OVERVIEW

Docker started as a tool for developers and test engineers to simplify software delivery, but it has rapidly evolved into a production-ready infrastructure platform. It promises to deliver software more flexibly and more scalably to your end users, while at the same time making microservices a reality.

As any new platform moves into production, monitoring becomes an important aspect of its viability. That’s especially true with a platform like Docker, where its architectural model actually changes how you need to instrument your systems in order to monitor it properly.

This Refcard will lay out the basics of the Docker monitoring challenge, give you hands on experience with basic monitoring options, and also spell out some more advanced options.

THE DOCKER MONITORING CHALLENGE

Containers have gained prominence as the building blocks of microservices. The speed, portability, and isolation of containers made it easy for developers to embrace a microservice model. There’s been a lot written on the benefits of containers, so we won’t recount it all here.

Containers are black boxes to most systems that live around them. That’s incredibly useful for development, enabling a high level of portability from Dev through Prod, from developer laptop to cloud. But when it comes to operating, monitoring, and troubleshooting a service, black boxes make common activities harder, leading us to wonder: what’s running in the container? How is the application code performing? Is it spitting out important custom metrics? From a DevOps perspective, you need deep visibility inside containers rather than just knowing that some containers exist.

INTRODUCTION TO DOCKER MONITORING

© DZONE, INC. DZONE.COM
WHAT ARE YOUR DOCKER CONTAINERS DOING?

“Sysdig gives my team unprecedented visibility into our applications. We're using it extensively to help us solve complex issues in our most innovative products” – Adam Hertz, VP of Engineering at Comcast

We’re about to make your life easier.

Enterprises are moving to Docker containers for faster software development that promises developer agility, software portability, and scalable microservices. That’s all good news.

Here’s the bad news: Docker environments get harder to operate, because microservices and containers break legacy monitoring tools. Containers are great for developers, but they will wreak havoc on your DevOps team.

Worried? Relax, we got your back.

Sysdig is the first and only solution that can natively monitor Docker environments. Sysdig ContainerVision™ provides request-level visibility inside containers without invasive instrumentation. This approach overcomes the limitations of your old monitoring tools, and at the same time makes monitoring Docker simpler and more robust. All that means you can sleep at night when Kubernetes is auto-scaling your apps. Go on, press the snooze button.

Visit sysdig.com and find out!
There is an additional, advanced topic that I’ll touch on briefly in this Refcard: Docker containers often also use an orchestration to aggregate containers into services. These orchestration systems provide additional metadata that can be used to better monitor Docker. We will see an example later on of using Docker labels in this way to assist in service-level monitoring.

Let’s now put some of this into practice with some common, open-source-based ways of gleaning metrics from Docker.

**DOCKER STATS API**

Docker has one unified API and in fact all commands you’d run from a CLI are simply tapping that endpoint.

For example, if you have a host running Docker, docker ps would return this, which is just a reformattting of API data.

<table>
<thead>
<tr>
<th>CONTAINER ID</th>
<th>IMAGE</th>
<th>COMMAND</th>
<th>CREATED</th>
</tr>
</thead>
<tbody>
<tr>
<td>888b97a45653</td>
<td>sysdig/sysdig-bot</td>
<td><code>python bot.py</code></td>
<td>26 hours ago</td>
</tr>
<tr>
<td>f561df0e9e5f</td>
<td>sysdig/agent</td>
<td><code>/entrypoint.sh</code></td>
<td>39 hours ago</td>
</tr>
<tr>
<td>9987f3a2e6bc2</td>
<td>wordpress</td>
<td><code>/entrypoint.sh</code></td>
<td>39 hours ago</td>
</tr>
<tr>
<td>11f1235aafde</td>
<td>mysql</td>
<td><code>docker-entrypoint.sh</code></td>
<td>39 hours ago</td>
</tr>
</tbody>
</table>

To show this let’s query the API via curl and ask for all containers running. For brevity we’re showing the JSON blob below for just one container, and prettied up the JSON.

```
curl --unix-socket /var/run/docker.sock http://containers/json
```
curl --unix-socket /var/run/docker.sock
http://containers/8a9973a456b3/stats

"system\_cpu\_usage":266670930000000,"throttling\_data":},
"cpu\_stats":{},"system\_cpu\_usage":266671910000000,"throttling\_data":},
"memory\_stats":{"usage":27516928,"max\_usage":31395840,"stats":{"active\_anon":17494016,"active\_file":5144576,"cache":10022912,

