VESPER(Virtual Embraced Space ProbER)の設計と実装

守屋 哲　金 成昊　大島 訓
株式会社　日立製作所　システム開発研究所
{satoru.moriya.br, sungho.kim.zd, satoshi.oshima.fk}@hitachi.com

概要

本論文では，我々が開発している，仮想化環境においてホストOS上でゲストOSの情報を効率的に収集するフレームワークであるVESPER(Virtual Embraced Space ProbER)について説明する．VESPERは仮想化環境においてクラスタシステムを構築する際に，クラスタ管理ソフトウェアに対し，ノード切り替え等の判断要素となる情報を提供することを目的に開発している．VESPERでは，ホストOSからゲストOSへKprobesを挿入することで詳細情報の取得を実現した．そして，VMM[1]が提供しているVM[2]間通信機構を利用することで，クラスタ管理ソフトウェアに対し収集した情報の迅速な通知を実現した．本論文ではVESPERの基本的な構造および実装を説明後，Xen環境におけるクラスタシステムへVESPERを適用し，その有効性を示す．なお，VESPERは 2008 年 8 月にオープンソースとして公開する予定である．

1. はじめに

現在 Linux は UNIX 系 OS の中で高いシェアを占め、企業におけるファイル共有サーバ、Web サーバだけでなく、金融機関における業務システムの様な大規模システムへと適用範囲が拡大している。このような基幹業務向けシステムでは信頼性に加え可用性が重要であり，多くの場合，複数のサーバを用いてクラスタシステムを構成することで可用性の向上を図っている．

一方，近年サーバ台数の増加による管理負荷の増大やサーバシステムのハードウェア性能の著しい向上により仮想化技術が注目されており，オープンソース・コミュニティでもXen[1]やKVM[2]等の開発が活発に行なわれている．仮想化技術を使うことにより，１つの物理ハードウェアの上で複数のOSを動かすことができ，これにより既存サーバの集約やリソースの効率的な利用が可能である．

以上により、仮想化環境におけるクラスタシステムの構築・運用技術は，効率的なサーバ資源の活用及びシステムの可用性向上の観点から,非常に重要である．また,仮想化環境でクラスタシステムを構成する場合，仮想化技術が提供している機能を利用することで，障害の検知等を高速に行なうことが可能である．

Heartbeat[3][4]に代表されるクラスタ管理ソフトウェアでは，仮想化環境においても，一定間隔のメッセージ交換による生死監視を行なっている．この方法では，サービスを提供しているノードから一定時間応答が無い場合，クラスタ管理ソフトウェアは該当ノードに障害が発生したと判断し，サービス提供ノードを切り替える．このような方法では，実際に障害が発生してから，クラスタ管理ソフトウェアが障害を検知するまでに時間がかかり結果としてフェイルオーバが遅くなる，障害が発生したノードの詳細な情報が取得できない等の問題があった.この問題を解決するため,我々は，仮想化環境のクラスタシステムにおける，クラスタノードの障害検知の高速化と詳細な障害情報の取得を実現するための障害検知フレームワークの検討，開発を行なった．

2. 障害検知フレームワークの検討

従来のメッセージ交換による生死監視は，ポーリング方式であるため障害検知に遅延が生じていた．これを解決する方法の一つとして，イベントドリブン方式の障害検知がある．これは，障害と判断するイベントをあらかじめ登録しておき，運用中に該当イベントが発生した場合，即座に障害と判断しフェイルオーバを実行する方式である．この方式では，イベントの登録と検出を行なう必要があるが，サービスを提供するVMに対してLinuxのKprobes[6]に代表される動的プローブ機能を適用することで，イベントの登録と検知が可能である．動的プローブ機能により，下記項目が可能になる．

[1] Virutal Machine Monitor
[2] Virutal Machine

- システム停止が不要な動的なプローブの挿入
- 任意のアドレスへのプローブの挿入
- プローブしたイベントが発生した際の迅速な検知

仮想化環境上で動作しているクラスタシステムにおいて、上記の特徴を持った動的プローブ機能を利用し,イベントドリブン方式の生死監視を行なうことで,フェイルオーバの高速化，効率化が可能である.

Linux では動的プローブ機能として Kprobes が提供されている．Kprobes を含むカーネルモジュールを作成，ロードすることで，Linux カーネルに対し，動的にプローブを挿入することができる．カーネルがプローブされた命令を実行すると，あらかじめユーザが定義した関数が呼ばれるため，詳細な情報の取得も可能である.

一方，仮想化環境におけるクラスタシステムの構成方法として,従来はサービスを提供するゲストOSにクラスタ管理ソフトウェアをインストールし，クラスタを構成していたが,ホストOSに障害が発生した場合の挙動の設定等が複雑であることなどから,ホストOSにクラスタ管理ソフトウェアをインストールし，ゲストOSを監視する方法が提案された[5]．本方式を用いてクラスタシステムを構築する場合，クラスタ管理ソフトウェアが動作しているホストOSで,サービスが動作しているゲストOSの情報を取得しなければならない.しかし，上述したKprobesは，ホスト間のプローブ挿入機能が無いため,Kprobesを利用した動的プローブ機能を本構成で利用する場合には,次の機能が必要となる.

