

Monitoring and Trending with Prometheus

BY ED SCHOUTEN

Founded in 2012 by software engineers at SoundCloud, Prometheus is a monitoring system whose design has been inspired by Borgmon, the system that keeps an eye on jobs running on Google's internal Borg cluster. Here, we'll discuss how Prometheus works, following a hands-on approach. In the process, we'll also look at some other tools that are often used in conjunction with Prometheus, such as Grafana and some of the metrics exporters. At the end of the article, we're also going to look at an automation tool that we are developing and will release as open-source software in the near future, called Promenade.

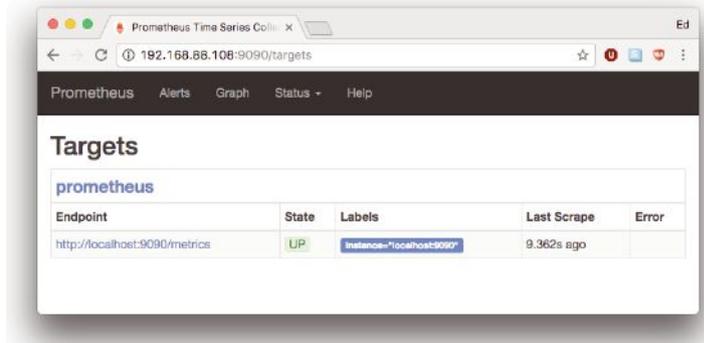
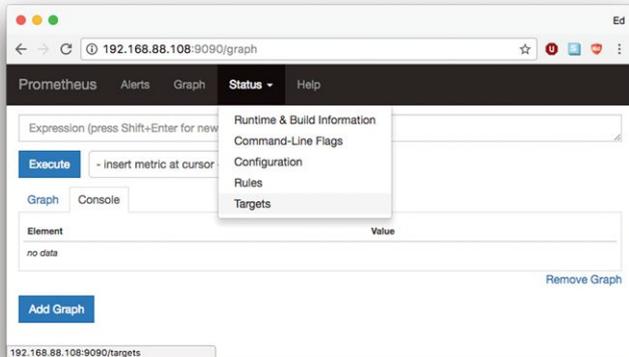
Getting Started with Prometheus

Though Prometheus can already be installed very easily using Go's build utility (`go get`), there is also a version that is packaged by the FreeBSD ports collection. One advantage of installing this package is that it ships with an `rc(8)` script that allows us to easily run it as a daemon. Prometheus should be up and running after running the commands below:

```
$ pkg install prometheus
...
New packages to be INSTALLED:
  prometheus: 1.7.1
...
Proceed with this action? [y/N]: y
...
$ sudo /usr/local/etc/rc.d/prometheus forcestart
Starting prometheus.
```

Starting Prometheus

By default, Prometheus binds a HTTP server on port 9090. If we point our browser to this server, we are presented with a page on which we can explore the data stored in Prometheus. As we've just fired up this instance, there will, of course, be little data to explore. Let's leave this page alone for now and head over to the "targets" page (2 screenshots below).



The targets page shows us which endpoints are being monitored by Prometheus, and already reveals something interesting. This instance of Prometheus has been configured to monitor itself. Prometheus is a white box monitoring system, which means that it can store metrics that are reported by targets them-

selves, as opposed to only measuring externally visible factors (e.g., TCP and HTTP health checks). By clicking on the HTTP link in the leftmost column, we can take a look at the raw metrics generated by the Prometheus server: stats related to the Go runtime's garbage collecting, threading, HTTP handling, and metrics storage (screenshot right).

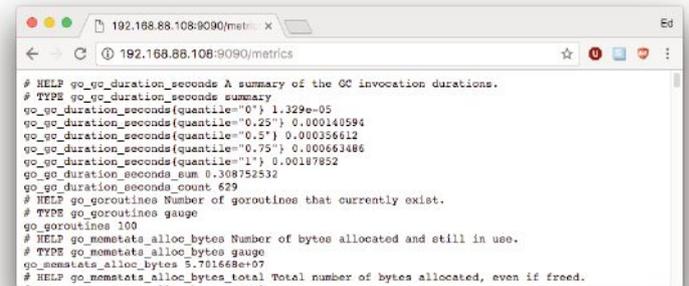
Prometheus's format for serving metrics over the network is rather simple. Each metric is placed on a single line of a HTTP response and has a numerical, 64-bits floating-point value. Metrics are uniquely identified by a name and an optional set of labels placed between curly brackets (key-value pairs).

