，100 個で 16.2 倍，500 個で 84.7 倍，1,000 個で 186.3 倍であり仮想ノード数を増加させるにつれ，オーバヘッドがほぼ線形に大きくなることが分かる．本ミドルウェアでは，1 つの仮想ノードにつき 1 つのスレッドが仮想化処理を行う．そのため，仮想ノード数が増加すると本ミドルウェアのスレッド数も同様に増加し，それぞれ仮想化処理を行うため全体のオーバヘッドが線形に大きくなると考えられる．またオーバヘッドが大きい要因としては，使用したテストプログラムが計算を行わず，ほぼ read・write システムコールのみを繰り返し実行しているためであると考えられる．実際の P2P システムでは，通常通信システムコールのみを繰り返し実行するということはなく，またこれほど連続して通信を行うということもないため，オーバヘッドはより小さいものとなると考えている．

6. まとめと今後の課題

1 台または数台の計算機上に数百〜数千の仮想環境を生成することで分散システムのテストを可能とし，また様々な機構により開発の支援を行うミドルウェアを提案した．実際に 20 台によるクラスタ上に 6,000 の仮想環境を生成し，その上で P2P ファイル共有システムを問題なく実行できることを確認した．

今後の課題としては，アプリケーションの異常をより効率的に開発者が検出できるようにすることである．そのために，開発者がどういった状態が異常なのかを定義できるような枠組みを設計・開発する必要がある．

謝辞　本研究の一部は，IPA 未踏ソフトウェア創造事業（未踏ユース），および科学技術振興機構戦略的創造研究推進事業「自律連合型基盤システムの構築」の支援により行われた．

参 考 文 献

1) Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M. and Bowman, M.: PlanetLab: An Overlay Testbed for Broad-Coverage Services, *ACM SIGCOMM Computer Communication Review*, Vol.33, No.3, pp.3–12 (2003).
2) White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C. and Joglekar, A.: An Integrated Experimental Environment for Distributed Systems and Networks, *Proc. 5th Symposium on Operating Systems Design and Implementation* (*OSDI '02*), Boston, MA, pp.255–270 (2002).
3) Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the Art of Virtualization, *Proc. 19th ACM Symposium on Operating Systems Principles*, pp.164–177 (2003).
4) VMware. http://www.vmware.com/
5) Dike, J.: A user-mode port of the linux kernel, *Proc. 4th Annual Linux Showcase & Conference* (2000).
6) Linux-VServer. http://linux-vserver.org/
7) Osman, S., Subhraveti, D., Su, G. and Nieh, J.: The Design and Implementation of Zap: A System for Migrating Computing Environments, *Proc. 5th Symposium on Operating Systems Design and Implementation* (*OSDI '02*), Boston, MA, pp.361–376 (2002).
8) 大山恵弘，神田勝規，加藤和彦：安全なソフトウェア実行システム SoftwarePot の設計と実装，コンピュータソフトウェア，Vol.19, No.6, pp.2–12 (2002).
9) ns-2. http://www.isi.edu/nsnam/ns/
10) Bajaj, L., Takai, M., Ahuja, R., Tang, K., Bagrodia, R. and Gerla, M.: GloMoSim: A Scalable Network Simulation Environment, UCLA Computer Science Department Technical Report, Vol.990027 (1999).
11) Miyachi, T., Chinen, K. and Shinoda, Y.: StarBED and SpringOS: Large-scale General Purpose Network Testbed and Supporting Software, *Proc. International Conference on Performance Evaluation Methodologies and Tools* (*Valuetools*) *2006* (2006).
12) 首藤一幸，田中良夫，関口智嗣：オーバレイ構築ツールキット Overlay Weaver，情報処理学会論文誌：コンピューティングシステム，Vol.47, No.SIG12 (ACS 15), pp.358–367 (2006).
13) Gtk-Gnutella. http://gtk-gnutella.sourceforge.net/
14) Mutella. http://mutella.sourceforge.net/
15) DB Hub. http://www.dbhub.org/
16) microdc2. http://corsair626.no-ip.org/microdc/
17) TouchGraph. http://www.touchgraph.com/
18) ImageMagick. http://www.imagemagick.org/
19) Provos, N.: Improving Host Security with System Call Policies, *Proc. 12th USENIX Security Symposium*, Washington, DC (2003).
20) Suranyi, P., Abe, H., Hirotsu, T., Shinjo, Y. and Kato, K.: General Virtual Hosting via Lightweight User-Level Virtualization, *Proc. 2005 Symposium on Applications and the Internet* (*SAINT '05*), Washington, DC (2005).
21) Nussbaum, L. and Richard, O.: Lightweight Emulation to Study Peer-to-Peer Systems, *3rd International Workshop on Hot Topics in Peer-to-Peer Systems* (*HotP2P '06*), Rhodes Island, Greece (2006).
22) Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kostic, D., Chase, J.S. and Becker, D.: Scalability and Accuracy in a Large-Scale Network Emulator, *Proc. 5th Symposium on Operating Systems Design and Implementation* (*OSDI '02*), Boston, MA (2002).
23) 鈴木未央，櫨山寛章，門林雄基：AnyBed：クラスタ環境に依存しない実験ネットワーク構築支援機構の設計と実装，情報処理学会研究報告システムソフトウェアとオペレーティング・システム，Vol.2004, No.63, pp.121–128 (2004).
24) Ricci, R., Duerig, J., Sanaga, P., Gebhardt, D., Hibler, M., Atkinson, K., Zhang, J., Kasera, S. and Lepreau, J.: The Flexlab Approach to Realistic Evaluation of Networked Systems, *Proc. 4th Symposium on Networked Systems Design and Implementation* (*NSDI '07*), Cambridge, MA (2007).
25) 西川賀樹，大山恵弘，米澤明憲：OS 資源ビューの仮想化を用いた分散システムテストベッド，第 6 回情報科学技術フォーラム（FIT2007），pp.383–386 (2007).

(平成 20 年 1 月 29 日受付)
(平成 20 年 5 月 18 日採録)

西川　賀樹（学生会員）

1980 年生．東京大学大学院情報理工学系研究科コンピュータ科学専攻博士課程 2 年．興味はシステムソフトウェア，仮想計算環境，並列分散処理．

大山　恵弘（正会員）

1973 年生．2001 年東京大学大学院理学系研究科情報科学専攻修了．博士（理学）．科学技術振興事業団研究員，東京大学大学院情報理工学系研究科助手を経て，現在，電気通信大学情報工学科准教授．興味はシステムソフトウェア，セキュリティ，プログラミング言語，並列分散処理．

米澤　明憲（正会員）

MIT 計算機科学修了（Ph.D.）．並列オブジェクトの研究により米国学会 ACM の Fellow．1988 年より東京大学大学院コンピュータ科学専攻教授．日本ソフトウェア科学会理事長，ドイツ国立情報科学技術研究所（GMD）科学顧問歴任．現在，産業技術総合研究所情報セキュリティセンター副所長，東京大学情報基盤センター長．2008 年国際オブジェクト技術協会（AITO）より，オブジェクト指向技術への多大な貢献者に与えられる，ダール/ニゴール賞を受賞．