- ホスト OS からゲスト OS へのプローブ挿入機能
- 収集した情報のホスト OS への転送転送

本課題を解決するために,Xenprobesが提案されている[7]．XenprobesはKprobeの概念をVM間プローブに拡張したもので,ホストOSからゲストOSにプローブを挿入することが可能である．しかし，XenprobesはVMMが提供するデバッグ機能[8]を必要であり,プローブ挿入時にはゲストOSを停止しなければならない．また,プローブされた命令を実行すると,必ずホストOSに処理が遷移するため，非常にオーバヘッドが大きく，サービスの品質を確保することが難しい.

そこで，Kprobes を使った障害検知のフレームワーク VESPER を設計，実装した.

具体的には，Kprobes と relayfs を用いて OS の情報を採取するプローブモジュールを，VESPER がホスト OS からゲスト OS へロードする．そして，プローブモジュールが採取したデータを，VESPER が，VMM が提供している VM 間通信機能を用いて，ゲスト OS へ転送する．ここで，プローブモジュールが採取した情報は relayfs のバッファに格納されており，また，VM 間通信機能により,ゲスト OS とホスト OS ではメモリを共有できるため，実際にデータを転送する必要はなく,relay バッファとして利用しているメモリを共有するだけである．本方法により，VESPER 自体は，Xenprobes と異なり，プローブハンドラを保持しないため，プローブ毎のホスト OS への処理遷移は発生せず，オーバヘッドを削減できる.

3. VESPER の仕様設計

VESPER は以下の 3 点を仕様設計の方針とした.

- 異なる動作環境への対応が容易なこと
- クラスタ管理ソフトウェアはホスト OS で動作すること
- ゲスト OS のユーザ空間の障害の影響を受けにくいこと

これに基づき,VESPRE の機能要件を次のように定義した．また，VESPER は Xen 及び KVM で動作できるようにするため，それらの機能を前提に設計する.

1. ホスト OS およびゲスト OS のカーネルの変更が不要であること．カーネルモジュールとして実装することで，移植性が向上すると共に，システムへの機能の追加，削除が容易に可能になる.

2. 特定の VMM に依存した構造にしないこと. VM 間通信部等 VMM に依存してしまう部分はあるが,最小限にすることで，移植性を高める.

3. ホスト OS のみがゲスト OS へプローブモジュールを転送できること（プローブを挿入できること）.前章で述べたように，仮想化環境においてはホスト OS 上でクラスタ管理ソフトウェアが動作する.そのため，プローブの挿入や削除といった作業もホスト OS で一括管理するべきである.

4. ゲスト OS で収集した情報は即座にホスト OS へ転送すること. 収集した情報を即座にホスト OS へ渡すことで,ホスト OS 上のクラスタ管理ソフトウェアへの迅速な障害発生の通知が可能になる.

5. ゲスト OS ではモジュールをロードするときにカーネル空間だけ利用すること. ホスト OS から転送されたモジュールをゲスト OS がロードする際,ユーザ空間を使用しないことで，ユーザ空間に障害が生じている場合でも，調査が可能となる.

上記 1，2 により移植性を確保し，3，4 によりホスト OS 上のクラスタ管理ソフトウェアへ対応する．また,5 によりゲスト OS のユーザ空間の障害の影響を低減できる．

1 を満たすために VESPER は仮想デバイスドライバとして実装する．2 の実現するために，VEPSER は，階層構造を持たせ，VMM に依存する部分を最小限にする．3，4 の要件を満たすためには VESPER はホスト OS とゲスト OS の間で,何らかの通信をする必要があるが，Xen や KVM では，既にフロントエンドドライバとバックエンドドライバからなる分割ドライバ方式の VM 間通信インフラが用意されている．Xen では Xenbus が提供されており，KVM では virtio という仕組みが利用できる．また，5 では，本来ゲスト OS のユーザ空間で実施する作業を，ホスト OS のユーザ空間で実施する．階層構造や実装の詳細については次章以降で説明を行なう．

4. VESPER の構造と動作

VESPER では,ゲスト OS の情報を収集するために，プローブの挿入には Kprobes を利用し，プローブハンドラにおける情報の記録には relayfs を利用する．本章では，3 階層構造を特徴とする VESPER の構造とその動作について説明する．

4.1. VESPER の構造

VEPSER はゲスト OS をプローブするために Kprobes を利用する．しかし，Kprobes はローカルシステムへのプローブ挿入機能であるため，ホスト OS からゲスト OS といった,外部の OS に対してプローブを直接挿入することはできない．また，Kprobes を利用してプローブを挿入する場合，カーネルモジュールとして実装するのが普通である．そのため，Kprobes を利用してゲスト OS にプローブを挿入するためには, VESPER は，Kprobes を利用したプローブモジュールをホスト OS からゲスト OS にロードする必要がある．VESPER では,このモジュールロード機能を Probe Loader とし，分割ドライバ方式で実装する．

VESPERでは，プローブモジュールは，プローブ時の情報を記録するためにrelayfsを利用する．プローブモジュールはゲストOS内で動作しているため，記録したデータをゲストOSからホストOSへ転送する必要がある．VESPERでは，この記録したデータを転送する機能をProbe Listenerとし, Probe Loaderと同様に分割ドライバ方式で実装する．上記，分割ドライバ方式を用いたVESPERの全体構造を**図 1**に示す．