Labels allow a program to return multiple metrics of the same name. For example, a web server could use labels to return HTTP statistics per registered virtual host or per HTTP error code class (e.g., latency of just the HTTP 200s for "www.freebsd.org").

To distinguish between identically named metrics returned by separate targets, Prometheus attaches additional labels while ingesting, such as **job** and **instance**. These labels contain values that uniquely identify the endpoint. The values for these labels are shown on Prometheus's targets page. At Kumina, we use this mechanism to attach custom labels relevant to our environment, such as physical location (data center name), system ownership (customer name), and support contract (24/7 or office hours only). These labels can then be used as part of queries and alert conditions.

On its own, Prometheus is not capable of obtaining any operating system metrics, such as CPU load, disk usage, and network I/O. It is a tool that is capable only of ingesting metrics over HTTP and indexing them. System-level metrics are instead provided by a tool called the node exporter. The node exporter is nothing more than a web server that, when visited, extracts kernel-level state through `/dev`, `sysctl`, `libkvm`, etc. and returns it in Prometheus's metrics format. Installing the node exporter on FreeBSD is quite easy:

```
$ sudo pkg install node_exporter
...
$ sudo /usr/local/etc/rc.d/node_exporter forrestart
Starting node_exporter.
```



Once started, we want to extend Prometheus's configuration to scrape the node exporter as well. By default, the node exporter listens on port 9100.

```
$ sudo vim /usr/local/etc/prometheus.yml
<!-- Add the following entry under "scrape_configs": -->
- job_name: 'node_exporter'
  static_configs:
    - targets: ['localhost:9100']
$ sudo /usr/local/etc/rc.d/prometheus onerestart
Stopping prometheus.
Starting prometheus.
```

After restarting Prometheus and refreshing the targets page, we can see that it now scrapes two targets, which is what is expected (right):

In addition to the node exporter, there are many other targets we could at this point configure. There exist exporters for most commonly used services (e.g., MySQL, Nginx, Java JMX) that convert metrics from their native format and serve them over HTTP. The Prometheus black box exporter can perform ICMP, DNS, TCP, HTTP, and SSH checks and report availability and latency. FreeBSD 12.x ships with

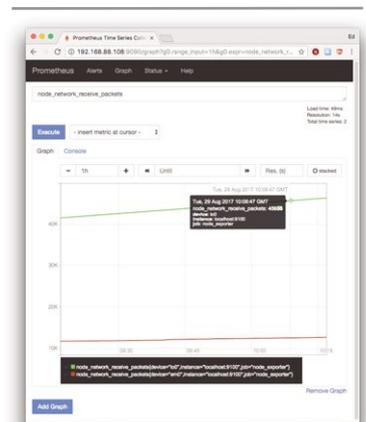
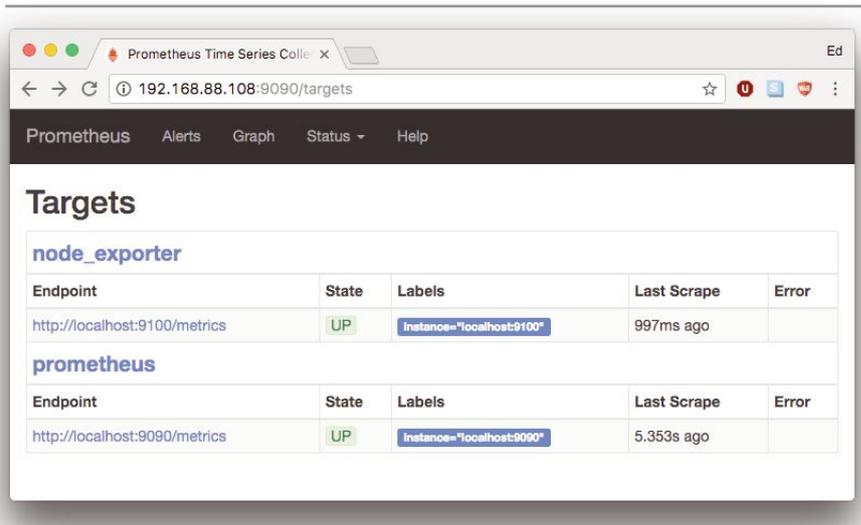
[prometheus_sysctl_exporter\(8\)](#), a tool that can serve values of arbitrary sysctls. Some of these exporters are officially maintained by the Prometheus project, while there are many others that are maintained by the community.