Not pretty, but an awful lot of metrics for us to work with!
If you wanted a one-shot set of metrics instead of streaming, use the
stream=false option:

curl --unix-socket /var/run/docker.sock
http://containers/8a9973a456b3/stats?stream=false

DOCKER MONITORING OVER TIME & IN-DEPTH

As you've probably guessed, the API is useful to get started but likely
not the only thing you need to robustly monitor your applications
running in Docker. The API is limiting in two ways: 1) it doesn’t allow
you to perform time-based trending and analysis, and 2) it doesn’t
give you the ability to do deep analysis on application- or system-
level data. Let’s attack these problems with cAdvisor and sysdig.

cAdvisor is a simple server that taps the Docker API and provides
one minute of historical data in 1-second increments. It’s a useful
way to visualize what’s going on at a high level with your Docker
containers on a given host. cAdvisor simply requires one container
per host that you’d like to visualize.

sudo docker run \
--volume=/rootfs:ro \
--volume=/var/run:/var/run:rw \
--volume=/sys:/sys:ro \
--volume=/var/lib/docker/:/var/lib/docker:ro \
--publish=8080:8080 \
--detach=true \
--name=cadvisor \
google/cadvisor:latest

cAdvisor is now running (in the background) on http://localhost:8080.
The setup includes directories with Docker state cAdvisor needs to
observe. Accessing the interface gives you this:

If you are looking to historically graph this data, you could also
route data from cAdvisor to numerous time-series datastores
via plugins, described here. Tying an open-source visualization
engine on top of this, like Grafana, will allow you to produce
something like this:

In most of these cases, however, we’re limited to basic CPU,
memory, and network data from these tools. What if we wanted to
get deeper—to not only monitor resource usage, but processes, files,
ports, and more?

DOCKER MONITORING AND DEEP TROUBLESHOOTING WITH SYSDIG

That’s where another open-source tool, sysdig, comes into play. It’s
a Linux visibility tool with powerful command-line options that
allow you to control what to look at and display it. You can also use
csysdig, its curses-based interface, for an easier way to start. Sysdig
also has the concept of chisels, which are pre-defined modules that
simplify common actions.

Once you install sysdig as a process or a container on your machine,
it sees every process, every network action, and every file action on
the host. You can use sysdig “live” or view any amount of historical
data via a system capture file.

As a next step, we can take a look at the total CPU usage of each
running container:

```bash
$ sudo sysdig -c topcontainers\_cpu

CPU% container.name
----------------------------------------------------------
96.13% mysql
15.93% wordpress1
7.27% haproxy
3.46% wordpress2
...```

This tells us which containers are consuming the machine’s CPU.
What if we want to observe the CPU usage of a single process,
but don’t know which container the process belongs to? Before
answering this question, let me introduce the -pc (or -pcontainer)
command-line switch. This switch tells sysdig that we are
requesting container context in the output.

For instance, sysdig offers a chisel called topprocs\_cpu, which we
can use to see the top processes in terms of CPU usage. Invoking
this chisel in conjunction with -pc will add information about
which container each process belongs to.

```
$ sudo sysdig -pc -c topprocs\_cpu
```

<table>
<thead>
<tr>
<th>CPU%</th>
<th>Process</th>
<th>Host_pid</th>
<th>Container_pid</th>
<th>container_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>34.10%</td>
<td>mate-termi</td>
<td>3831</td>
<td>3831</td>
<td>Host</td>
</tr>
<tr>
<td>39.08%</td>
<td>bash</td>
<td>2093</td>
<td>2093</td>
<td>Host</td>
</tr>
<tr>
<td>19.99%</td>
<td>curl</td>
<td>12177</td>
<td>44</td>
<td>wordpress4</td>
</tr>
<tr>
<td>13.04%</td>
<td>sleep</td>
<td>2392</td>
<td>45</td>
<td>wordpress2</td>
</tr>
<tr>
<td>9.80%</td>
<td>mysql</td>
<td>18113</td>
<td>44</td>
<td>wordpress1</td>
</tr>
<tr>
<td>9.58%</td>
<td>apache2</td>
<td>11940</td>
<td>44</td>
<td>phpmyadmin</td>
</tr>
<tr>
<td>9.88%</td>
<td>apache2</td>
<td>2381</td>
<td>44</td>
<td>wordpress2</td>
</tr>
<tr>
<td>8.78%</td>
<td>mysql</td>
<td>99</td>
<td>99</td>
<td>wordpress4</td>
</tr>
<tr>
<td>6.91%</td>
<td>mysql</td>
<td>1789</td>
<td>32</td>
<td>mysql</td>
</tr>
</tbody>
</table>

As you can see, this includes details such as both the external and the internal PID and the container name.

Keep in mind: `-pc` will add container context to many of the command lines that you use, including the vanilla `sysdig` output.

By the way, you can do all of these actions live or create a "capture" of historical data. Captures are specified by:

```
$ sysdig –w myfile.scap
```

And then analysis works exactly the same.

What if we want to zoom into a single container and only see the processes running inside it? It’s just a matter of using the same `topprocs\_cpu` chisel, but this time with a filter:

```
$ sudo sysdig -pc -c topprocs\_cpu container.name=client
```

Compared to docker `top` and friends, this filtering functionality gives us the flexibility to decide which containers we see. For example, this command line shows processes from all of the `wordpress` containers:

```
$ sudo sysdig -pc -c topprocs\_cpu container.name=wordpress
```

<table>
<thead>
<tr>
<th>CPU%</th>
<th>Process</th>
<th>container.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>02.69%</td>
<td>bash</td>
<td>client</td>
</tr>
<tr>
<td>31.04%</td>
<td>curl</td>
<td>client</td>
</tr>
<tr>
<td>0.74%</td>
<td>sleep</td>
<td>client</td>
</tr>
</tbody>
</table>

So to recap, we can:

- See every process running in each container including internal and external PIDs
- Dig down into individual containers
- Filter to any set of containers using simple, intuitive filters

...all without installing a single thing inside each container.

Now let’s move on to the network, where things get even more interesting.

We can see network utilization broken up by process:

```
$ sudo sysdig -pc -c topprocs\_net
```

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Process</th>
<th>Host_pid</th>
<th>Container_pid</th>
<th>container_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>72.06KB</td>
<td>haproxy</td>
<td>7385</td>
<td>13</td>
<td>haproxy</td>
</tr>
<tr>
<td>56.96KB</td>
<td>docker_io</td>
<td>1775</td>
<td>7039</td>
<td>host</td>
</tr>
<tr>
<td>44.45KB</td>
<td>mysql</td>
<td>6995</td>
<td>91</td>
<td>mysql</td>
</tr>
<tr>
<td>29.36KB</td>
<td>apache2</td>
<td>7963</td>
<td>124</td>
<td>wordpress1</td>
</tr>
<tr>
<td>29.36KB</td>
<td>apache2</td>
<td>26895</td>
<td>126</td>
<td>wordpress2</td>
</tr>
<tr>
<td>29.36KB</td>
<td>apache2</td>
<td>27935</td>
<td>132</td>
<td>wordpress3</td>
</tr>
<tr>
<td>29.36KB</td>
<td>apache2</td>
<td>27306</td>
<td>125</td>
<td>wordpress4</td>
</tr>
<tr>
<td>22.23KB</td>
<td>mysql</td>
<td>6995</td>
<td>90</td>
<td>mysql</td>
</tr>
</tbody>
</table>

Note how this includes the internal PID and the container name of the processes that are causing most network activity, which is useful if we need to attach to the container to fix stuff. We can also see the top connections on this machine:

```
$ sudo sysdig -pc -c topconns
```

<table>
<thead>
<tr>
<th>Bytes</th>
<th>container.name</th>
<th>Proto</th>
<th>Conn</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.23KB</td>
<td>wordpress3</td>
<td>tcp</td>
<td>172.17.0.5:46955-&gt;172.17.0.2:3306</td>
</tr>
<tr>
<td>22.23KB</td>
<td>wordpress1</td>
<td>tcp</td>
<td>172.17.0.2:3306-&gt;172.17.0.3:47244</td>
</tr>
<tr>
<td>22.23KB</td>
<td>mysql</td>
<td>tcp</td>
<td>172.17.0.4:55780-&gt;172.17.0.2:3306</td>
</tr>
<tr>
<td>14.21KB</td>
<td>host</td>
<td>tcp</td>
<td>127.0.0.1:60149-&gt;172.17.0.2:3306</td>
</tr>
</tbody>
</table>

This command line shows the top files in terms of file I/O, and tells you which container they belong to:

```
$ sudo sysdig -pc -c topfiles\_bytes
```

<table>
<thead>
<tr>
<th>Bytes</th>
<th>container.name</th>
<th>Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>63.21KB</td>
<td>mysql</td>
<td>/tmp/#sql_1_0.MYI</td>
</tr>
<tr>
<td>6.50KB</td>
<td>client</td>
<td>/lib/x86_64-linux-gnu/libc.so.6</td>
</tr>
<tr>
<td>3.25KB</td>
<td>client</td>
<td>/lib/x86_64-linux-gnu/libpthread.so.0</td>
</tr>
<tr>
<td>3.25KB</td>
<td>client</td>
<td>/lib/x86_64-linux-gnu/libcrypt.so.1</td>
</tr>
<tr>
<td>3.25KB</td>
<td>client</td>
<td>/lib/x86_64-linux-gnu/libwind.so.0</td>
</tr>
<tr>
<td>3.25KB</td>
<td>client</td>
<td>/lib/x86_64-linux-gnu/libgssapi_krb5.so.2</td>
</tr>
<tr>
<td>3.25KB</td>
<td>client</td>
<td>/lib/x86_64-linux-gnu/libblur_2.4.so.2</td>
</tr>
<tr>
<td>3.25KB</td>
<td>client</td>
<td>/lib/x86_64-linux-gnu/libssh.so.1.0.0</td>
</tr>
<tr>
<td>3.25KB</td>
<td>client</td>
<td>/lib/x86_64-linux-gnu/libheimbase. so.1</td>
</tr>
<tr>
<td>3.25KB</td>
<td>client</td>
<td>/lib/x86_64-linux-gnu/libcrypt.so.1</td>
</tr>
</tbody>
</table>

Naturally there is a lot more you can do with a tool like this, but that should be a sufficient start to put our knowledge to work in some real-life examples.
So now we’ve done some of the basics, and it’s time to take the training wheels off. Let’s take a look at some more complex, real-world metrics you should pay attention to. We’ll show you the metrics, talk about why they’re important and what they might mean. For this section we’ve visualized the data using Sysdig Cloud, the commercial version of Sysdig that’s designed to aggregate data across many hosts and display within a web UI. You could do the following examples via any of the open-source time-series databases, provided you’re collecting the correct information.

**REAL-WORLD EXAMPLES: WHAT TO MONITOR, WHY, AND HOW**

**VISUALIZING CPU SHARES & QUOTA**

For those of you used to monitoring in a VM-based world, you’re likely familiar with the concepts of CPU allocation, stolen CPU, and greedy VMs. Those same issues apply with containers, except they are magnified significantly. Because you may be packing containers densely on a machine, and because workloads are typically much more dynamic than in VM-based environments, you may encounter significantly more resource conflict if you’re not carefully monitoring and managing allocation. Let’s focus on CPU, as it’s a bit more complex than memory.

Let’s start by visualizing CPU shares. Imagine a host with 1 core and 3 containers using as much CPU as possible. We assign 1024 shares to one container and 512 shares to the other two. This is what we get:

The amount of unused shares is given to others relative to their weight. So if Third does not need any CPU at all?

Let’s assign a quota in milliseconds relative to a period, and the process will be able to spend on CPU only that fraction of time in a period. For example let’s consider the same case as above, now with a quota of 50ms/100ms for First and 25ms/100ms for Second and Third:

A percentage of shares used that’s consistently over 100 means we are not allocating enough resources to our services. The implication in the example above is that First and Second instead are using 140% of CPU Shares. In general, it’s OK to consume more shares than originally allocated, because the kernel tries not to waste CPU.

**VISUALIZING CPU QUOTA**

Giving processes the maximum available CPU may be not always be what you want. If your cluster is multi-tenant, or if you just need a safe ceiling for an unpredictable application, you might like to implement a hard limit on CPU utilization. The Linux kernel supports absolute CPU limits with CPU quotas. You assign a quota in milliseconds relative to a period, and the process will be able to spend on CPU only that fraction of time in a period.

For example let’s consider the same case as above, now with a quota of 50ms/100ms for First and 25ms/100ms for Second and Third:
The result is the same as with shares. The difference occurs when Third does not use the CPU allocated to it.

Now instead of giving CPU to other containers, the kernel is enforcing the absolute quota given. The total CPU usage we will see reported for the host will be 75%.

### BASIC NETWORKING DATA

Regardless of your platform, some things don’t change… and that’s certainly true when it comes to networking data. Especially with Docker in the mix, networking can become more complex and communication patterns can become more convoluted. It’s important to keep track of basic information, such as how much data is a container consuming? Emitting?

This type of data collection requires something more full-featured than the Docker API, so instead you could collect this type of information from open-source sysdig. Let’s look at some basic network data for a set of three containers each running the same Java application:

<table>
<thead>
<tr>
<th>Name</th>
<th>☐ Network Bytes In KB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>kvb_java:app43b382_kvbapp-3954293597-e46w...</td>