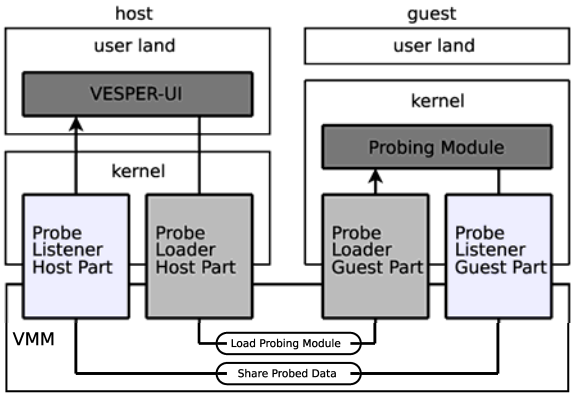

図 1　VESPER の構造

上述したように，VESPERは 2 組の分割ドライバから構成されるが, これらのドライバは下層にあるVMMに強く依存しているため, VMM毎に実装する必要がある．そのため，VESPERは，VMM依存部分を最小限にするため，図 2に示すようにUI層，Action層，Communication層からなる 3 層構造で設計した. これにより, VESPERはVMMに依存するCommunication層を入れ替えるだけで，XenやKVM上で動作することが可能になる

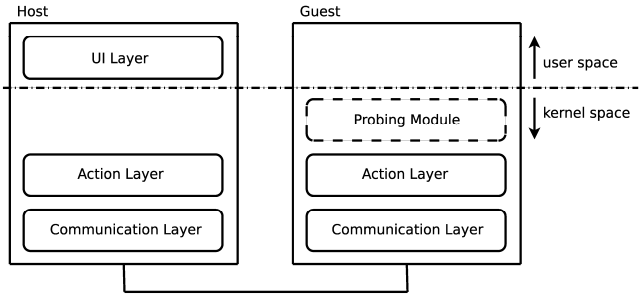

図 2　VESPER の 3 階層構造

各層の説明を以下に示す.

1. UI 層

   VESPERとアプリケーションとのインタフェースとなる層である．UI層はアプリケーションに対し，ゲストOSへのモジュールのロードやアンロード，ゲストOSで記録されたデータへのアクセス用インタフェースを提供する．これらのインタフェースは6章で説明する.

2. Action 層

   UI 層，Communication 層からの要求を処理する中心的な層である．具体的には，ゲスト OS へのモジュールのロードやアンロード, ゲスト OS とホスト OS 間でのデータ共有の実質的な処理を行なう.

3. Communication 層

ホスト OS とゲスト OS 間の通信チャネルを提供する. 本層は VMM のアーキテクチャに強く依存しているため, VMM 毎に実装する必要がある. VESPER では，VMM 間の差異をこの層で吸収する.

4.2. VESPER の動作

VESPER では，プローブをゲスト OS に挿入するために, プローブモジュールをホスト OS からゲスト OS にロードしなければならない．そして，プローブモジュールが収集したデータをゲスト OS からホスト OS へ転送しなければならない.

VESPER の処理の流れを図 3に示す.

4.2.1. モジュールのロード

ゲスト OS をプローブするための最初の処理は，ゲスト OS にプローブモジュールをロードすることである．モジュールのロード処理の流れを以下に示す．なお，A1～A8 は図 3に対応している.

0. プローブモジュールの作成

最初に Kprobes 及び relayfs を利用したプローブモジュールを作成する.

A1.モジュールロードコマンドの実行

Probe Loader が提供するモジュールロードコマンドを実行する. なお, ユーザは必要に応じてモジュールパラメタを指定することも可能である.

A2.モジュール情報の取得

ホストOS側のProbe LoaderのAction層において，A1 において指定されたモジュールに関して,その名前やサイズ，アドレス等をユーザ空間から取得する.

A3.モジュールロード要求の送信/受信

VMM が提供している VM 間通信機能を利用して, ホスト OS からゲスト OS へモジュール及び関連情報を転送する.

A4.モジュールのロード

ゲスト OS 側 Probe Loader の Action 層では,ユーザ空間を利用せずにモジュールのロードを実行する.

A5.relay バッファの共有

ゲスト OS 側 Probe Listener の Action 層が，プローブモジュールより，relay バッファのサイズやアドレス等の情報を取得し, ホスト OS との共有を行なうための処理を実行する.

A6.relay バッファ情報の送信/受信

Probe ListenerのCommunication層では，A5にてゲストOS側Action層で取得したrelayバッファに関する情報を，VM間通信機能を利用して，ゲストOSからホストOSへ転送する.

A7.relayfs 用構造体の作成

ホスト OS 側 Probe Listener の Action 層は relayfs のインタフェースを利用して，ゲスト OS の情報を UI 層に提供する．そのため，ゲスト OS から取得した情報を利用して，relayfs 管理構造体を作成する.

A8.プローブしたデータの提供/分析

UI 層では relayfs のインタフェースを利用して，ホスト OS 側 Probe Listener の Action 層から情報を取得し，ユーザアプリケーションに提供する.

