LL-007

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

Distributed System Test Bed by Virtualization of OS Resource View

西川　賀樹† 大山　恵弘‡ 米澤　明憲†

Yoshiki Nishikawa Yoshihiro Oyama Akinori Yonezawa

1. はじめに

本研究は、P2P・ネットゲーム等の分散システムの開発を、仮想化技術を用いて 1 台の計算機上で行う手法を提案する。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。これに対し本研究の提案手法では、1 台の計算機上に多数の仮想環境を生成し、それらの仮想環境間の通信を制御・監視することで、分散システムの開発・テスト環境を低コストで実現する。

仮想マシンを用いて分散システムのテスト環境を構築する上で問題となるのが、従来の仮想マシンでは、仮想化のオーバーヘッドのために 1 計算機上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・テストに最低限必要な OS 資源のみを仮想化することで、多数の仮想環境を 1 計算機上に実現できるようにした。実際に CPU が Intel Core Duo 2GHz、メモリが 2GB のマシン上で、100 個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。

また本研究では、提案手法に基づき、分散システムの開発を支援するミドルウェアシステムを実装した。このシステムは、ユーザから与えられた仮想分散環境のノードとネットワークに関する仕様を基に、ノードごとに仮想環境を生成し、それらの仮想環境間の通信を制御・監視する。またこのシステムは、ネットワークトポロジの変化や通信状況の提示を行う GUI、仮想環境上で動作している多数のアプリケーションをまとめて管理・制御できるデバッグ支援機構も提供する。本ミドルウェアは Linux 上に実装されており、ptrace システムコールを用いて仮想環境を構築する。具体的には、システムコールを監視・制御することにより、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、より低い層の仮想化（CPU やデバイスの仮想化）を必要とするテストは対象としない。

2. 分散システムテストベッドの必要性

近年、ネットワークの高速化によって P2P システムに代表される分散システムが数多く開発されるようになった。しかし従来のクライアント・サーバモデルが一対一通信でデータのやり取りを行うのに対し、これらのシステムは、常に数多くのマシンが相互に通信を行う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、多数のマシン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。

このため、P2P 型の分散システムのテスト・検証を行うには、多数の計算機が必要となる。しかし、開発者がテスト環境構築の為に実機を用意することは、コストが大きい。この問題に対しては、近年発展した仮想マシンを用いて、仮想的に計算機を用意するという手法が考えられる。しかし、既存の仮想マシンの多くは大量の資源を消費する為、1 台の計算機上に実現できる実行環境が、高々数個から数十個に限られるという問題がある。またテスト環境を用意でき実行できたとしても、どのマシン同士が通信しているか等の通信状態が把握しにくいという問題もある。

現在分散システムのテスト環境としては PlanetLab[1]や Netbed[2]、StarBED[3]等のテストベッドやネットワークシミュレータ(ns-2[4]等)が利用されている。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、計算リソースの提供が必要等の条件がある場合もあり、手軽には利用できない。またネットワークシミュレータはテスト専用のコードを記述してシミュレーションを行うものであり、アプリケーションを実際に実行してテストを行うものではない。アプリケーションを実行できるシミュレータに関しても専用 API の使用等、アプリケーションの修正が必要となる場合がある。

3. 提案するミドルウェア

本研究では 1 台の計算機上に多数の仮想実行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 GUI・デバッグ支援機構により開発の支援を行うミドルウェアを提案する。本ミドルウェアは全てユーザレベルで実装されているため、オペレーティングシステムやアプリケーションコードの修正、管理者権限は不要である。このため、ユーザが容易に使用でき、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。テストの度に多数の実機を用意したり、テストベッド利用申請をしたりする必要もない。

3.1 仮想環境の設定

分散システムの種類・性質により、開発・テストに最適なネットワーク環境は異なるため、本ミドルウェアで

† 東京大学大学院情報理工学系研究科

‡ 電気通信大学電気通信学部情報工学科

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の指定
- ノード間の通信遅延
- 各ノードに割り当てる仮想 IP アドレス・ポート番号・仮想ファイル空間
- ノードの動的な参加・脱退

