VESPER (Virtual Embraced Space ProbER)

Sungho Kim
Hitachi, Ltd., Systems Development Lab
sungho.kim.zd@hitachi.com

Satoru Moriya
Hitachi, Ltd., Systems Development Lab
satoru.moriya.br@hitachi.com

Satoshi Oshima
Hitachi, Ltd., Systems Development Lab
satoshi.oshima.fk@hitachi.com

Abstract

This paper describes VESPER (Virtual Embraced Space ProbER), the framework that gathers guest information effectively in a virtualized environment. VESPER is designed to provide evaluation criteria for system reliability and serviceability, used in decision-making for system switching or migration by a cluster manager. In general, the cluster manager exchanges messages between underlying nodes to check their health through the network. In this way, however, the manager can not discover a fault immediately and get detailed information about faulty nodes. VESPER injects Kprobes into the guest to gather the guest's detailed information. By communicating with the guest in kprobes through the VMM infrastructure, VESPER can provide the manager with prompt fault information much more quickly.

In this paper, we explain how VESPER injects Kprobes into a guest and clarify the benefits by showing a use case on Xen. As VESPER is not strongly coupled to a specific VMM, we also show its portability to KVM and lguest.

1 Introduction

Recently, the trend of applying virtualization technology to enterprise server systems is getting much more noticeable, such as in server consolidation. The technology enables a single server to execute multiple tasks which would usually run on multiple physical machines. From that point, applying virtualization technology to cluster computing is very attractive technology and worth considering as well, in terms of efficient resource utilization and system dependability. Furthermore, use of virtualized environments for cluster systems allows us to make improvements in several areas of general clustering technology—especially, fail-over response latency in high-availability clusters.

Even in a virtualized environment, a cluster manager such as Heartbeat [1][2] software delivers messages between underlying nodes to check their health through network periodically (known as heartbeat). If any node fails to reply in a certain time, the manager will assign the service which the faulty node was providing to another node. This amount of time before switching to another node is called the deadtime, key to ascertaining node death in Heartbeat. With this approach, however, the manager can not immediately determine faults, nor get detailed information about faulty nodes; this results in fail-over response latency. So we have focused on improving upon this latency, and on the failure analysis the manager should facilitate in clustering virtual machines, by considering features of virtualization technology.

One solution to the latency and failure analysis issues is to add dynamic probing technology into virtual machines. This allows the cluster manager to have:

- Dynamic probe insertion to guarantee service availability while inserting a probe.
- Arbitrary probe insertion to any process address to hold its versatile probing capability.
- Prompt notification when corresponding events happen around a probe.

Utilizing this featured probe technology to examine the health of a virtual cluster member machine could lead to faster and more efficient evaluation criteria for system switching or migration than a simple, periodic message delivery mechanism.

Speaking of the probe technology, Kprobes [3] are available in the Linux kernel community. Kprobes can insert a probe dynamically at a given address in the running kernel of a targeted virtual machine. Its execution of the probed instruction in the kernel is capable of dealing with the detailed information on the targeted virtual machine in an event-driven fashion.

From a system management point of view, however, one privileged system running the cluster manager (called ***host*** hereafter) should obtain all probed data from the targeted virtual machine (called ***guest*** hereafter). So, there needs a mechanism to insert probes from the host into the guests to improve the manageability of the manager, which Kprobes does not take into consideration.

Xenprobes [4] has already addressed this topic. Xenprobes is newly devised for probing virtual machines, but adopts Kprobes's concept in inserting breakpoints where one needs a probe. However, Xenprobes needs the help of a VMM-like furnished debugging mechanism [5] and should stop the guest to insert the probes. Moreover, every breakpoint causes VMM to give execution control to the host, especially in Xen [6] technology. These could be significant problems in service availability.

In this paper, therefore, we propose the framework named VESPER (Virtual Embraced Space Prober) which gathers guest information effectively in a virtualized environment, taking advantage of the full features of Kprobes, adopted for its probing component. In contrast to Xenprobes, VESPER never gets involved with probe handlers; this acts to avoid unnecessary probing overhead and to improve service availability. VESPER simply transfers Kprobes generated in the host to the targeted guest. The transferred Kprobes do all the necessary probing work themselves in the guest, and then VESPER simultaneously obtains the result of Kprobes through shared buffers (such as relayfs) built across the host and guest.