4.2.2. ゲスト OS 情報の取得

4.2.1の処理が終了すると，Probe ListenerはゲストOSとホストOSの間で, プローブモジュールのrelayバッファの共有を開始する．ここで，採取した情報自体は共有バッファに格納されているため, ゲストOSからホストOSへ転送する必要はないが, 読み込み開始インデックス等のバッファ管理情報は，共有していないため，ゲストOSからホストOSへ転送する必要がある. このため，ゲストOSを監視中のVESPERの動作は下記のようになる.

B1. ゲスト OS 情報の収集

プローブモジュールは，ロードされると，プローブされた命令が実行される度に情報を採取し，relay バッファへ格納する.

B2. ゲスト OS 側インデックス情報の取得

relay バッファは複数のサブバッファから構成されている．Probe Listener ではサブバッファの切り替えが発生した際に，relay バッファのインデックス情報を取得し, ホスト OS に通知するためのメッセージを作成する.

B3. ゲスト OS 側インデックス情報の転送

VM 間通信機能を利用して，ホスト OS 側 Probe Listenerはバッファのインデックス情報を含むメッセージをゲスト OS から取得する.

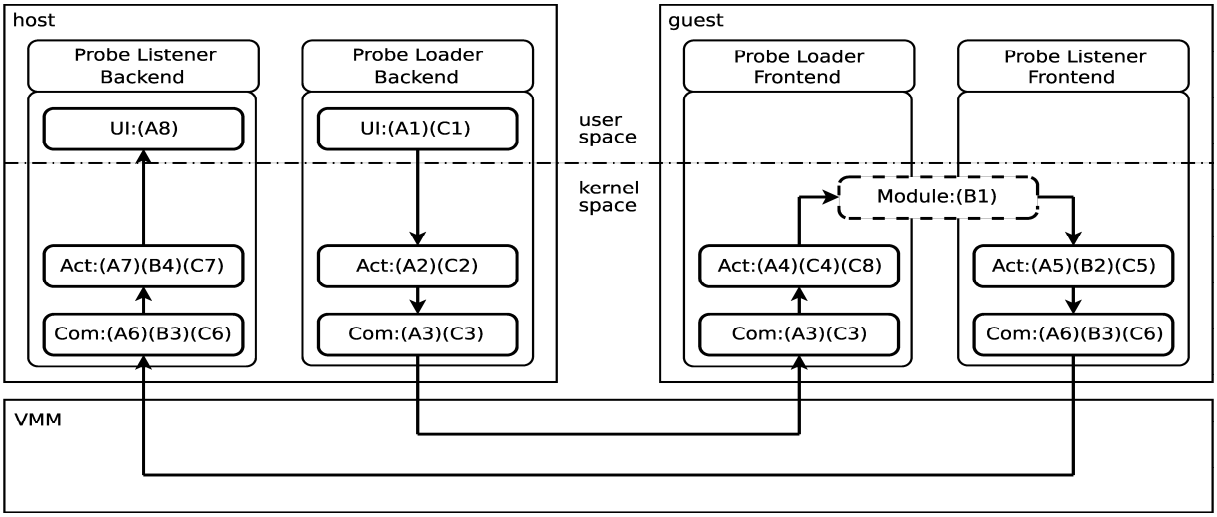

図 3　VESPER の処理の流れ

B4. ホスト OS 側インデックス情報の更新

ホスト OS 側 Probe Listener は, ゲスト OS から取得したインデックス情報に基づき, ホスト OS 内の relay バッファのインデックスを更新する.

4.2.3. モジュールのアンロード

モジュールのアンロード処理は，基本的に，4.2.1 に示したモジュールロードの処理とほぼ同じである．相違点は，実際にモジュールのアンロードをする部分が 2 段階に分けられている点である．これは，ゲスト OSのモジュールをアンロードする前に, ホストOSに作成してあるインタフェースを削除する必要があるためである．モジュールアンロードの処理を下記に示す.

C1. モジュールアンロードコマンドの実行
C2. モジュール情報の取得
C3. モジュールアンロード要求の送信/受信
C4. モジュールのアンロード 1
C5. relay バッファ共有の終了
C6. バッファ共有終了要求の送信/受信
C7. relay バッファの削除
C8. モジュールのアンロード 2

5. VESPER の実装

4章で説明したように, VESPERはProbe LoaderとProbe Listenerの二つから構成されている．現在，Xen向けのVESPERを開発しており，本章では，その実装について説明する.

5.1. Probe Loader

Probe Loader はゲスト OS のユーザ空間を利用せずに, ホスト OS からゲスト OS へプローブモジュールをロードする．これを実現するために，Probe Loader には 2 つの機能が必要である．1 つは，プローブモジュールをホストOSからゲストOSへ転送する機能であり, もう一つは，ゲスト OS においてユーザ空間を利用せずにモジュールをロードする機能である.

5.1.1. モジュール転送機能

プローブモジュールをホストOSからゲストOSへロードするために，VESPER は，まず，プローブモジュールをホストOSからゲストOSへ転送する必要がある. 通常，VMM はメモリ共有機能を利用した VM 間通信機能を備えており，Xen では grant table と I/O ring が提供されている．本機能を利用するために，Probe Loader は，ゲスト OS 上にフロントエンドドライバ，ホスト OS 上にバックエンドドライバを有する，Xenbus を利用した分割ドライバ方式で実装する.

