Migration Workflow Tracing

Anupama Chandwani, Ricardo Koller, Arunachalam Ramanathan and Gabriel Tasaruk-Levin

January 30, 2014

Abstract

Debugging through a virtualization platform like vSphere [5] can be challenging as it involves multiple layers starting from management layer all the way down to hypervisor. It requires inspection of different processes spread through the management and operating system stack. To add to the complexity, these layers are written in different programming languages. Live migration, for instance, has modules that span across Virtual Center (VC), hostd, VMX, and the VMKernel. These modules correlate by common cross-layer identifiers. It is very difficult to trace a particular operation in detail across these layers, let alone the interaction between operations.

We propose a workflow tracing mechanism like Dapper [1], that will span all layers in the stack, following an operation, reporting specific task events, sub-tasks, and stats to a central authority. These stats and events can then be used to graphically represent with an ordered operation timeline, overlaying sub-tasks and their associated events and stats. Visualizing this timeline can easily help isolate problematic components of an operation, for example immediately identifying sub-tasks taking more time than expected. More importantly it can answer difficult inter-operation issues like sudden drop in network bandwidth or rise in latency around acquiring a certain lock. Operation timelines can also be chained together to track VM behavior over time. Such historical data could be used as input in predicting future migration performance or in understanding the interaction between different layers.

1 Introduction

Migration workflow spans multiple layers. The VC's workflow engine executes tasks like preparing the source and destination VMs and initiating migration. These tasks are handled by hostd. VMX inturn waits for some of these tasks to complete. So it's really a state machine across all these layers.

To best explain the interaction, lets take a migration failure due to timeout scenario. Now a migration can timeout while performing any of its sub-operations, for the sake of our discussion, lets assume its while registering the destination VM. In such a failure, the source VM will timeout waiting for destination VM to powerOn. However, when destination VM eventually powers on, it will get to *WaitForData* and timeout since source will not be sending any data to it.

Debugging this scenario with existing mechanisms, will require the developer to get vm-support of the failure, *migrationID* of the operation and find relevant logs in VM home directory. In this instance, the destination VMX logfile has a log `"Failed waiting for data. Error 195887137. Timeout. "`. The source VMX logfile has `"The source detected that the destination failed to resume."` Now based on the timestamps, the developer can find that it's the source VMX that declared failure first while waiting for something from a layer above.

Going up the stack, developer tries to find the *op-ID* corresponding to this migration, since hostd and vmx use different identifiers for the same migration. From the hostd logs, we need information of how much time each task took and which one took exceptionally long. Once this information is retrieved by matching the start and end of each operation, the developer can isolate the cause of migration failure. You can only imagine how much more complex this would get if more layers were interacting for an operation or if hostd was handling multiple migrations simultaneously.

Consider the same failure with our tracing mechanism enabled. The QE can request to build a timeline for this operation along with vm-support. Now the developer takes a look at the graph that overlays all tasks on timeline and immediately points out that the time taken for registering destination VM is in minutes versus the usual milliseconds and can root cause it fairly quickly.

On the other hand, operation timelines can be chained together to montior VM behavior. Want to know where a VM has vMotioned to over the last few days? We can compute a timeline of operations for the VM and interactively drill down into each operation. Want to know whether there were concurrent operations at the same time? Pop up a level and view operations happening on the host or VC at the same time.

Inspecting the operation timeline can also helps us understand the VM workload. For example, in case of a vMotion, if we see some *preCopy* iterations take longer than the other iterations, it could possibly mean that the VM's dirty rate was higher during these intervals, meaning the workload suddenly became memory intensive. Such guest specific information can be very useful for load balancing and stats measuring modules like DRS and vCOPs.

1.1 Existing solutions

We have vProbes loosely based on Dtrace [4] to instrument and analyse code. But it is more commonly used for understanding interaction and code flow than for debugging. This framework also lacks an analysis phase, developers still have to go over the output to isolate relevant entry.

Our idea is a natural extension to the existing VOB framework, but on steriods: don't just report errors, report each step along the way to paint a complete picture of the operation spanning the entire stack.

2 Design Goals

With the above objectives of debugging, performance monitoring and behavior prediction, we have identified certain design goals that need to be met while trying to achieve our objectives.