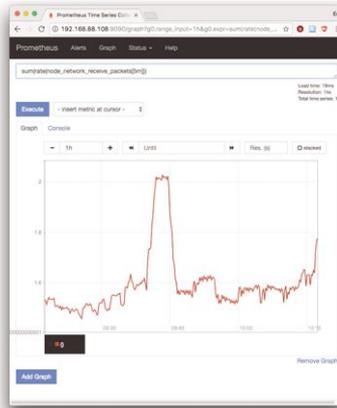
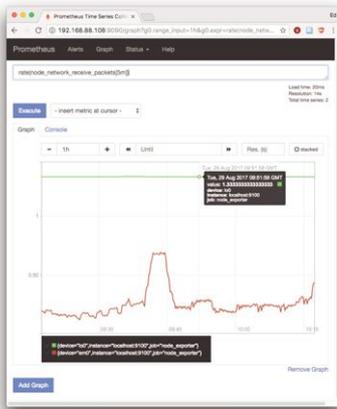
At Kumina, we have developed exporters for services including Dovecot, PHP-FPM, libvirt, OpenVPN, and Postfix. We have also designed a simple network traffic accounting daemon based on libpcap that exports per-address statistics, called Promacct. All of these tools are available on our company's GitHub page (<https://github.com/kumina>).

If you want to use Prometheus to obtain metrics from software that is being developed in-house, there is no need to make use of separate metrics exporter processes. The Prometheus project provides client libraries for various programming languages (Go, Java, Python, Ruby, etc.) that make it possible to directly annotate your code with metrics objects. For languages like Python and Java, these libraries also provide convenient function decorators to automatically count function invocations and create histograms of their running times.

PromQL: Prometheus's Query Language

After letting Prometheus gather metrics for a couple of hours, we can head over to the graphing page to explore Prometheus's dataset. Let's start off by graphing a single metric that is generated by the node exporter, [node_network_receive_packets](#). As the name suggests, this metric corresponds with the number of network packets received by the system's network interfaces. This expression produces a graph with two lines on my system: one for the loopback interface (`device="lo0"`) and one for the physical interface (`device="em0"`).





If we wanted to plot metrics only for certain hosts or network interfaces, we could append filters to the end of the expression. For example, appending `{device!~"lo[0-9]+"}` to our query would remove metrics for loopback devices by making use of negative regular expression matching. `{datacenter="frankfurt"}` would only give us results for systems in a single data center, if such a label were to exist.

Graphing the number of network packets directly doesn't seem to show us anything useful; it only renders two diagonal lines. The reason for this is that the node exporter reports the number of network packets as a counter that accumulates over time. Only when the system running the node exporter reboots (or an integer overflow of the counter occurs), it resets to zero. By letting an exporter use cumulative counters, it may safely be scraped by multiple Prometheus servers. It also makes exporters oblivious of the scraping interval configured on the Prometheus server. Even if the Prometheus server needs to reduce the number of scrapes due to high load, no packets will remain unmeasured.

In order to convert the number of packets into something meaningful, we will first need to compute its derivative. This can be accomplished by performing two operations on the metric. First of all, instead of querying for the metric's scalar value, we're going to request vectors of consecutive samples, called range vectors. We can then use the `rate()` function to turn those range vectors back into scalars, representing the rate of increase of a range vector per second. The size of the range vectors used will determine the smoothness of the resulting graph. Five-minute range vectors are good for spotting brief bursts of traffic, whereas one-hour range vectors can visualize daily traffic curves. At Kumina, we even use 7-day and 31-day range vectors for long-term network and CPU capacity planning. When changing the graph expression to

`rate(node_network_receive_packets[5m])`, we get a graph that shows the number of received packets per second, using a 5-minute rate computation.