VESPER では, バックエンドドライバにおいて grant table を利用して共有メモリを確保し, モジュールのイメージを書き込む. そして, バックエンドドライバは, I/O ring にメモリ共有要求を送信する. その後, I/O ring を介し，フロントエンドドライバは要求を受信し，該当する共有メモリを自身の仮想アドレス空間にマップする.

通常は，セキュリティ上の問題から，上記 VESPER の場合とは異なり，フロントエンドドライバがバックエンドドライバに対して，I/O ring を利用して要求を送信する．これは，仮に，バックエンドドライバがフロントエンドドライバに対して I/O ring を利用して要

求を送信した場合，フロントエンドドライバでは要求を満たすために，共有メモリに対して read/write を行なわなければならないが，これによりホスト OS の内容をゲスト OS が書き換えることになるからである．

しかしながら，VESPER では前述したとおり，ホスト OS がゲスト OS に対して，モジュールロードの要求をする必要がある．一方で，セキュリティの側面を考慮すると，VESPER においても，ゲスト OS がホスト OS に対して要求する形式をとるべきである．そこで，この本来の形式を守るために，VESPER ではゲスト OS のフロントエンドドライバが，最初にダミーの要求をホスト OS のバックエンドドライバに対して送信することにした．バックエンドドライバでは，リクエストを受信後，UI 層でモジュールロードのコマンドが実行されると，モジュールロードの要求を，最初のダミー要求への応答としてフロントエンドドライバへ送信する．このようにして，フロントエンドドライバは，サイズ等，モジュールに関する情報を取得し，モジュールイメージ用の共有メモリを grant table を利用して確保する．共有メモリを確保し終わると，フロントエンドドライバは，実際のモジュールロードの要求をバックエンドドライバに対して発行する．バックエンドドライバではモジュールロード要求を受信すると，*copy_from_user()* を利用して，モジュールイメージをユーザ空間から共有メモリへコピーし，フロントエンドドライバからの応答としてそのイメージを送信する．モジュールのロードが終了すると，フロントエンドドライバは，その結果を，新たなダミー要求としてバックエンドドライバへ送信する．再度，ホスト OS にてモジュールロードのコマンドが実行された場合には，本ダミー要求への応答として，バックエンドドライバはモジュールロードの要求をフロントエンドドライバに対して送信する．上記，フロントエンドドライバとバックエンドドライバ間の通信プロトコルを下記に示す．

1. フロントエンドドライバがダミー要求を送信
2. ホスト OS 上でアプリケーションが VESPER のモジュールロードコマンドを実行
3. バックエンドドライバがモジュールロード要求をダミー要求への応答として送信
4. フロントエンドドライバが共有メモリを確保した後，本当のモジュールロード要求を送信
5. バックエンドドライバはモジュールのイメージを共有メモリへコピー
6. フロントエンドドライバがモジュールをロード
7. フロントエンドドライバが，ロード結果を新たなダミー要求として送信
8. バックエンドドライバはモジュールのロード結果をアプリケーションへ通知
9. 以降，2〜8 の繰り返し

上記プロトコルを実現するために，VESPER では初期化時に，1 回ダミー要求を送信している．この方法で現在は本プロトコルを利用しなかった場合に発生する，セキュリティ上の問題を回避している．

5.1.2. モジュールロード機能

実際にシステムを運用していると，ユーザアプリケーションが制御不可能になり，アプリケーションは実行できないが，カーネルは動作可能な場合が時々発生する．このような状況において，仮に，ユーザ空間を利用せずにプローブモジュールがロードできれば，システムの不具合の原因を調査可能である．そのため，VESPER では，モジュールロード機能をゲスト OS のユーザ空間を利用せずに実装する．

現在，Linuxにおけるモジュールロードの中心的機能を提供する*load_module()* 等の関数では，モジュールのロードはユーザ空間から実行されるという前提で実装されているため，*copy_from_user()* を利用してモジュールのイメージを取得している．*copy_from_user()*内部ではアドレスを確認するために*access_ok()* を呼び出す．ここで，*access_ok()* は実際には呼び出し元の関数がユーザ空間に存在しているかを確認しているのではなく，アドレスが該当プロセスのアドレス空間内かどうかを確認している．そのため，VESPERでは，モジュールロード機能をゲストOS側Action層におけるカーネルスレッドとして実装した．このカーネルスレッドは*load_module()* を呼び出す*sys_init_module()* を利用する．モジュールのイメージは，5.1.1で説明したモジュール転送機能により，既にゲストOSのアドレス空間にマップされており，*load_module()* 内で呼び出される *copy_from_user()* は問題なく動作する．

しかしながら，sys_init_module()や load_module()はエクスポートされていないため，カーネルモジュールから直接呼び出すことができない．この問題に対し，VESPER では，ゲスト OS のシンボルテーブルからこれらのシンボルのアドレスをあらかじめ取得しておき，ホスト OS でモジュールをロードする際に，パラメタとして渡すことで解決している．

5.2. Probe Listener

Probe Listener はゲスト OS からプローブ時に記録したデータを取得し，ホスト OS 上のアプリケーションに対し，そのデータを提供する必要がある．プローブモジュールはプローブポイントにおけるゲスト OS の

情報を記録するために relayfs をする．ここで，relayfs は自身のバッファをページ単位で管理している.また，grant table による共有メモリも，ページ単位の管理を行なっている．そのため，relay バッファを grant table を用いて共有すれば，ホスト OS とゲスト OS の間で，データのコピーをする必要はない．また，relay バッファを共有することにより，その他に必要な制御機構も大幅に削減することが可能である.