We will describe how VESPER injects the probes into guest and provides the solution to fail-over response latency with failure analysis in a virtualized environment by showing a use case with Xen, in the following sections of this paper.

Section 2 briefly describes design requirements in VESPER; Section 3 presents the architecture of VESPER. Section 4 presents implementation details of VESPER, while Section 5 shows some benefits of VESPER for simple web services. Finally, we conclude this paper in Section 6.

2 Design overview

In designing VESPER, we took special interest in simplicity of implementation and robustness against disasters in userland. We thus set up some design requirements on VESPER as follows, to reflect our concepts.

1. No modifications on host or guest kernels. Lightweight implementation as a kernel module could assure better usability and availability of VESPER in systems due to dynamic loading and unloading features of kernel modules.

2. Only the host can insert probes into the guest, and guest itself loads them using guest kernel space only. As mentioned in the previous section, the cluster manager running on the host might as well control probes into the guest from a management point of view. In addition, the guest itself loads the probes inserted by the host into its kernel space dynamically to keep its serviceability. Furthermore, if some disaster should mangle user space but not kernel space in the guest, VEPSER should still be available to find out what the problem is.

3. All probed data from the guest is sent to host as quickly as possible. To receive prompt alerts via probed data is the main purpose of VESPER, and thus to improve fail-over response latency.

To satisfy the requirements above, VESPER thinks of Xen and KVM mainly as target VMMs because of the popularity of Xen and KVM [7] in the OSS (Open Source Software) community in this writing. Firstly, to address Requirement *1* above, VESPER is, by preference, developed as a virtual device driver. Requirements *2* and *3* dictate that VESPER communicate with host and guest. As a matter of fact, Xen and KVM technology provide a well-defined device driver model, split device driver, and communication infrastructure between host and guest. Xenbus is infrastructure specific to Xen technology, whereas `virtio` is applicable to Xen and KVM as well. Especially, virtio is available in 2.6.24 Linux kernel. VESPER uses hierarchical layers in its software structure to accommodate various infrastructures, making it more available and portable.

The layer dependent on a certain infrastructure prepares the set of functions and data items for the use of the layers independent of the VMM architecture. Details on that structure and implementation will be discussed in the upcoming sections.

3 Architecture and Semantics of VESPER

For probing the guest, VESPER uses Kprobes to hook into the guest Linux kernel, and uses `relayfs` to record probed data in the probe handler of Kprobes. In this section, we take a brief look at the VESPER architecture featuring a 3-layered structure, then we present its semantics.

3.1 VESPER Architecture

As mentioned before, VESPER uses Kprobes to hook into the guest kernel. However, because Kprobes is the probing interface for the local system, it can not implant probes into the remote system directly. Besides, in the typical use case of Kprobes, the probe handler in Kprobes comes in the form of a kernel module. Therefore, in using Kprobes to hook into the guest kernel, VESPER should be able to load probing kernel modules, on which the handlers to probe are implemented, from host to guest. This loading capability of VESPER is implemented as split drivers and named *Probe Loader*.

In VESPER, the probing modules use relay buffers to record data in the probe handlers. At this point, probing modules are in the guest; thus, VESPER needs to transfer the buffer data from the guest to the host. This relayed data transfer capability is also implemented as split drivers and named *Probe Listener*.

Figure 1 is the block diagram of the VESPER component.

As just described, VESPER contains two pairs of split drivers. These drivers are implemented for each VMM because they strongly depend on the underlying VMM. So, we divide VESPER into three layers (shown in Figure 2): UI Layer, Action Layer, and Communication Layer, in order to localize VMM-dependent code. This structure lets VESPER run on Xen and KVM by replacing only the VMM architecture-dependent layer—the Communication Layer.