<td>1.9</td>
</tr>
<tr>
<td>kvb_java:app43b382_kvbapp-3954293597-p4t6...</td>
<td>3.8</td>
</tr>
<tr>
<td>kvb_java:app43b382_kvbapp-3954293597-f8w1b...</td>
<td>3.4</td>
</tr>
</tbody>
</table>

As you can see, there is some slight variation among these three containers. If, however, we saw an extreme variation, we may want to investigate further.

At the same time, since these containers are all running the same Java application, it may be more useful to consider them a “service” and see how they are performing in aggregate. This leads up to our last example.

### FROM CONTAINER TO MICROSERVICE DATA WITH LABELS

Docker provides a concept called “labels.” These are much like they sound—additional, contextual information is applied on a per-container basis. They are unstructured and non-hierarchical. As such, you can use them to broadly identify subcategories of your containers. All the containers of a given service could carry the same label, non-standard containers could carry another label, different versions of software could have yet another label. If you’re a filer and an organizer, labels will be heaven for you.

So what can we do with a label? Well, the first thing is that we can aggregate data. From the example above, let’s suppose we applied the label “javapp” to those three containers. Now, when we show our network data we see something much simpler:

<table>
<thead>
<tr>
<th>Name</th>
<th>☐ Network Bytes In KB/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>javapp (3)</td>
<td>3.7</td>
</tr>
</tbody>
</table>

One line—that’s it. In this case we’re showing the average network data across all three containers, but you could easily calculate anything that helps you better understand the performance of this collection of containers.

But let’s go a little further with labels, network data, and the “top connections” example we showed in the open-source section.

Using this information and an appropriate visualization, we can do more than create a table of network data: we can actually create a map of our services, the containers that make them up, and who they are communicating with. Here we can see the aggregated java service, the individual containers that make up the service, and (in a more complete view) would show all the other services in your environment that the java service communicates with. Note that this is a little more advanced than the other examples, and
in particular the visualization may require some coding in D3 or something similar if you want to stay fully open source.

Here we see a few different things: our “javaapp” consists of three containers (blue) and a service called “javapp” (grey), which is just an abstraction created by whoever is routing requests to those containers. We see each of those containers communicating with a Mongo service and a Redis service, and presumably those are made up of containers as well (hidden here to avoid too much complexity).

This view helps us in a few different ways:

• We quickly can understand the logical composition of our application.
• We can aggregate containers into higher-level services.
• We can easily see communication patterns among containers.
• We may be able to easily spot outliers or anomalies.

CONCLUSION
In this Refcard, we’ve walked from first principles using the Docker Stats API all the way up to more complex analysis of our system’s performance. We’ve used data sources such as cAdvisor and sysdig to analyze real-world use cases such as greedy containers or mapping network communication.

As you can see, Docker monitoring can start very simply but grow complex as you actually take containers into production. Get experience early and then grow your monitoring sophistication to what your environment requires.

ABOUT THE AUTHOR
APURVA DAVÉ @ApurvaBDave is the VP of marketing at Sysdig. He’s in marketing and (gasp!) not afraid of a command line. He’s been helping people analyze and accelerate infrastructure for the better part of two decades. He previously worked at Riverbed on both WAN acceleration and Network Analysis products, and at Inktomi on infrastructure products. He has a computer science degree from Brown University and an MBA from UC Berkley.

RESOURCES
Docker Stats Documentation: https://docs.docker.com/engine/reference/api/docker_remote_api
Sysdig Open Source Documentation: http://www.sysdig.org/wiki

BROWSE OUR COLLECTION OF FREE RESOURCES, INCLUDING:
RESEARCH GUIDES: Unbiased insight from leading tech experts
REFCARDZ: Library of 200+ reference cards covering the latest tech topics
COMMUNITIES: Share links, author articles, and engage with other tech experts

DZONE OFFERS SOMETHING FOR EVERYONE, INCLUDING NEWS, TUTORIALS, CHEAT SHEETS, RESEARCH GUIDES, FEATURE ARTICLES, SOURCE CODE AND MORE.

"DZone is a developer's dream," says PC Magazine.

Copyright © 2016 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

SPONSORSHIP OPPORTUNITIES
sales@dzone.com