上記のように，プローブしたデータを格納するバッファをホスト OS とゲスト OS で共有する場合,ホスト OS 上のアプリケーションが正しいデータにアクセスするためには,relay バッファの読み出しインデックスや padding 値等の管理情報を更新する必要がある.

このため，Probe Listener はバッファ共有機能と管理情報更新機能とから構成する．また，Probe Loader と同様に，分割ドライバ方式で実装する.

5.2.1. バッファ共有機能

前述したように，プローブモジュールは relay バッファにゲスト OS の情報を記録するため,Probe Listener はホスト OS とゲスト OS の間で，relay バッファを共有する必要がある．バッファの共有は，Probe Loader のモジュール転送機能と同様に，grant table を用いて実現する．また，共有バッファに関する情報に関しても，I/O ring を利用してゲスト OS からホスト OS に対して送信する.

ゲスト OS 側で relay バッファの共有処理が終了したことをホスト OS が受信すると，ホスト OS では，UI 層に対して，relayfs のインタフェースを利用してデータを提供するために，relayfs の管理用構造体を作成する．このとき，Probe Listener では，通常 relayfs 管理用構造体を作成するために利用する *relay_open()*を使用しない．これは，Probe Listener では，relay バッファ用のページを新たに確保する必要はなく,ゲスト OS と共有しているページを relay バッファの仮想アドレス領域にマップすればよいからである．このため，Probe Listener では relayfs 管理用構造体である rchan を手動で設定している．rchan の設定が終了すると，他の relayfs を利用しているサブシステムと同様に,sysfs 上にエントリが作成される．実際のパスはゲスト OS の ID やモジュールの名前を含み，次のようになる.
/sys/kernel/debug/vesper/guestID/modname/.

バッファの共有を終了する場合には，上記の共有開始手順を逆順に実行し，次のようになる．ホスト OS 上の relayfs 管理構造体 rchan を削除後，ゲスト OS にてバッファ共有の終了処理を実行する.

5.2.2. 管理情報更新機能

Probe Listener がバッファを共有すると，ホスト OS とゲスト OS で relayfs のバッファだけ共有している状態になる.このとき,実際にデータ記録が行なわれる，ゲスト OS 側ではバッファのインデックス等，relayfs 管理構造体のデータが更新されるが，ホスト OS 側では管理構造体は共有していないため，更新されない．そのため,ホスト OS 上の UI 層へ記録したデータを正しく提供するためには，ホスト OS 上の relayfs 管理構造体の情報を更新する必要がある．そこで，Probe Listener では，この問題を解決するために，I/O ring を利用して,必要な管理情報をゲスト OS からホスト OS へ転送する．ゲスト OS 側 Probe Listener は更新が必要な情報を含めたメッセージを作成し，I/O ring へ書き込む．ホスト OS 側 Probe Listener は I/O ring からメッセージを読み出し,relayfs の管理用構造体を更新する.

上記管理情報の更新は，ゲスト OS の情報が変化したら即座にホスト OS 上の管理情報へ反映させるのが理想的である．しかしながら，I/O ring を介したメッセージの送信は，割り込みを伴うため，管理情報の更新の度に実施するには，オーバヘッドが大きい．そのため，Probe Listener では，relay バッファのサブバッファの切り替えが発生した際に，管理用構造体を更新することにした.

6. インタフェース

6.1. アプリケーション用インタフェース

VESPER はアプリケーションに対し，ゲスト OS に対しモジュールをロード/アンロード用及びゲスト OS の情報取得用の簡潔なインタフェースを提供する．これらのインタフェースを下記に示す.

```
int virt_insmod( const int target_guest,
                 const char *modname,
                 const char *opt);

int virt_rmmod( const int target_guest,
                const char *modname,
                const long flags);
```

virt_insmod 及び virt_rmmod はモジュールのロード/アンロードに使用し，引数は target_guest を追加した以外は，insmod や rmmod と同様である．また，現在は,アプリケーションは直接 relayfs のインタフェースを利用してゲスト OS の情報を取得する仕組みとなっている.