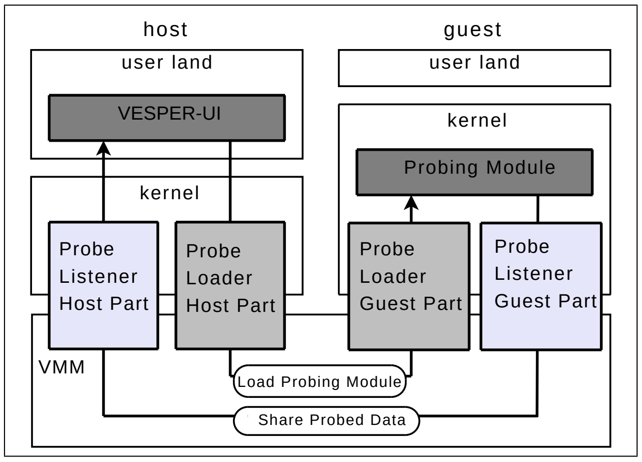

Figure 1: Architecture of VESPER

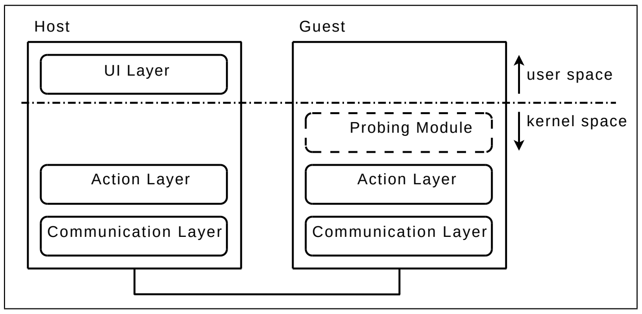

Figure 2: Layer of VESPER

1. UI Layer

   This is the interface layer between VESPER and user applications which manage the guest in the host. This layer provides interfaces for loading and unloading probing modules to/from the guest and accessing probed data recorded in the guest. These are presented in detail in Section 5.

2. Action Layer

   This is the worker layer which processes requests from the UI and Communication Layer. Concretely speaking, in this layer VESPER does actual work for load/unload probing modules to/from guest and sharing probed data between host and guest.

3. Communication Layer

   This is the layer which provides communication channels between host and guest. It strongly depends on VMM architecture. By implementing this layer with respect to each VMM, VESPER confines the differences between VMM environments to this layer.

3.2 VESPER Semantics

In order to implant probes into guests, VESPER must load probing modules from the host to the guest. And then, VESPER sends probed data from the guest to the host.

Figure 3 illustrates the semantics overview of VESPER.

3.2.1 Module Loading

The first step to probe the guest kernel is to load the probing module onto the guest.

0. Make Module

First of all, one should make a probing module which uses Kprobes and relayfs.

A1. Module Load Command

Execute the probing module insertion via interfaces provided by the probe loader. One can also specify module parameters, if needed.

A2. Obtain Module Information

On the host-side Action Layer of the probe loader, from user space, VESPER obtains the module information to insert such as the module's name, its size, and its address with others related to module parameters, if any.

A3. Send/Receive Request

Through the interface provided by the VMM, the probe loader transfers the probing module's information between the host and guest.

A4. Load Module

In the Action Layer, the probe loader on the guest side loads the module without userspace help.

A5. Share Relay Buffer

In the Action Layer, the guest's probe listener gets relayfs buffer information such as the read index, buffer ID, etc., from the probe modules; it then exports the buffer to the host.

A6. Send/Receive Buffer Information

Communication layer of guest probe listener transfer the shared buffer information to host via VMM interface, and then, host probe listener receives it.

A7. Setup Relayfs Structure

The Action Layer of the host's probe listener builds the relayfs structure based on the information received from the guest.

A8. Analyze/Offer Probed Data

One can read probed data through the UI Layer of the probe listener.

3.2.2 Probing

After finishing the procedures in Section 3.2.1, the host and guest probe listeners share relay buffers of the probing module. Consequently, it is not necessary to transfer all the recorded data from guest to host, but it is necessary to transfer index information about shared relay buffers, where the data is, to get the start index for the actual access to relay buffers by host.

