Tracing the HA Cluster of Guests with 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

Recently, many tracing infrastructures, like kprobes, tracepoints, ftrace, etc. have been merged into the mainline kernel. They seem useful to tell what is going on inside the kernel in the physical machine. So, it is natural that we tend to question if can we use them to trace virtual machines.

In this paper, we introduce VESPER, the framework to trace guest kernel states from the host utilizing in-tree tracing stuff in the just same manner as host kernel tracing. In particular, the mechanism of injecting probes to guest and splicing guest tracing reports onto host to alleviate data copy overhead will be focused upon. To verify the efficiency of VESPER, we take HA cluster with guests on in-tree hypervisors, KVM, for test cases. By combining tracepoints with kprobes to monitor guests, VESPER shows the improvement on fail-over response latency caused by application-bound as well as system-wide failure, against conventional heartbeat.

1 Introduction

Tracing issues have recently been focused on Linux kernel community. Kernel Markers and trace points have already merged into the mainline kernel and various Ftrace engines have been introduced at the LKML [1]. Those tracing facilities can provide lightweight mechanism to understand the behavior of the kernel compared to dynamic tracing facilities such as Kprobes which utilize `breakpoint`. It seems to be reasonable to use the tracing facilities to debug the kernel. However, we have been thinking about other use cases for the facilities than the debugging. Our suggestion is applying the kernel tracing to the decision maker of fail-over or migration in the clustering of virtual machines in enterprise server systems requiring efficient resource utilization and system dependability. In a certain system, fail-over response latency is the bottleneck for the high availability of service, which comes from the polling-way of heartbeating between cluster nodes. Even in a virtualized environment, a cluster manager such as Heartbeat [2][3] software delivers messages between cluster 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 cannot immediately determine faults, nor get detailed information about faulty nodes; this results in fail-over response latency. But what if the tracing technology is applied to the heartbeat? This allows the cluster manager to have:

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

Utilizing this featured 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 de-

livery mechanism. Therefore, we have proposed VESPER (Virtual Embraced Space Prober)[4] which gathers guest information effectively in a virtualized environment, taking advantage of the full features of the tracing facilities. VESPER simply transfers probes generated in the host to the targeted guest. The transferred probes do all the necessary probing work themselves in the guest, and then VESPER simultaneously obtains the result of probes through shared memory built across the host and guest. However, VESPER was only available for the paravirtualized guests on Xen [5] because it utilized the Xenbus for the communication between the host and the guest. To make VESPER more available for the various virtualization technology we adopt Virtio [6] proposed as the standard of virtual I/O mechanism in the Linux community. In this paper, we will describe how to port VESPER to Virtio and how to inject the probes into hardware-virtualized guest on KVM [7], which is a in-tree hypervisor supporting Virtio.

Section 2 briefly describes the mechanism of QEMU [8] and KVM using Virtio to implement virtual I/O devices. Section 3 presents the brief description of VESPER architecture and implementation of VESPER as a virtual I/O device in QEMU, while Section 4 discusses trace issues using VESPER. Finally, we conclude this paper in Section 5.

2 Overview of Virtio in QEMU

Before getting into the porting issue, we briefly describe how QEMU and KVM works for virtual I/O devices (called virtio devised hereafter) such as `virtio_blk` and `virtio_net`.

In Virtio, all virtio devices are treated as pci devices under `virtio_pci` in the guest. So, virtio devices need to invoke pci emulator in QEMU. Fortunately, `virtio_init_pci` in `qemu/hw/virtio.c` in QEMU does necessary works for them. In the process of initializing virtual machine hardware, virtio devices invoke `virtio_init_pci` to register them as pci devices in QEMU. Then `virtio_init_pci` allocates VirtQueue, the control block for the shared memory between the host and the guest in Virtio, for the devices and invokes `pci_register_io_region` to store the configuration information of the devices just like native pci devices. The different thing from the native PCI devices is to register callback functions to each configuration area other than specific data. The callback functions are the key components to share virtio devices between the host and the guest. Especially in KVM, I/O accessing of the guest to those io regions occurs `VM_EXIT` to switch cpu context from the guest to the host and QEMU at last. As a matter of fact, `virtqueue_ops.kick` of `virtio_pci`, the communication method in Virtio, in the guest invokes `iowrite16`. In the cpu context switch between the host and guest in KVM, callback functions registered in the specific area invoke service routines for each devices. Accessing other features data of the device follows the same processing below.

1. Register Device in QEMU.

   Register the device in the PCI bus in QEMU.

2. Register IO Region in QEMU.

   Map the allocated IO Region with proper callbacks.

3. Register VirtQueue in QEMU.

   Allocate VirtQueue to handle the shared memory prepared by the guest in QEMU.

