WOMBATOAM

Getting Started Guide

Erlang SOLUTIONS

Table Of Contents

What is WombatOAM?

WombatOAM is an **operations and maintenance tool** for proprietary and open source **Erlang and Elixir systems**. All of the generic functionality to monitor and manage scalable, highly available systems, often only in part reimplemented from project to project, is now available in a battle-proven standalone solution. Using WombatOAM allows you to focus on your business logic, while reusing thousands of lines of code and tapping into decades of operational experience of Erlang-based systems.

WombatOAM's functionality is divided into two categories:

- **WombatOAM Monitoring**: Monitoring nodes, including collecting and displaying metrics, alarms and logs from the managed nodes. WombatOAM also contains tools that allow you to inspect and troubleshoot your system, as well as APIs that you can use to hook into your existing OAM and SAAS tools.
- **WombatOAM Orchestration**: Deploying nodes in the cloud or on specific proprietary clusters.

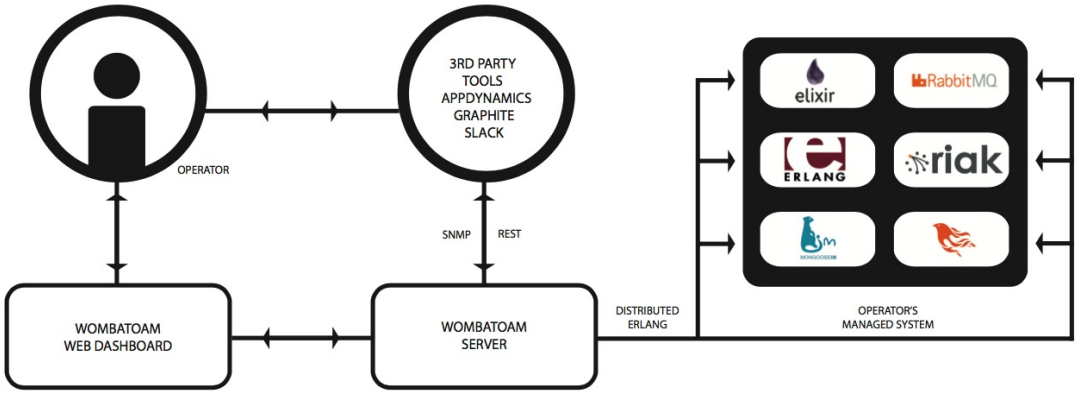
> Orchestration is currently in Beta. We are looking for users interested in working with us while we bring it to R1.

The WombatOAM server consists of one or more standalone nodes that connect to a system running on the Erlang VM using Distributed Erlang. The system consists of one or more Erlang VMs, possibly running different releases. The nodes in the system WombatOAM is connected to are called the managed nodes. They could be a proprietary application, or standalone open source applications such as CouchDB, Riak, RabbitMQ or Phoenix. Depending on the OTP applications that are running on the managed nodes, WombatOAM starts non-intrusive agents that enable it to monitor alarms and notifications, and collect information such as metrics and logs. This agent code is loaded on the fly, without having to include any proprietary WombatOAM applications in your release and without the nodes having to be restarted. WombatOAM will connect to them seamlessly – even if they have been running for years, or are running older versions of the Erlang VM.

The plugins used by WombatOAM have been optimized to reduce overheads caused by the agents monitoring the managed nodes. Overheads vary between 0.5%–1.5% of the total CPU cycles used by the Erlang VM, depending on how many applications you are monitoring and the underlying hardware, operating system, and possible virtualization layers. The performance of the business logic running in the managed nodes will barely be affected by the minimal overhead.

The WombatOAM web dashboard presents information to the operator in a manageable way. The dashboard is intended for use by DevOps teams when troubleshooting Erlang systems and for companies who have not integrated their Erlang systems to their existing OAM infrastructure. WombatOAM can also share this information with other third-party OAM tools, acting as a hub towards Nagios, Cacti, Graphite, Grafana or SAAS providers such as Splunk or PagerDuty (to mention just a few). As a result, it provides a single point of integration with the wider OAM infrastructure and SAAS providers without the need to upgrade and

manipulate production code running in the managed nodes.



- Autodiscovery of your node topology. Provide a node and its cookie, and WombatOAM will discover all existing nodes in your cluster, group them into releases (called node families), and start monitoring them. WombatOAM supports both long and short names, and can currently monitor the R14 Erlang runtime alongside later versions. Versions older than R14 can also be supported on demand.
- Anomaly detection and early warnings, presented in the form of alarms and notifications. This allows DevOps teams to address and resolve problems before they escalate and cause service disruption. Integration has been made with the SASL alarm handler and the Elarm application, forwarding alarms specific to your system. SASL and Lager logs, including crash, warning and error reports, are also forwarded, giving you access to data specific to all your clusters in one place.
- Automatic collection of more than a hundred built-in metrics from the Erlang runtime system, including different memory types, system limits, socket, port and process-specific metrics. Additional metrics retrieved from plugin modules of other supported OTP applications are also uploaded. Metrics from Folsom and Exometer are collected seamlessly if the managed nodes are using them to generate business metrics.
- Application-specific plugins that run on the managed node and send metrics, notifications and alarms to WombatOAM. A set of built-in plugins covering many standard OTP applications and popular open source ones are shipped with WombatOAM. They can be turned on and off for each individual managed node. You can also implement proprietary plugins following a simple API.
- Plugins have been implemented to monitor SASL, Lager, Folsom, Exometer, Elarm, Mnesia, OSmon and Cowboy OTP applications. For Riak, WombatOAM monitors Riak Core, key value (Bitcask and Active Anti-Entropy), replication services and Yokozuna. Many more plugins are being implemented, and you can write your own following a simple API.
- Seamless integration with your OAM infrastructure by having existing integration plugins with Graphite, Grafana, Cacti, Graylog, Splunk, Zabbix, Datadog, Nagios, Logstash, AppDynamics and PagerDuty. If you are using proprietary OAM tools or SAAS providers currently not supported, the WombatOAM architecture provides flexibility to add integration points without the need to

upgrade or restart your managed nodes. Integration happens in the WombatOAM node, reducing overhead in the managed nodes while reducing the risk of OAM-related issues escalating in the nodes managing the business logic.