- *No workflow deviation*: we do not want the code to behave differently in the presence and absence of this mechanism. We want our tracing mechanism to silently monitor execution without playing an active role in influencing its execution sequence.
- *Low overhead*: we want our tracing mechanism to have little or negligible impact on the operation's timeline. A lot of effort has gone into making migration workflow highly optimized, the last thing we would want to do is induce any kind of delay in this workflow.
- *Complete coverage*: ease in debugging is our main objective, so we want to make sure we have coverage of all the operations via our tracing mechanism.

The above design goals form the baseline of all our implementation decisions. Though we would like our mechanism to always abide by them, it is not a realistic assumption. We design our system to never deviate from existing workflow, thus it is always achieved. However, when the system is heavily overloaded, achieving negligible performance overhead can be tricky. We still strive to achieve it by sacrificing complete coverage of all operations. Our logging mechanism will selectively drop low priority sub-task information. However, we always have *some* information about an operation, so while debugging, the mechanism can point to a sub-task that caused a failure.

3 Architecture

Let us start with defining a few key concepts and components of our tracing mechanism. A background on these ideas will help build the concept further.

3.1 Identifier

The crux of the tracing mechanism is to be able to co-relate all the tasks, sub-tasks of an operation from different layers of the stack. This is achieved by having a globally unique identifier for an operation throughout the stack.

While this identifier is unique for an operation, it should also be able to convey the dependency and relationship of different tasks and sub-tasks within that operation. This is achieved by adding a fixed length suffix to this identifier while initiating any subtask. We call this new identifier as task identifier. The suffix will be different for each subtask of a given task and will increment in ascending order, giving us an order in which subtasks were issued. From a given subtask, if a new subtask is issued, then the surfix is not replaced, rather it further appended to maintain its relation with the original initiator of the task. This behavior of adding a suffix to the subtask helps identify a task's initiator, in terms of a relational graph, it's parent.

VMWare already has a concept of op-ID across VC and hostd. Op-ID is a unique identifier string for a VC operation which appends suffix for each new sub-task. We could either extend op-ID and use them in VMX and VMKernel code or implement something very similar to it that can be used throughout the stack.

3.2 Trace tree

A trace tree is a flow-chart (or state machine diagram) for an operation. The tree is built to graphically represent the relation between different tasks, subtasks and events. The tree is indexed by the task identifier, which is the unique operation identifier plus it's suffix. Each task and event will be a node in this tree and their task identifier will be the index of that node. All subtasks of a task, will point to it's caller as the parent. The parent-child information is available by removing the fixed length suffix from the sub-task's identifier, thus getting the parent task's identifier. So on and so forth, till the initiating task in VC forms the root of this tree.

This relationship tree can later be used to overlay the operation on an ordered timeline, while clicking on tasks to browse detailed timing and event information of subtasks.

3.3 Logger API

The most important step in the tracing mechanism is reporting tasks and events during execution. To achieve this, we provide APIs that will query relevant information from the caller and log this information. We use well defined APIs to log information because we want the information to be in a certain format so it's easier to analyse it.

We identify three basic types of APIs, task API, event API and stat API.

- task API: A call to this API should usually be followed by a function call that will perform the new task, or a code snippet that does it. There should be a subsequent complete call for this task.
- event API: This API is used to report a certain state of the task. For example, in storage vMotion, moving to a different phase in disk copy is an event and scheduling a disk to copy is a sub-task.
- stat API: This API is used to report stats from different layers. Data from these APIs along with event information can give a very detailed picture of the operation. For example, number of pages dirtied during each *preCopy* iteration.

3.4 Logging abstraction

Unlike Dapper [1], our logging APIs are very closely tied with the code semantics. Also these APIs are explicitly added in the code in locations that the developer feels is important. We do not try to make our logging mechanism abstract of the layers it is tracing, the reason being, we can achieve better semantic tracing when enabled from the code itself. Also, since all the layers in the stack belongs to VMware, we do not think this is a limitation, rather it makes our tracing more powerful.