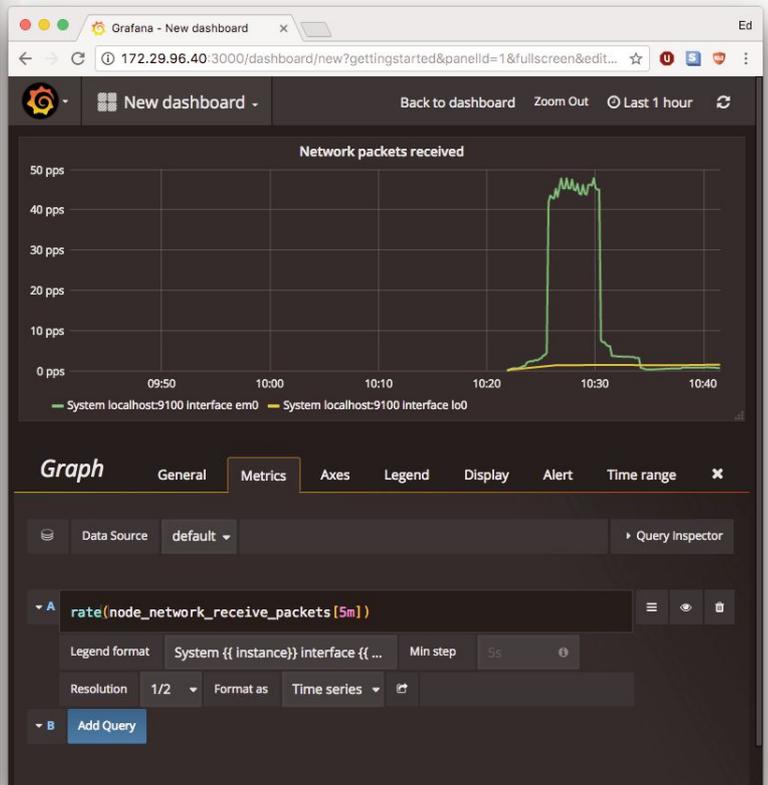
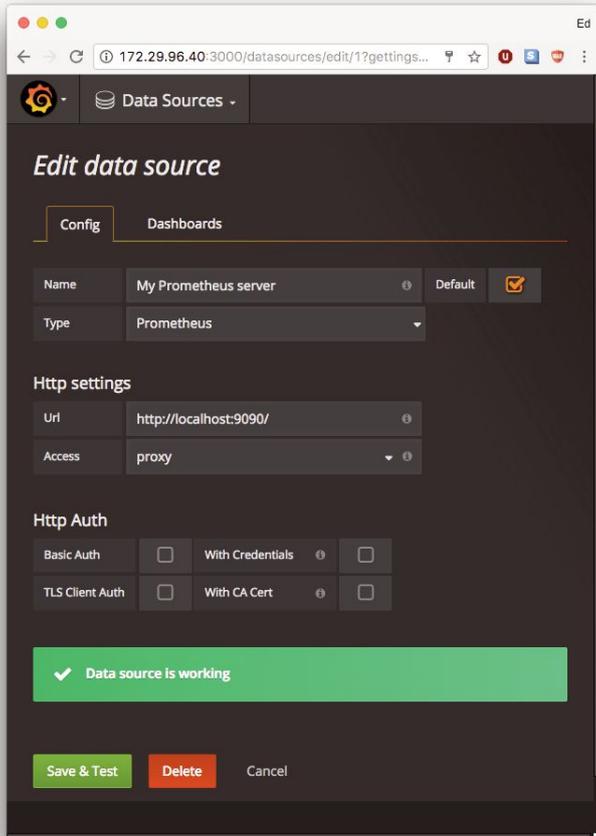
Prometheus's query syntax also supports aggregation operations with which we can reduce the number of metrics graphed. For example, the query `sum(rate(node_network_receive_packets[5m])) by (instance)` will compute bandwidth per server, as opposed to showing bandwidth per network interface. The query `topk(10, rate(node_network_receive_packets[5m])) by (datacenter)` will limit the output to just the top 10 per data center.

Creating Dashboards with Grafana

Though Prometheus's built-in web application is all right for running queries interactively, it is often desired to design dashboards that show graphs in an organized way. Prometheus used to ship with a feature called Promdash that allowed the server to serve templated HTML files. This feature has been removed in the meantime, as there are now various third-party dashboard tools that can interface with Prometheus directly, the most prominent one being Grafana. Getting Grafana to work on FreeBSD essentially follows the same recipe as what we've seen before:

```
$ sudo pkg install grafana4
...
$ sudo /usr/local/etc/rc.d/grafana forstart
Starting grafana.
```

With its default settings, Grafana will spawn a web server that listens on TCP port 3000. After pointing our browser to it and logging in with the default credentials (username "admin", password "admin"), we are presented with Grafana's home screen. The first thing we'd want to do is click on "Add data source" to configure the address of the Prometheus server that Grafana should use to fetch data—in our case, <http://localhost:9090/>.