B1. Gather Guest Kernel Data

Once loaded, the probing module puts data into the relay buffer in the probe handler.

B2. Get Index Information

When change occurs in the relay sub-buffer, the action layer of the guest probe listener gets the index information and creates a message to notify host.

B3. Send/Receive Message

Through the communication layer, the host probe listener is notified of the index data from the guest.

B4. Update Index

The action layer function of the host probe listener updates the index of relayfs in the host, based on the received message.

3.2.3 Module Unloading

Basically, unloading a probing module below is similar to loading a module. The significant difference is that it takes two steps in the unloading module process, because before removing the relay buffer in the handler in the guest, the exported user interface for the buffer in host should be dropped.

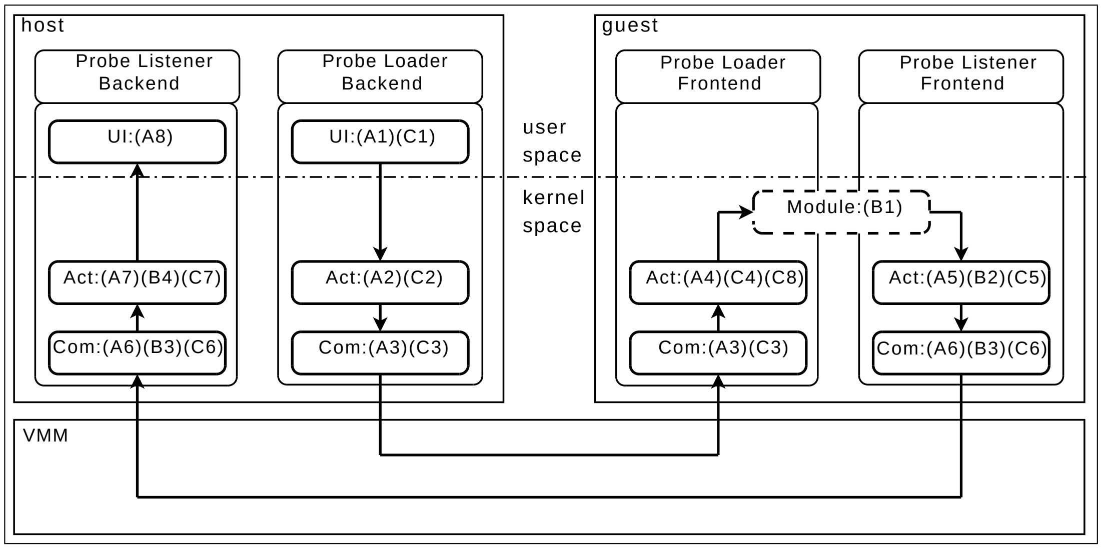

Figure 3: Process Flow of VESPER.

C1. Module Unload Command

C2. Obtain Module Information

C3. Send/Receive Request

C4. Unload Module (step1)

C5. Stop Sharing Relay Buffer

C6. Send/Receive Buffer Information

C7. Destroy Relayfs Buffer

C8. Unload Module (step2)

In the next section, we describe the detailed implementation of the probe loader and listener.

4 Implementation of VESPER

As previously described, VESPER consists of two components named probe loader and probe listener. Each component is split into three parts, UI Layer, Action Layer, and Communication Layer, to confine the dependency on the underlying VMM. In this section, we present the implementation of VESPER from the viewpoint of the Communication Layer on Xen.

4.1 Probe Loader

Probe Loader loads probing modules from host to guest without using the guest's user space. To make this concept real, probe loader needs to perform two functions. One is to transfer the probing module from the host to guest, and the other is to load the module in the guest without involving user space.

4.1.1 Module Transfer Function

In order to implant the probing module from the host into the guest memory space, VESPER needs to be able to transfer the module image somehow. Generally, VMM provides I/O infrastructure between host and guest using a shared memory mechanism. In the Xen environment, `grant table` and `I/O ring` are provided for that. To use them for transferring the module, probe loader is implemented as split drivers—frontend driver in the guest, and backend driver in the host, on Xenbus.