4. Retrieve Device Configuration.

   Registered virtio device driver probed in the guest, the driver executes I/O access to retrieve the device features.

5. Set `virtqueue`.

   Virtio device driver in the guest allocates `virtqueue` and `vring` memory, the control block for `vring` and the shared memory between the host and the guest, respectively. Then it executes I/O access to the IO Region in QEMU and sets vring physical address of the guest. In this procedures, the host and the guest finally establish the shared memory with `vring`.

6. Send Request.

   Virtio device driver in the guest sends requests written on the `vring` by I/O accessing to the IO Region.

7. Send Response in QEMU.

   Callbacks in the IO Region do serve and send back the result by inject interrupt to the guest via KVM.

In the next section, we describe the detailed implementation of the VESPER for the Virtio in QEMU and KVM.

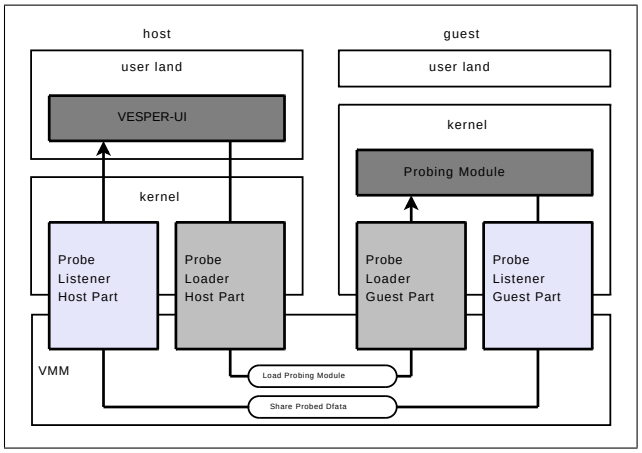

Figure 1: Architecture of VESPER

3 Implementation of VESPER

For probing the guest, VESPER uses the tracing facilities to hook into the guest Linux kernel, and uses `relayfs` to record probed data in the probe handler of the tracing facilities. In this section, we take a brief look at the VESPER architecture and its semantics. Then we present the implementation of Virtio support in VESPER.

3.1 VESPER Architecture

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

In addition, 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.

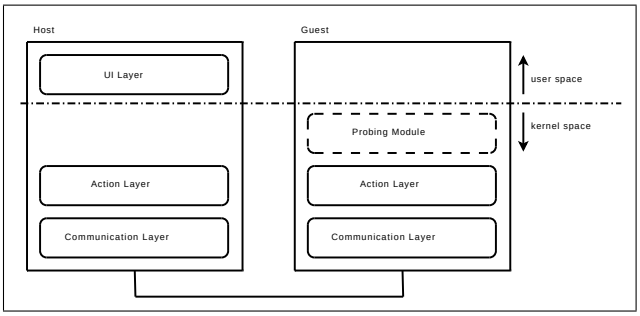

Figure 2: Layer of VESPER

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.

3.2 VESPER Semantics

In order to implant probes into the guest, VESPER should 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 the tracing facilities 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

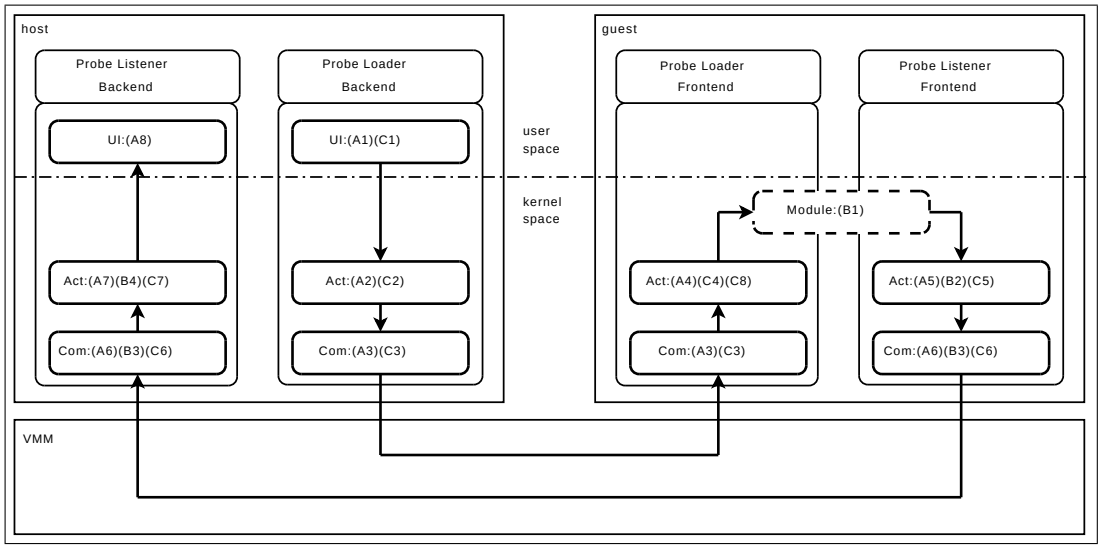

Figure 3: Process Flow of VESPER.

size, and its address with others related to module parameters, if any.