Once completed, we can click on the next button on the home screen, titled “Create your first dashboard”. We then get presented with an empty dashboard page on which we can place panels, such as graphs, tables, heat maps, and lists. In the case of Prometheus, using graphs makes sense most of the time. When creating graphs, we can use the same query syntax as we’ve used before. By pressing the save icon at the top of the page, the dashboard gets saved on the Grafana server.

Depending on the specifications of your hardware, you may notice that dashboards will take a longer time to render as the number of graphs increase and graph queries become more complex. Executing many complex queries at the same time may cause a significant load on the Prometheus server. To solve this, Prometheus has the option to pre-compute complex queries while scraping and store its result under a different name, using recording rules. As a rule of thumb, you should use recording rules for graph queries as soon as they are more complex than simple selection expressions.

```
$ sudo vim /usr/local/etc/prometheus-rules.yml
<!-- Add the following contents: -->
my_recording_rule =
    sum(rate(node_network_receive_packets[5m]))
    by (instance)
$ sudo vim /usr/local/etc/prometheus.yml
<!-- Add the following entry under "rule_files". -->
- prometheus-rules.yml
$ sudo /usr/local/etc/rc.d/prometheus onerestart
```

The commands above show how Prometheus can be configured to make use of recording rules. In this example, we’re adding a recording rule called `my_recording_rule`, which may from now on be used in graph queries. Recording rules can be named arbitrarily, but to improve readability, it makes sense to apply some best practices. The Prometheus documentation has an article on the naming scheme suggested by the developers. For the recording rule above, the Prometheus documentation suggests it should be named `instance:node_network_receive_packets:rate5m`.

Alerting

In addition to graphing, Prometheus is also capable of generating alerts based on metric values. Alerts can be configured by declaring alerting rules that may be placed in the same file as recording rules. Below is an example of what an alerting rule looks like:

```
ALERT TargetFailedToScrape
  IF up == 0
  FOR 15m
  LABELS { severity = "page" }
  ANNOTATIONS {
    summary = "Instance {{ $labels.instance }} is down!",
    playbook_url = "http://intranet.company.com/...",
    ...
  }
```

This alert will trigger if a metric called “up” has the value zero for at least 15 minutes. The “up” metric is created by Prometheus implicitly to denote whether it has been able to scrape a target successfully. Labels that are part of the metric used in the alerting expression will also get attached to the alert itself. These labels are useful for formatting user-friendly alert messages, but also to create “silences”, patterns for alerts that should be suppressed temporarily (e.g., due to planned maintenance). Prometheus will show all registered alerting rules and their state on its “Alerts” page.

To keep its design simple, the Prometheus server only supports a single mechanism for announcing active alerts, namely by sending REST calls to some other service. The Prometheus project provides a separate daemon called Alertmanager that can process these REST calls, generate email, SMS, and Slack messages, and manage silences. The URL that Prometheus uses to perform REST calls can be configured through the `--alertmanager.url` command line flag. It may also be necessary to set the `--web.external-url` flag to the public URL of the Prometheus server, so that Alertmanager can add clickable links to its alert messages that point back to Prometheus.

Federation: a Hierarchy of Prometheus Servers

There are scenarios in which it is undesirable to use a single Prometheus server to collect metrics for all of your infrastructure. The number of targets to scrape and metrics to ingest may become too large for a single Prometheus instance. Targets may also be spread out geographically, meaning there is not a single location from which all targets can be scraped reliably. To solve this, Prometheus servers can receive HTTP GET requests on `/federate?match[]=...` to print selected stored metrics in its own format, allowing them to be ingested by other servers.

In practice, you see that this mechanism is often used to introduce hierarchy. A service that is distributed across multiple data centers may have one Prometheus server per location, scraping only the systems nearby. These Prometheus servers then use recording rules to aggregate key metrics for the entire data center, which are then scraped by a global Prometheus server. With root causing problems, one can first investigate the dashboards provided by the global Prometheus instance to determine which data centers are affected. Access to metrics on a system level can be obtained by switching to the appropriate local instance.