Backend driver allocates shared memory using `grant table` and write the module image into it. And then, the driver pushes a request onto `I/O ring` for the frontend driver. After receiving the request through `I/O ring`, frontend driver maps the shared memory to its virtual memory space.

For security reasons, it is normally the frontend driver that requests I/O from the backend driver. If, instead, the backend driver issued I/O requests to the frontend driver, because frontend driver should have read/write permissions on the host to accomplish the requests, this would result in the guest being able change the contents of the host's page.

In VESPER, however, (as mentioned above) the host must issue requests to the guest for loading the module. From a security perspective, we must ensure that VESPER's communication protocol between host and guest follows this pattern—that is, that the frontend driver requests, and the backend driver responds. To keep this pattern, the frontend driver first issues a dummy request to backend driver. Then, the backend driver issues the module loading request as a response to the dummy request in the protocol, when the UI layer in host triggers. At this phase, the frontend driver has received all the information about the module to insert and then allocates a `grant table` for the module image. After `grant table` allocation, the frontend driver issues the real module loading request. With that request, the backend driver does `copy_from_user` to the `grant table` with respect to the module for the response. Finally, the result of loading the module is sent as a new dummy request from frontend driver. When next request is triggered from the host, the backend driver only issues a new request as the response to the last dummy request for the next process. The processes above are involved for every module insertion request in the host. The details of the protocol follow.

1. Frontend driver issues a dummy request.
2. Application triggers the module loading into guest.
3. Backend driver issues the module loading request as a response to the dummy request.
4. Frontend driver issues a real request with `grant table` allocated.
5. Backend driver copies the module into the `grant table`.
6. Frontend driver loads the module.
7. Frontend driver issues a new dummy request with the last loading module result.
8. Backend driver hands in the result of the module loading to the application.
9. Repeat Steps 2 through 8 for every request from the application.

From the benefit of the protocol, we sacrifice only one dummy request incurred in the initialization phase of the drivers, which is very trivial compared to the security hole.

4.1.2 Module Load Function

Sometimes there is the situation that a user program goes out of control. In such cases, the user program cannot be executed, but a kernel program can. If it is possible to load the module without user space help, one is able to analyze system faults even in the case above. Thus, in VESPER, the module load function is implemented without using a user-space program.

Currently the core system of the module loader in Linux, such as `load_module`, calls `copy_from_user` to get module images because it assumes that module loading is executed in user space. Inside `copy_from_user`, `access_ok` is called to verify its memory address. However, it never checks whether the calling function is executed in user space; it only checks whether the address limit is in its process address space. Hence, we implemented the module load function as a kernel thread in the Action Layer of the guest probe loader. This kernel thread calls `sys_init_module`, which calls `load_module`. Because the loaded module is already copied from the host via the module transfer function described above, `copy_from_user` works properly.

Nevertheless, it is impossible to invoke `sys_init_module` and `load_module` from external kernel modules, because they are not exported by `EXPORT_SYMBOL`. To address this problem, in VESPER, we get the address of these symbols from the guest kernel symbol table, and then pass them as parameters to the user interface probe loader provides for loading.

4.2 Probe Listener

The Probe Listener should retrieve the probed data from the guest and make it available to user-space applications in the host. Probing modules use relayfs to record the probed data which describes the behavior of the guest kernel around the probe points.

In fact, relayfs tends to allocate its buffers by pages, and `grant table` is also a page-oriented mechanism. So, provided that the buffers controlled by relayfs are allocated by `grant table` in the guest, a copy process is definitely unnecessary between host and guest to share the data, because `grant table` is transparent to both host and guest. A design decision on using `grant table` as relayfs buffers could also eliminate the need of other control mechanisms onto buffers than relayfs rchan.

In the case of sharing the probed data in the buffers, VESPER should also share and update some information about the buffers, such as the read index and padding value for relayfs in the host, to access the buffers.

As a result, Probe Listener consists of two components, which are a buffer share function and an index update function, and it is implemented as split drivers, just like Probe Loader.