この設定により、各ノードは固有の IP アドレス・ポート番号と独立したファイル空間を仮想的に持つことができる。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。

ユーザが与えたいネットワークトポロジ、動作検証に関するパラメータ、仮想環境に関する設定を表現するための記述言語の仕様を図 1 に示す。まず create は、仮想ネットワーク環境内のノードを指定する。各ノードは、ノード名と type からなる。type にはノード上で実行するアプリケーションを記述する。次に各ノードに対して、仮想 IP アドレス・ポート番号とそれに対応する実 IP アドレス・ポート番号を記述する。link は、create で生成したノード間にどのような伝送路があるかを指定する。latency には伝送路の遅延時間を記述する。また nodectl は実行時に動的に生成・消滅するノードを指定する。ctime にはノードが生成される時刻、dtime にはノードが消滅する時刻を指定する。最後に仮想ノードのファイル空間について仮想パスと実パスの対応を記述し定義する。

```
/*ノードの生成*/
create = nodeA[type]:...:nodeN[type];
/*仮想 IP アドレス・ポート番号の設定*/
nodeA[addr] = [virtual_IP_port][real_IP_port];
/*伝送路の設定*/
link = nodeA:nodeB[latency];
/*ノードの生成・削除*/
nodectl = nodeA[ctime][dtime];
/*仮想ファイル空間の設定*/
nodeA[file] = [virtual_path][real_path];
```

図 1: 仕様記述言語

3.2 GUI

GUI はアプリケーションプロセスをモニタすることによって得た通信やプロセスの状態等の情報を基にアニメーションを表示する。分散システムでは多くのノードが相互に通信を行い動作するので、実行状況の把握が難しい。これに対し本システムの GUI は、リアルタイムに仮想環境の状況を提示し、ユーザは各ノードの動作やネットワークの変化を確認することができる。これにより、ユーザが実装したアプリケーションが意図した通りにネットワークを形成・再構築しているか、通信を行っているか等の動作の確認が容易となる。また通信ログ等の解析の負担も少なくなる。

3.3 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において、デバッグ作業を行うことは困難である。なぜなら、分散システム内のノードに対応する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からである。この問題に対し、本システムのデバッグ支援機構は、仮想環境上の複数のノードに対して一つの仮想端末(kterm や xterm 等)からデバッグを行うことを可能にする。また GUI による情報により、仮想環境の状況を把握しながらデバッグ作業を効率的に行うことができる。デバッグ支援機構はユーザからの入力、もしくはメモリアクセスエラー等ノードに異常が発生した場合は自動的に起動される。それによりノードに異常が発生した時のノードの状態が把握でき、原因の究明をより効率的に行える。

4. 実装の詳細

4.1 仮想化

本ミドルウェアでは、必要最小限の OS 資源に対してのみ仮想化を行う。このため、既存の仮想マシンに比べ、より多くの仮想環境を構築し、また高速に動作させることができる。具体的には、ファイルとネットワークのみ仮想化を行い、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。

本ミドルウェアでは ptrace システムコールを用いてユーザレベルで実装を行う。ptrace を用いた仮想化の流れを図 2 に示す。

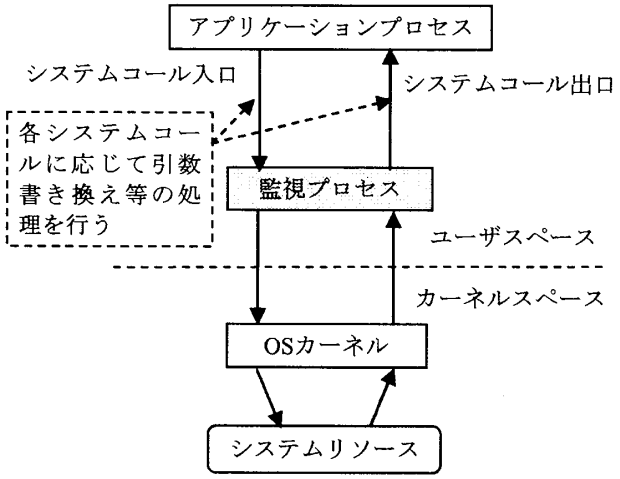

図 2: 仮想化の流れ

まず各監視プロセスがアプリケーションを実行し ptrace により監視する。アプリケーションがシステムコールを発行すると、監視プロセスはそのシステムコールをフックし、アプリケーション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,chmod 等