A3. Send/Receive Request

Through the interface provided by the VMM or Virtio, 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, if any, or Virtio, 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.

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.

3.3 Implementation of VESPER supporting Virtio

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 supporting Virtio.

3.3.1 Probe Loader

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. Usually, the guest sends request. However, in this case, the host should send the module image and request to implant it to the guest memory. Therefore, the loader utilizes the event mechanism of QEMU. Like `virtio_console`, the loader attaches the watcher, to poll UI layer, to QEMU. When the module image is written to UI layer, QEMU sends the event notification to the loader. Then the loader dumps the image to `vring`, established between the host and the guest, and injects interrupt to the guest.

In writing the image to the `vring`, `struct iovec` and `struct scatterlist` translation is needed to use Virtio facilities. In QEMU, `VirtQueueElement` and `virtqueue_push` in `qemu/hw/virtio.c` helpers are prepared for the needs. On the other hand, the loader in the guest uses `sg_set_buf` and `sg_init_one` to prepare the memory where the host writes the image. To load the module into the kernel, the name of module and the size of module are needed in invoking `load_module` in the kernel. So, the header of the request, `struct virtio_loader_inhdr` is attached to inform the guest of the related information of the module. From the viewpoint of the guest, the guest is not able to find out how much memory is needed to accomplish the the request. So, the guest prepares only one page for the request and notify the memory size per request in the header of `struct virtio_loader_outhdr`. Then the host sets the left length of the module onto the header as well. Figure 4 depicts the details.

3.3.2 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.

Relayfs in the guest tends to allocate its buffers by pages, and the listener sets their physical address to `vring` to share them with the host. So, the listener in QEMU can see the all data on the buffers. However, those data should be copied to the relayfs in the host kernel to keep user interface to the relayfs for user application utilizing the probed data on the host. In addition, even QEMU and the relayfs can share the buffers, they should share the control information such as the read index and padding value to access proper data as well. Our design decision to tackle these problems is :

1. QEMU `mmaps` the buffers pages notified by `vring` into the action layer of the listener in the host. The action layer creates relayfs rchan with the buffers.

```
typedef struct VirtIOLoader
{
    VirtIODevice vdev;
    VirtQueue *vq;
};

typedef struct VirtIOLoaderReq
{
    VirtIOVsp *dev;
    VirtQueueElement elem;
    struct virtio_loader_inhdr *in;
    struct virtio_loader_outhdr *out;
};

struct virtio_loader_outhdr
{
    uint32_t type;
    uint8_t max_size_per_req;
    uint8_t status;
};

struct virtio_loader_inhdr
{
    uint32_t type;
    uint8_t mod_name[];
    uint64_t mod_size;
    uint64_t left_len;
};
```

Figure 4: Prototype of the device of ProbeLoader

2. Every time the relayfs in the guest updates the read index, the guest will notify the index via Virtio. QEMU will access the action layer in the host to inform it. Then the action layer updates its index of the rchan.

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.

3.3.3 Buffer Share Function

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

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 in the host does not call `relay_open` to create a `rchan` structure, which is a control structure of relayfs. This is because Probe Listener does not need to newly allocate the pages for the relay buffers, but should just map the pages exported by the guest through QEMU. 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.

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.

3.3.4 Index Update Function

When the probe listener shares the relay buffers between the host and the 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 Virtio for the information transfer. The guest's probe listener creates a message including the information, pushes it to `vring`, and then notifies the host's probe listener in QEMU. The host's probe listener gets the message from `vring` and accesses the action layer in the host to update 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 `vring` 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.

Figure 5 depicts the definition of the device for Probe Listener.