4.2.1 Buffer Share Function

As mentioned before, because the probing module records probed data into relay buffers, Probe Listener shares them between host and guest. Sharing the buffers is implemented by using `grant table` like the module transfer function in Probe Loader. Similarly, information about which buffers need to be shared is provided from guest to host by using `I/O ring`.

Once the relay buffers are exported by the guest and their information is received by the host, the relayfs structure is built on the host to provide probed data in the buffers to user-space applications. At this time, Probe Listener does not call `relay_open` to create a `rchan` structure, which is a control structure of relayfs. This is because Probe Loader does not need to newly allocate the pages for the relay buffers, but should just map the pages exported by the guest. Therefore, Probe Listener sets up the `rchan` structure manually. After rchan is set up, the interface to read this relay buffers is created on `/sys/kernel/debug/vesper/domid/modname/` like other subsystems which use relayfs. User applications can read this interface directly, or use APIs abstracted by VESPER.

Finally, to stop sharing the buffer, the probe listener executes the above process in reverse. Removing the relay structure is done at first, and then exporting relay buffers is stopped.

4.2.2 Index Update Function

When the probe listener shares the relay buffers between host and guest, it must synchronize some buffer information such as the read index between both rchan structures. If it does not, the user application on the host cannot read the probed data correctly. Probe Listener uses `I/O ring` for the information transfer. The guest's probe listener creates a message including the information, pushes it to `I/O ring`, and then notifies the host's probe listener. The host's probe listener gets the message from `I/O ring` and updates its own relay buffer information with the message.

Ideally, probe listener should update that information immediately whenever the guest rchan is changed. However, message passing by `I/O ring` is too expensive to update each time due to the intervention of the interrupt mechanism to notify host of the existence of pending messages. Hence, Probe Listener updates the buffer information when switching to sub-buffer occurs. In doing so, probe listener updates the buffer control information, and the user application can get the latest data probed from the guest.

5 VESPER Interface

This section describes the VESPER Interface.

5.1 The VESPER User API

VESPER provides user applications with simple interfaces to insert probing modules to target guests and to obtain probed data from the guests in Figure 4. Arguments of `virt_insmod` and `virt_rmmod` are simply the same as `insmod` and `rmmod`, Linux user commands to handle kernel modules, except that they target the guest. In addition, `virt_is_alive` is available for the application to check if some error occurs around the probed point.

5.2 The VESPER Module API

VESPER defines an API to export relay buffers of the probing module to the host. Additionally, VESPER provides a callback function for use when the sub-buffer of

```
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);

bool virt_is_alive(
    const int target_guest,
    const char *modname);
```

Figure 4: Prototype of the VESPER user API

the relay is changed. All probing modules should call the exported function after `relay_open`, and the stop export function before `relay_close`. The callback function also is set up to `subbuf_start`, the member of `struct rchan_callbacks`. Figure 5 shows the prototype of the module API.

```
int relay_export_start(
    struct rchan *rchan,
    const chaq *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);
```

Figure 5: Prototype of the VESPER module API

6 Evaluation

In this section, we will examine the benefits of VESPER with a webserver running on a Xen-based guest, and will explain how VESPER works with Heartbeat to eliminate the latency using the test case we plan to build.

6.1 Test environment

For this experiment, we plan to prepare two physical machines. We set up two guests as resources managed by the LRM (Local Resource Manager) of Heartbeat on each physical machine. In fact, it is a controversial issue on how a guest is treated in the cluster, as one of the cluster nodes, or as a resource like IP. However, we will treat guests as a resource to avoid any complexity of management caused by difficulty in identifying host and guest from the all nodes, in case a guest were treated as a node. On each physical machine, the webserver is on the one guest, we say VM1; it is actively performing web service. On the other hand, the webserver in the other guest, we say VM2, is inactive. Figure 6 depicts the details.

When something wrong happens to VM1, Heartbeat lets VM2 take over all of roles which VM1 was performing. However, one physical machine, P1, has Heartbeat's LRM without involvement of VESPER to show how Heartbeat works in the usual way. However, the other physical machine, P2, has LRM cooperating with VESPER. Here, we treat the guest as a resource so that we plan to add virtual machine resource plugin conforming to OCF (Open Cluster Framework) to LRM to make Heartbeat and LRM recognize a virtual machine as a resource. The plugin has interfaces for start, stop, and monitor (a.k.a. status) the resource. In implementing the resource plugin, the monitor interface of the plugin will invoke VESPER for LRM on P2 only.