An advantage of such a hierarchical setup is that it may reduce disk space usage significantly. Local Prometheus instances can be configured to use volatile storage and have a short retention period (days), whereas the global instance may make use of persistent storage and have a very long retention period (years), as it only needs to hold on to a tiny fraction of the data. Storing data for multiple years is useful for long-term capacity planning.

Creating Dashboards from Python with Promenade

At Kumina, we’ve noticed that managing recording rules on a large scale may be quite challenging. Refactoring Prometheus configuration files always has a risk of breaking existing Grafana dashboards, especially because dashboards are stored in a web-based tool, as opposed to being stored in a version control system next to the Prometheus configuration files. We’re therefore working on implementing a utility to configure Prometheus and Grafana installations programmatically, called Promenade. With

Promenade, you can write Python classes that declare Grafana dashboards as follows:

```
class UnboundMetricsBuilder:
    def construct(self):
        yield Dashboard(
            title='Unbound',
            rows=[
                DashboardRow(title='Queries', graphs=self._row_queries()),
                ...
            ]
        )
    def _row_queries(self):
        yield Graph(
            title='Query rate per data center',
            queries=[
                GraphQuery(
                    expression=Sum(
                        expression=Rate(
                            expression=Metric('unbound_queries_total'),
                            duration=datetime.timedelta(minutes=5)),
                            by={'datacenter'}),
                    format='Data center %(datacenter)s')
            ],
            unit=Unit.OPERATIONS_PER_SECOND,
            stacking=Stacking.STACKED,
            width=WIDTH_FULL // 2)
        yield Graph(
            ...
```

An interesting aspect of Promenade is that it can take a hierarchy of Prometheus servers into account, which can also be declared as Python code. In the code below, we first declare a DAG (Directed Acyclic Graph) of how labels on metrics are related to each other (e.g., all metrics with the same “datacenter” label value will also have the same “country” label value). We then declare a hierarchy of Prometheus server objects for which we want to generate configuration files.

```
label_implications = LabelImplications({
    ('instance', 'customer'),
    ('instance', 'rack'),
    ('rack', 'datacenter'),
    ('datacenter', 'country'),
    ...
})

local = ScrapingRecordingServer(label_implications)
global = CompositeRecordingServer(
    label_implications, 'datacenter', {local}, ...)
```

When combining the Python code for our dashboard with this hierarchy, Promenade will automatically create the following recording rule on the local Prometheus instance:

```
datacenter:unbound_queries:rate5m =
    sum(rate(unbound_queries_total[5m]))
    by (datacenter, country)
```

It will also generate a configuration file for the global Prometheus instance, so that it will scrape `datacenter:unbound_queries:rate5m` from all of the local instances, which allows Grafana to access them.

At Kumina, we are currently migrating all of our existing Prometheus and Grafana setups to be built on top of Promenade, which is why the design of Promenade is still being tweaked to meet our requirements. We are planning on releasing it on our company's GitHub page (<https://github.com/kumina>) as soon as the code has stabilized, so stay tuned!

Wrapping Up

We hope that this article gives a good impression of how easy it is to get started with using Prometheus to monitor your systems. At Kumina, we have been happy users of Prometheus for about a year now. It is a robust, flexible, and extensible monitoring system, having a healthy ecosystem of both developers and users. Within the next couple of months, we will see the release of Prometheus 2.0, which will include lots of new features. At Kumina, we are most excited about the redesign of the storage layer, which will allow us to collect millions of samples per minute on servers with commodity hardware.

Since 2016, the Prometheus team has organized an annual conference called PromCon (<https://promcon.io/>). This year it took place in Munich, Germany, on August 17–18. If you are interested in knowing more about some of the latest developments taking place in the Prometheus project, be sure to check out the conference's website, as recordings of all of the talks are publicly available. ●

ED SCHOUTEN is the lead software developer at Kumina, a managed services provider and consultancy firm based in Eindhoven, the Netherlands. Kumina provides companies with fully managed platforms and offers support, training, and consultancy for Prometheus and Kubernetes.

Feel free to visit our website <https://kumina.nl/> or contact us at info@kumina.nl to tell us about your project or request more information about our offerings.



Write For Us!

freeBSDTM JOURNAL



Contact Jim Maurer (jmaurer@freebsdjournal.com) with your article ideas.