6.2. モジュール用インタフェース

VESPER は，ゲスト OS で動作するプローブモジュールが，relay バッファをホスト OS と共有するためのインタフェースを提供している．また，relay バッファのサブバッファ切り替え時に呼び出される callback 関数も提供している．これらのインタフェースを下記に示す．

```
int relay_export_start( struct rchan :rchan,
                        const char *modname);

void relay_export_stop( const char *modname);

int virtrelay_subbuf_start_callback(
                        struct rchan_buf *buf,
                        void *subbuf,
                        void *prev_subbuf,
                        size_t prev_padding);
```

全てのプローブモジュールは relay_open()の後に relay_export_start() を，relay_close() の前に relay_export_end()を呼び出す必要がある．また，relay_open()を呼び出すときに必要な，コールバック関数登録用構造体である，struct rchan_callbacks の subbuf_start メンバに virtrelay_subbuf_start_callback() を設定する必要がある．

7. 評価

本章では Xen 上のゲストで Web サービスを提供するクラスタシステムを構成し, VESPER の評価を行なう．

7.1. 評価環境

評価環境としては，物理サーバを 2 台用意し，各サーバでゲスト OS を 2 つ起動する．物理サーバ 1 で動作しているゲスト OS を VM1-1，VM1-2 とし，物理サーバ 2 で動作しているゲスト OS を VM2-1，VM2-2 としたとき，VM1-1 と VM2-1 でクラスタ 1 を構成し，VM1-2 と VM2-2 でクラスタ 2 を構成する．クラスタ管理ソフトウェアとしては Linux-HA が提供している Heartbeat を利用する．ここで，クラスタ 1 は通常の Heartbeat で監視し，クラスタ 2 は Heartbeat を拡張し VESPER で監視を行なう．クラスタ 1 では Heartbeat で監視するが，監視方法としては，ゲスト OS を一つの資源として監視する．そのため，内部の http サーバの監視は対応していないため行なわない．監視対象の VM に異常が発生した場合には，Heartbeat 内の LRM(Local Resource Manager)に知らされ，サービス提供ノードの切り替えを行なう. 評価環境を図 4に示す．

7.2. heartbeat によるゲスト OS の監視方法

heartbeat では, 仮想化環境でクラスタを構築する際, ホスト OS 上で動作することで, ゲスト OS を一つの資源として制御・監視することが可能である．資源の制御としては次の 3 つがある．

- start
  資源を開始(起動)する．xen のゲスト OS の場合 xm コマンドを利用して起動する．

- stop
  資源を停止する．xen 環境では，start 同様 xm コマンドを利用している．

- monitor (status)
  資源の監視を行なう．xen では xm コマンドを利用して，ゲスト OS が存在していれば，問題なし，存在していなければ障害発生と判断している．

7.3. VESPER によるゲスト OS の監視方法

VESPER を用いてゲスト OS の情報をホスト OS で取得し，正常／異常終了の監視を行なった．プローブポイントとしては panic()関数，exit()関数をプローブした．

7.4. 結果と考察

上記の環境で実験を行なった結果，ゲスト OS にパニックが発生する等，ゲスト OS が停止してしまうような状況では，heartbeat による監視と VESPER による監視ではフェイルオーバまでの時間に差が生じなかった．これは，heartbeat が xm コマンドでゲスト OS を監視しているため，ゲスト OS が停止した場合にも即座に検知できる仕組みになっているからだと考えられる．一方，ゲスト OS は動作しているが，内部の http サーバに障害が発生して，アプリケーションが終了する場合等には，heartbeat による監視では障害を検出できず，フェイルオーバを実行することは無かった．しかし， VESPER で監視している際には障害を検知し，即座にフェイルオーバを開始し，サービスの停止時間を最小限にすることができた．

これらのことから，VESPER を利用することで，Heartbeat では検知不可能であった障害の検知に成功した．また，本実験では確認できなかったが，イベントドリブン方式の障害通知を採用していることから，障害発生から検知までの時間を縮め，サービスの停止時間を最小限にすることが可能であると予想している．

一方で，VESPER での監視を行う際に，2 つの問題が生じた. 1 つは, ゲスト OS の監視場所の決定である．

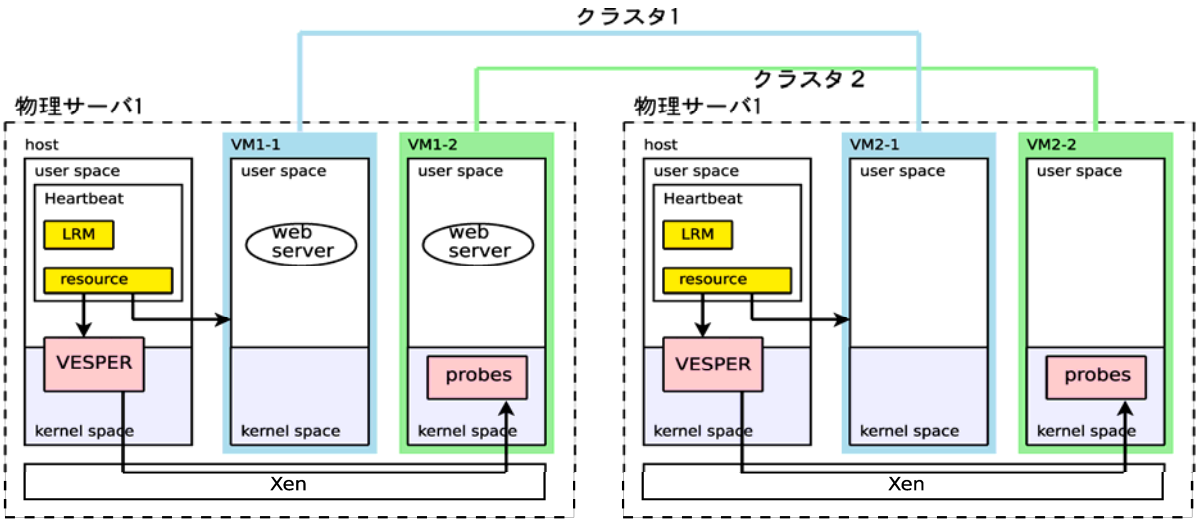

図 4　評価環境

VESPER では，提供しているサービスに応じて監視場所を変更することが可能だが，逆に監視場所を決定することが非常に難しく，熟練者による知識が必要となってしまう．2 つ目は，プローブのオーバヘッドである．Xenprobes に比べ，オーバヘッドは格段に小さくなったが，それでも，多数のプローブを挿入すると，本来のサービスに影響を与える可能性がある．1 つ目の問題と関係するが，やはり，適切な場所だけにプローブを挿入する必要がある．