6.2 Implementation of virtual machine resource

In Heartbeat, CRM (Cluster Resource Manager) coordinates what resources ought to run where, or which status they are running, working with the resource configuration it maintains. And it commands LRM to achieve all the things. LRM then searches for proper resource to handle by way of the PILS subsystem of Heartbeat. LRM calls exported interfaces by the resource to execute CRM requests, start/stop/monitor, etc. We newly define a virtual machine resource, and it exports a start/stop/monitor interface implemented as follows.

- *start*. The resource invokes the `xm` command (Xen tool) to start a specific guest.
- *stop*. The resource also invokes `xm` (Xen tool) to stop a specific guest.

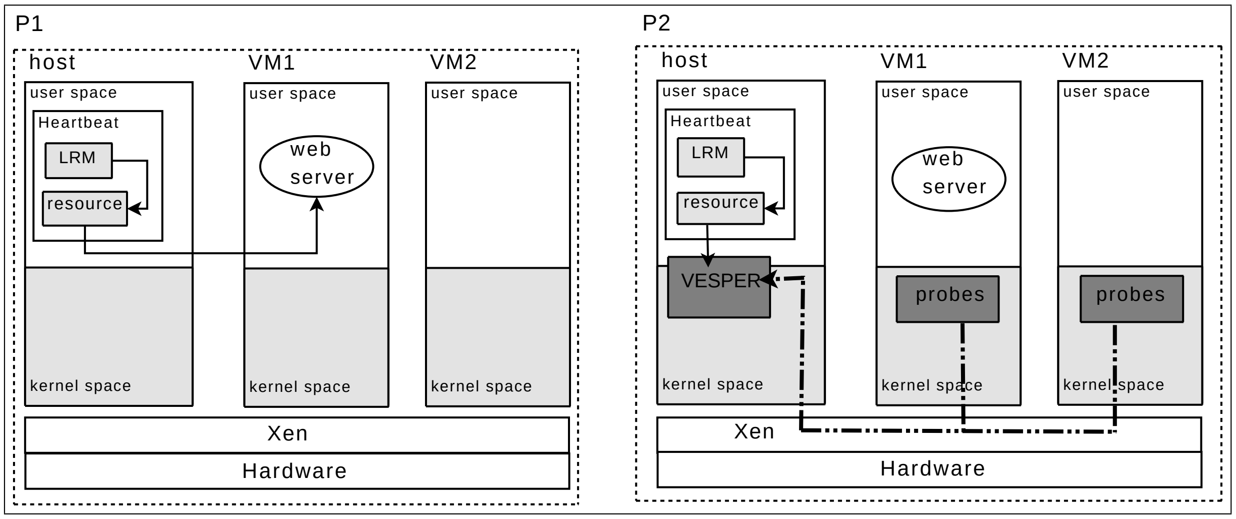

Figure 6: Test environment to demonstrate the usability of VESPER.

- *monitor*. The resource invokes the xm command (Xen tool) to get the status of a specific guest as well as monitor procedure, prepared for this test, to get status of the services running on that guest. What is more, the resource on P2 invokes the VESPER interface and does a logical OR on the results of the above two for the return value, only in case of P2.

6.3 Expected result and discussion

For simplicity, we insert Kprobes around the panic path of the guest kernel via VESPER. Kprobes thus inserted will put clues like the callstack to the panic on the relayfs when the panic occurs. Then, we cause a panic intentionally on VM1 and measure the recovery time on P1 and P2. We can easily expect that P2 will recognize what happened to VM1 of P2 as soon as VM1 panics, because of the prompt notification done by VESPER. Obviously service recovery time is supposed to be dramatically improved by as much as deadtime we set for Heartbeat. Moreover, one can find out the reason why VM1 on P2 panicked through the probed data on the relayfs later on.