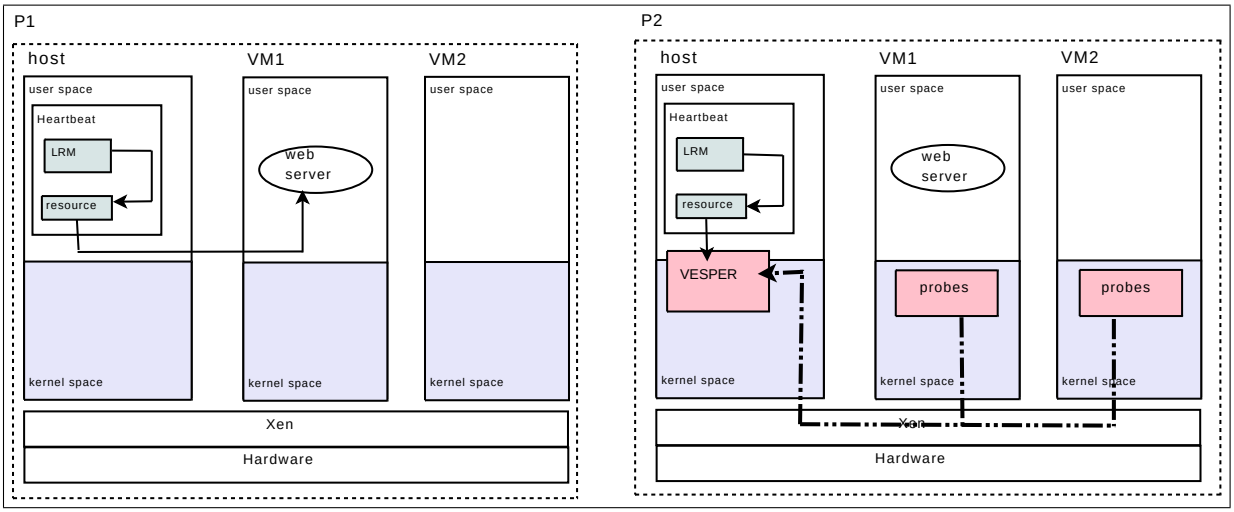

Figure 6: Test environment to demonstrate the usability of VESPER.

4 Tracing and evaluation

We have two different tracing facilities. one is for static tracing technology, tracepoints representatively and the other is for dynamic tracing technology, Kprobes. each technology has advantages and disadvantages compared to each other. The static tracing tends to show lower overhead than the dynamic tracing. However, the static tracing requires the kernel recompiling. So, we choose the proper tracing technology case by case. In our test environment, Figure 6, we use trace points to monitor system-wide figures like memory pressures, scheduling and etc. On the other hand, we use Kprobes to monitor application-specific figures like "signal" to the specific application.

In the environment, we prepare two physical machines. We set up two guests as resources managed by the LRM (Local Resource Manager) of Heartbeat on each physical machine. 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.

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. To evaluate VESPER, we insert Kprobes around `send_sigal` and trace points probe around `out_of_memory`. Then, we send `SIGTERM` to the webserver at first. Kprobes thus inserted will put clues like the callstack to the `SIGTERM` on the relayfs. Then, we 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 the webserver is gone, because of the prompt notification done by VESPER. For the next, we execute a test program allocating a huge memory to occur `OOM`. Also, we measure the recovery time on P1 and P2. In this case, P1 did not recognize `OOM` because the test program is killed.

Through the experiment, we could verify better performance on response latency with 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 especially in dynamic tracing tracing technology. 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.

```
typedef struct VirtIOListener
{
    VirtIODevice vdev;
    VirtQueue *vq;
};

typedef struct VirtIOListenerReq
{
    VirtIOVsp *dev;
    VirtQueueElement elem;
    struct virtio_listener_inhdr *in;
    struct virtio_listener_outhdr *out;
};

struct virtio_listener_outhdr
{
    uint32_t type;
    unit8_t guestid;
    target_phys_addr_t buf_addr;
    uint8_t mod_name[];
    uint64_t buf_size;
};

struct virtio_listener_inhdr
{
    uint8_t status;
};
```

Figure 5: Prototype of the device of ProbeListener

5 Conclusion and future works

In this paper, we expanded VESPER to Virtio on KVM 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. Finally we discussed some considerations on places and overhead of probing. To address these considerations, we have some plans for future VESPER developments.

5.1 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.

5.2 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.

6 Acknowledgements

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

7 Legal Statements

References

[1] LKML, `http://lkml.org`

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

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

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

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

[6] Virtio, `http://lwn.net/Articles/239238`.

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

[8] QEMU, `http://www.qemu.org`

Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*

Programme Committee

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
James Bottomley, *Novell*
Bdale Garbee, *HP*
Dave Jones, *Red Hat*
Dirk Hohndel, *Intel*
Gerrit Huizenga, *IBM*
Alasdair Kergon, *Red Hat*
Matthew Wilson, *rPath*

Proceedings Committee

Robyn Bergeron
Chris Dukes, *workfrog.com*
Jonas Fonseca
John 'Warthog9' Hawley

With thanks to
John W. Lockhart, *Red Hat*