・socketcall システムコール－bind,connect,accept 等

・プロセスの生成・新たなジョブの実行－clone,fork,exec

・ソケット記述子を引数に持つ－read,write,send 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。監視プロセスは 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プロセス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。

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。監視プロセスは 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。ただし、実際に使用する IP アドレスは一つであり、実 IP アドレスからは対応する仮想 IP アドレスを判断することはできない。そのため、監視プロセスはアプリケーションが使用する実ポート番号から対応する仮想 IP アドレス・ポート番号を判断する。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、監視プロセス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。その為、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、監視プロセスはアプリケーションに bind システムコールを発行させる。それにより、各監視プロセス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、監視プロセス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、監視プロセス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り込みを発生させる命令である`cd80'(int 0x80)に書き換える。その後、監視プロセス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。

アプリケーションが新たなプロセスを生成した場合、そのプロセスに対しても監視を行う必要があるので clone, fork 等のシステムコールに関しても処理を行う。

read, write, send 等のソケット記述子を引数に持つシステムコールについては、GUI によりアニメーションを表示する為に処理が必要となる。

4.2 GUI

GUI については TouchGraph[5]のライブラリを用いて実装を行った。被監視プロセスが socketcall・write・read 等の通信システムコールを発行したり、起動・終了・停止等の状態の変化が起こったりした場合に監視プロセスが GUI プロセスと通信を行うことで仮想環境の状況を提示することができる。図 3 に 50 個の仮想環境を構築し、P2P アプリケーションを動作させた時の 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、それらを結ぶ線が伝送路を表している。通信時には伝送路の色が変化する。GUI はノードや伝送路の生成・削除により、そのトポロジを動的に変化させる。

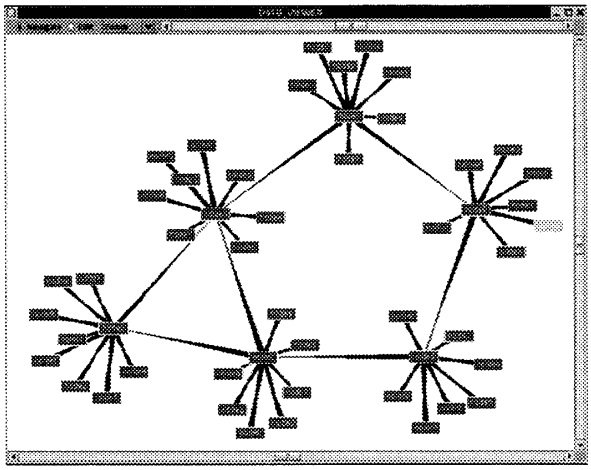

図 3: 仮想環境の状況を提示する GUI

4.3 デバッグ支援機構

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[6]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。この際、複数のプロセスが ptrace を用いて一つのプロセスを監視することはできないので監視プロセスはいったん被監視プロセスからディタッチし GDB プロセスにアタッチさせる。GDB プロセスが終了すると再び監視プロセスは被監視プロセスにアタッチする。このとき、監視プロセスがディタッチした際に被監視プロセスが実行を再開してしまうためシグナル発生時等に、GDB プロセスが再アタッチする前に、被監視プロセスが終了してしまうという問題がある。これに対処するため、監視プロセスは被監視プロセスが次に実行する命令を保存し現在のアドレスにループする命令に書き換えを行う。その後、GDB プロセスを通して本来の命令に再度書き換えを行う。また全ての GDB プロセスを管理するプロセスを作成し、このプロセスによりユーザからの入力や GDB からの出力を制御する。これにより複数 GDB プロセスを端末から使用することができる。ユーザは GDB によって提供される全ての機能を使用することができる。