8. まとめと今後の開発項目

本論文では，仮想化環境においてホスト OS 上でゲスト OS の情報を効率的に収集することが可能なフレームワークである VESPER を提案し，その設計や実装方法について説明した．そして，クラスタシステムにおけるフェイルオーバ遅延を改善したことを示し，VESPER の有効性を示すとともに，課題を述べた．

最後に，今後の開発項目について説明する．なお，8.6は直接的にはVESPERと関係ないが，今後，仮想化環境におけるクラスタシステムを推進していく際に，解決すべき課題と考えるため，ここに挙げておく．

これらの機能を開発し，仮想化環境でのより高信頼なクラスタシステムを実現していく予定である．

8.1. VESPER 基本機能の強化

現在の VESPER においても，障害検知の迅速化等を実現することはできた．しかしながら，ホスト OS のバッファ管理情報をサブバッファの切り替え時にしか更新しないことや，ゲスト OS のマイグレーションに対応指定いない等，未だ不足している機能があり，基本機能の強化が必要である．

8.2. プローブ補助機能

クラスタ管理ソフトウェアやその他のアプリケーションが VESPER を容易に利用できるように，プローブ補助機能を開発する．本機能により，各プローブポイントはその特徴により，複数のグループへと分類され管理される．また，本機能では，メモリ関連グループやネットワーク関連グループなどを，前もって定義しておき,ユーザは必要に応じてグループを選択れば，必要な部分だけの情報を採取可能になる．

8.3. SystemTap との連携

VESPERではプローブモジュールを必要とするが，通常モジュールの作成にはC言語の知識に加え，カーネルの知識も必要となり，開発が難しい．この問題に対し，簡単なスクリプトを書くことで，モジュールを作成可能なSystemTap[9]というツールが存在する．SystemTapによって，VESPERで利用するプローブモジュールが作成できれば，非常に利便性が向上すると考えられる．そのため，今後SystemTapの拡張へ取り組んでいく予定である．

8.4. virtio への対応

virtio は KVM 等いくつかの VMM において，ゲスト OS とホスト OS 間の通信機能を実現している．今後，VESPER は KVM へ対応する予定であるため，virtio への対応は必須である．

8.5. ホスト OS 障害の予兆検知

仮想化環境では，ホスト OS に障害が発生した場合には,そのホスト OS 上で動作しているゲスト OS は全て影響を受ける．このため，ホスト OS の障害の予兆

を検知することは非常に重要である．一方，仮想化環境においては，基本的にホスト OS のみが実ハードウェアにアクセスでき,本当の状態を知ることができる．以上のことから，VESPER では今後ホスト OS に対してもプローブを挿入し，実ハードウェアも含め監視を行なえるよう拡張する．

8.6. LRM における VM 内サービス監視のサポート

前述したとおり，仮想化環境において，クラスタシステムを構成する際に，クラスタ管理ソフトウェアをホスト OS 上で動作させ,ゲスト OS を資源として監視する方法が提案されている．この方法の場合，ゲスト OS 内で動作しているサービスの監視については,現在サポートされておらず，監視方法も含め今後検討する必要がある．

9. 参考文献

[1] The Xen virtual machine monitor, http://www.cl.cam.ac.uk/Research/SRG/netos/xen/

[2] KVM, http://kvm.qumranet.com/

[3] Heartbeat, http://linux-ha.org

[4] Alan Robertson, Linux-HA Heartbeat Design, In Proceedings of the 4th International Linux Showcase and Conference, 2000.

[5] Alan Robertson, World Class HA with Linux-HA, 7th The Linux Foudation Japan Symposium, http://www.linux-foundation.jp/uploads/seminar20080312/Linux-HA-Japan2008.pdf

[6] Ananth N. Mavimalayanahalli et al., Probing the Guts of Kprobes, In Proceedings of Ottawa Linux Symposium, 2006.

[7] Nguyen Anh Quynh, Xenprobes, A Lightweight User-space Probing Framework for Xen Virtual Machine, In USENIX Annual Technical Conference Proceedings, 2007.

[8] Nitin A. Kamble et al., Evolusion in Kernel Debugging using Hardware Virtualization With Xen, In Proceedings of Ottawa Linux Symposium, 2006.

[9] SystemTap, http://sourceware.org/systemtap/

[10] Rusty Russel, lguest:Implementating the little Linux hypervisor, In Proceedings of Ottawa Linux Symposium, 2007.

[11] VESPER, http://vesper.sourceforge.net/