Integrating our logger with the code also takes the load of recreating operation semantic model off the analysis phase. Since logging is done from within the code we can keep the semantic knowledge intact, we do not need to recreate the model of our operation, like Magpie [2] to understand the logs in the analysis phase.

3.5 Components

Our tracing mechanism has three major components:

1. *Tracer*: This component consists of each layer in the stack that calls the logger APIs. The API invocation is integrated with the code to semantically give information about all the important tasks and subtasks. The code handling these APIs asynchronously writes the data collected from the caller site into a log file.

2. *Collector daemon*: This component collects logs from the logfile in the background and adds them as entries into a huge database that is indexed by operation identifier of the root task.

3. *Query engine*: This is the offline user interactive component of the tracing mechanism. Based on the query, this module will build up a graphical output from the collected data in the database. A complete operation timeline can be determined by sub-task identifer and timestamp.

We try to optimize each of these components, for instance, the logger API handler is a multi-threaded program working asynchronously in the background. It is possible that the lock on log file becomes a bottleneck and induce overhead, so we plan on using multiple log files. Data from all of them can later be collaborated based on timestamp. The asynchronous handler helps achieve low overhead, but at the same time could introduce ambiguity in the sequence of events, if the later API, having the same timestamp, gets reported before the one issued first. We understand there can exist such ambiguity in the sequence of events that get invoked from a very small window of time. For ambiguity between related events, we recommend looking at code to determine the expected sequence.

4 Related work

There is a lot of interesting work done in the field of improvising tracing mechanism for distributed and multi-tier systems. We borrow some of the concepts from successful tracing mechanisms like Dapper [1] and Magpie [2]. However, our mechanism is largely specialized for VMWare's architecture.

Dapper [1] heavily relays on sampling to achieve low overhead. The tracing mechanism logs traces for one in every one thousand calls. This has proven enough for their debugging process. Also the scaling operation either logs all subtasks of an operation or none. They do not do selective sampling. On the other hand, we want to trace all operations to be able to help debug every bug that gets reported. Also having *some* information about all operations rather than no information is more interesting to us for inter-operation behavior analysis.

Dapper [1] also implements it's tracing layer in lower level libraries, like threading and RPC library. This helps it to avoid having any application level change. Because of such low level tracing, the trace tree is indexed in a context based manner. On the other hand, we found it a lot more informative to have logging induced at semantic locations. This also means that the trace tree that gets generated from our logs already has semantic information and can directly be used for graphically representation. We avoid a complete phase of analysing the run with a learning model to understand the operation, something that Magpie [2] has to do.

VMWare already borrows the concept of Dtrace [4] from Solaris, however, our vProbe implementation is largely used to understand code. Also there is no existing way of unifying the output from different layers, according to timestamp, using vProbes. Our tracing mechanism becomes really strong and interactive due to its integration with a database to help generate different outputs on demand.

5 Conclusion

In this paper we have described the usefulness of an end-to-end tracing mechanism that will trace an operation across multiple layers of a stack. With the example of migration, we concluded how debugging becomes intuitive and easy for complicated interdependent operations.

Moreover this mechanism provides VM behavior information like, history of operations, performance of a VM over multiple runs, etc for free. Our ideology of not just reporting errors, but each step along the way to paint a complete picture of the operation spanning the entire stack really makes our mechanism powerful.

We have described in detail the design, architecture and usecases of our mechanism, there is much more we could do with the information this tracing mechanism collects and use this information in even more sophisticated ways. Starting with migration workflow, our aim is to spread this across all the components of vSphere.

References

[1] B.H.Sigelman, L.A.Barroso, M.Burrows, P.Stephenson, M.Plakal, D.Beaver, S.Jaspan, C.Shanbhag. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure . In Proceedings of Google Technical Report.* 2010

[2] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. *Magpie: online modelling and performance-aware systems. In Proceedings of USENIX HotOS IX.* 2010

[3] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. *X-Trace: A Pervasive Network Tracing Framework. In Proceedings of USENIX NSDI.* 2007

[4] B.M.Cantrill, M.W.Shapiro and A.H.Leventhal. *Dynamic Instrumentation of Production Systems. In Proceedings of USENIX.* 2004

[5] http://www.vmware.com/products/vsphere/