Through the experiment, we could verify better performance on response latency and usability of failure analysis provided by VESPER.

However, special care should be taken with two issues regarding probes. One is what probe points are suitable for proper monitoring. If the targeted, necessary probe points miss, no more improvement over usual Heartbeat can be expected. Actually, the problem on where probe points should be inserted seems very tricky to handle, because highly experienced developers or system administrators on kernel context and applications running on the server are required to select optimal probe points. The other is about overhead produced by the execution of probes. One should adjust the overhead according to the required service performance. Both issues exclude each other. More probes inserted to hit fine-grained events cause more overhead in probing, obviously. Some mechanism to help one select optimal probes could be needed. Some suggestions for these issues will be mentioned as future works of VESPER in the next section.

7 Conclusion and future works

In this paper, we proposed VESPER as a framework to insert probes in virtualized environments and discussed what topics VESPER can solve in clustering computing. After that, we described the design and the implementation of VESPER. Then we suggested a test bed to show the performance improvement on fail-over response latency and failure analysis. Finally we discussed some considerations on places and overhead of probing. To address these considerations, we have some plans about future VEPSER developments.

7.1 Probing aid subsystem

For the ease use of cluster manager or other applications, we plan to develop a probing aid subsystem. Probing points could be classified into several groups based on their functionality. The subsystem thus can pre-define several groups of probe points and abstract them to its clients or application—like memory group, network group, block-io group, etc. The clients just select one of groups, and the subsystem will generate all needed probes relayed to VESPER. Also, fine-grained selection from several groups will be supported by the subsystem.

7.2 SystemTap Enhancement

We have a plan to integrate the feature of VESPERs for virtualization into the SystemTap [8] for its versatile usage in the native kernel as well as the virtualized kernel.

7.3 Virtio support and evaluation on KVM and lguest

Virtio is likely to promise one standard solution to host and guest communication infrastructure on various VMMs. VESPER should support virtio and be evaluated on KVM and lguest [9] to verify its portability and usability.

The next two are not related directly to probing technique but are worth examining as enhancements of functionality of virtual machine resource facilities in clusters in terms of virtualization technology.

7.4 Virtual machine resource support for LRM

Arguably, there is a question on how services running on virtual machines should be treated if a virtual machine is treated as a resource. Should the services be handled at the same level as the virtual machine itself in LRM? When the services go fail-down, only the failed service must be moved to another virtual machine on the same physical machine—better than virtual machine itself should be switched to another virtual machine on a different physical machine. The next version of interface VESPER exports will suggest the solution to that.

7.5 Precaution capability on collapse of the host

If the host collapsed, all services running on the guests would be lost. It is obviously a big problem. Therefore, VESPER should probe the host simultaneously to check whether the host is in good condition. When VESPER catches a sign of the host's collapse, the cluster manager notified by VESPER could take necessary action, such as live migration to other host.

8 Acknowledgments

We would like to thank Yumiko Sugita and our colleagues for reviewing this paper.

9 Legal Statements

References

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

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

[3] Ananth N. Mavinakayanahalli et al., "Probing the Guts of Kprobes," In *Proceedings of the Linux Symposium*, Ottawa, Canada, 2006.

[4] Nguyen A. Quynh et al., "Xenprobes, A Lightweight User-space Probing Framework for Xen Virtual Machine," In *USENIX Annual Technical Conference Proceedings*, 2007.

[5] Nitin A. Kamble et al., "Evolution in Kernel Debugging using Hardware Virtualization With Xen," In *Proceedings of the Linux Symposium*, Ottawa, Canada, 2006.

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

[7] KVM, `http://kvm.qumranet.com/`.

[8] SystemTap, `http://sourceware.org/systemtap/`.

[9] Rusty Russell, "`lguest`: Implementing the little Linux hypervisor," In *Proceedings of the Linux Symposium*, Ottawa, Canada, 2007.

[10] VESPER, `http://vesper.sourceforge.net/`.

Proceedings of the Linux Symposium

Volume One

July 23rd–26th, 2008
Ottawa, Ontario
Canada