図 4 にデバッグ支援機構の使用例を示す。使用例ではまず nodeA と nodeE 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。その後、bt:nodeA.nodeE コマンドにより nodeA と nodeE のスタック情報を表示している。最後に quit:all コマンドにより全てのノードに対

してのデバッグ支援機構を終了している。この例から各ノードに対する入力が正常に行えており、また各ノード毎に出力を確認できていることがわかる。

```
DSTB>gdb_start:nodeA.nodeE
nodeE_gdb_out:
Using host libthread_db library "/lib/tls/libthread db.so.1".
Attaching to program: /home/vnet/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>
nodeA_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:nodeA.nodeE
nodeA_gdb_out:
#0 0xb7f6c8de in read () from /lib/tls/libc.so.6
#1 0x080485a8 in main () at sample.c:21
(gdb)
DSTB>
nodeE_gdb_out:
#0 0xb7f2495e in write () from /lib/tls/libc.so.6
#1 0x08048588 in main () at sample.c:20
(gdb)
DSTB>quit:all
nodeA_gdb_out:
Detaching from program: /home/vnet/sample, process 8524
 DSTB>
nodeE_gdb_out:
Detaching from program: /home/vnet/sample, process 8532
DSTB>
```

図 4: デバッグ支援機構の使用例

5. 実験

Gtk-Gnutella[7]という P2P ソフトウェアを利用して本システムの実験を行った。Gtk-Gnutella は GUI により操作が可能なファイル共有ソフトである。実験には CPU が Intel Core Duo 2.0GHz、メモリが 2GB のマシンを用いた。1 台の PC 上に 100 個の仮想環境を構築し、各仮想環境上で Gtk-Gnutella を動作させた。100 個の Gtk-Gnutella は仮想環境上で現実的な性能で問題なく動作し、ファイルの検索や交換等も正常に行うことができた。図 5 に仮想環境上で動作する Gtk-Gnutella のスナップショットを示す。図 5 から Gtk-Gnutella に通信先の仮想 IP アドレスが表示されており、仮想環境が正常に構築されていることがわかる。また GUI を用いて、ノードを削除した場合や検索・ダウンロードが発生した場合の通信・ネットワークの変化等も容易に把握することができた。

図 5: 仮想環境上の Gtk-Gnutella

6. おわりに

本論文ではプロセスレベル仮想化技術を用いて多数の仮想環境を構築することで分散システムのテストが可能であり、また GUI・デバッグ支援機構を用いて開発の支援を行うミドルウェアを提案した。また実際に 1 計算機上に 100 個の仮想環境を構築し、仮想環境上で P2P アプリケーションが現実的な速度で正常に動作することを確認した。GUI・デバッグ支援機構についても問題なく動作した。今後、1 計算機上だけでなく複数の計算機上で仮想環境の構築を可能にする。それによって、より多くの仮想環境を必要とする実験にも対応する。また NAT やファイアーウォール等のように分散システムで問題となる可能性のあるネットワーク環境のエミュレーション機能や、分散システムのパラメータ設定を支援する為の機構、チェックポイント・リスタート機能を実装することにより分散システムの動作検証を更に正確かつ容易に行えるようにする。

参考文献

[1] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. PlanetLab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, Vol. 33, No. 3, pp. 3-12, July 2003.

[2] 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. In *Proceedings of Operating System Design and Implementation (OSDI'02)*, pp. 255-270, December 2002. USENIX ASSOC.

[3] Toshiyuki Miyachi, Ken-ichi Chinen, and Yoichi Shinoda. StarBED and SpringOS: Large-scale general purpose network testbed and supporting software. In *Proceedings of the 1st International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS'06)*, October 2006.

[4] ns-2. http://nsnam.isi.edu/nsnam/index.php/Main_Page

[5] TouchGraph. http://www.touchgraph.com/

[6] GDB. http://sourceware.org/gdb/

[7] Gtk-Gnutella. http://gtk-gnutella.sourceforge.net/