- A Web Dashboard and a REST interface, the former for interactive use if you need all of the information in one place, the latter to support automation and integration with your existing tool chain and scripts.
- Visualization of the node topology and the ability to inspect specific node information, which facilitates troubleshooting. WombatOAM also acts as a point of integration of tools that allow closer inspection of what is happening in the managed nodes and facilitate troubleshooting, without the need to access the Erlang nodes and shell.
- The dashboard can plot both historic and live metrics, showing you memory usage in real time, or helping you detect spikes no one would have noticed otherwise. Multiple metrics can be shown on the same graph for comparison purposes. Besides numeric metrics such as counters and gauges, WombatOAM also supports meters, spirals and histograms.
- Deployment of Erlang nodes in heterogeneous clouds or on specified machines that scale to tens of thousands of nodes with no single point of failure. WombatOAM orchestration and monitoring has been tested on a cluster of 10,000 Erlang VMs, but is linearly scalable beyond that.

Getting started with WombatOAM

Prerequisites

Before installing WombatOAM, ensure that the machine meets the prerequisites. The following is required before installing WombatOAM:

- A UNIX operating system (WombatOAM has been tested on Linux and OS X)
- Erlang 17.5 or later installed for running WombatOAM, the managed nodes can be as old as R14.
- OpenSSL and libcrypto (already present on Mac OS X)

Installing WombatOAM

To install WombatOAM on your computer, retrieve the current WombatOAM package and run the installation script.

1. Extract the WombatOAM package and go inside directory:

   ```
   tar xfz wombat-[VERSION].tar.gz
   ```

   ```
   cd wombat-[VERSION]
   ```

2. Execute the following command:

   ```
   ./wombat install
   ```

If you already have a WombatOAM installation follow the instructions in the "Upgrading WombatOAM" chapter.

Installing WombatOAM on Windows

Currently on Windows only running it inside Docker is supported. Please get in touch with us if you need native Windows support.

Upgrading WombatOAM

To upgrade an existing installation of a previous version of WombatOAM:

1. Extract the WombatOAM package:

   ```
   tar xfz wombat-[VERSION].tar.gz
   ```

   ```
   cd wombat-[VERSION]
   ```

2. If you would like to use a new license key, copy it to the new WombatOAM installation's root directory. (If the version of the old WombatOAM installation is 2.0.0-rc1 or earlier, this is necessary.) If you would like to use the same license key as in the old installation, then skip this step.

   ```
   cp .../my.lic_key .
   ```

3. Execute the following command:

   ```
   ./wombat upgrade
   ```

   You will have to input the previous Wombat installation directory.

4. The script will perform the upgrade and after running it you can start WombatOAM.

The command `./wombat upgrade` will preserve your collected data and custom configuration (in your old `_build/default/rel/wombat/files/wombat.config` file). It will copy the old license from the old installation. Backups are created to the `backups` directory from the current installation.

The WombatOAM web dashboard

To access the web dashboard, go to `http://localhost:8080` in your browser. Log in with the following details:

- **Name:** admin
- **Password:** admin

Running WombatOAM

The command `./wombat start` will start WombatOAM automatically. You can use the following commands to manipulate WombatOAM:

```
./wombat start  # start WombatOAM
./wombat stop   # stop WombatOAM
./wombat start_script # the OTP release start script
```

The env variable WOMBAT_LICENSE can be used by WombatOAM to read the license text, this is useful when doing continuous redeployments.

To check whether WombatOAM is already running, navigate your browser to the WombatOAM web dashboard on `http://localhost:8080`.

Running inside containers using Docker

You should have access already to the WombatOAM Docker image, if not, please get in touch with us.

1. Import the image locally by running the following command, it will show you the tag of the image which is used to start the container:
   ```
   docker load --input wombat_<version>.image.tar
   ```
2. Start the container and expose the dashboard port. By defining an environment variable we can tell Wombat's webserver to listen on all interfaces:
   ```
   docker run -it -p 8080:8080 -e WOMBAT_NODENAME=wombat@0.0.0.0 wombat:<version>
   ```
3. Navigate to `localhost:8080` in your browser

If you start WombatOAM this way, your container will have no access to other Erlang nodes running on your machine, only the `wombat@127.0.0.1` node will be accessible. We suggests setting up a docker-compose.yml file including your application. For testing purposes you can connect to the container using `docker exec -it $CONTAINER_ID bash` and start an Erlang node manually. Examples can be found in the next chapters.

Please refer to the manual's "Configuring WombatOAM for servers and containers" chapter for more information.

Running on Amazon AWS

We build AMIs from Wombat. Please get in touch at wombat@erlang-solutions.com if you have no access to them.

Please check the Wombat-Manual.pdf or the html documentation "Using the official AMI containing WombatOAM" section.

Using WombatOAM

To see how WombatOAM handles nodes, you can let WombatOAM monitor itself:

1. Select the **Topology** tab and click **Add Node**.
2. Enter the following:
   - **Node name**: `wombat@127.0.0.1`
   - **Cookie**: `wombat`
3. Click the **Add node** button.

The WombatOAM node should come up in a few seconds. If it doesn't, check the following log file for details:
`_build/default/rel/wombat/log/wombat@127.0.0.1/wombat.log`

The following chapter will help you to get to know WombatOAM.

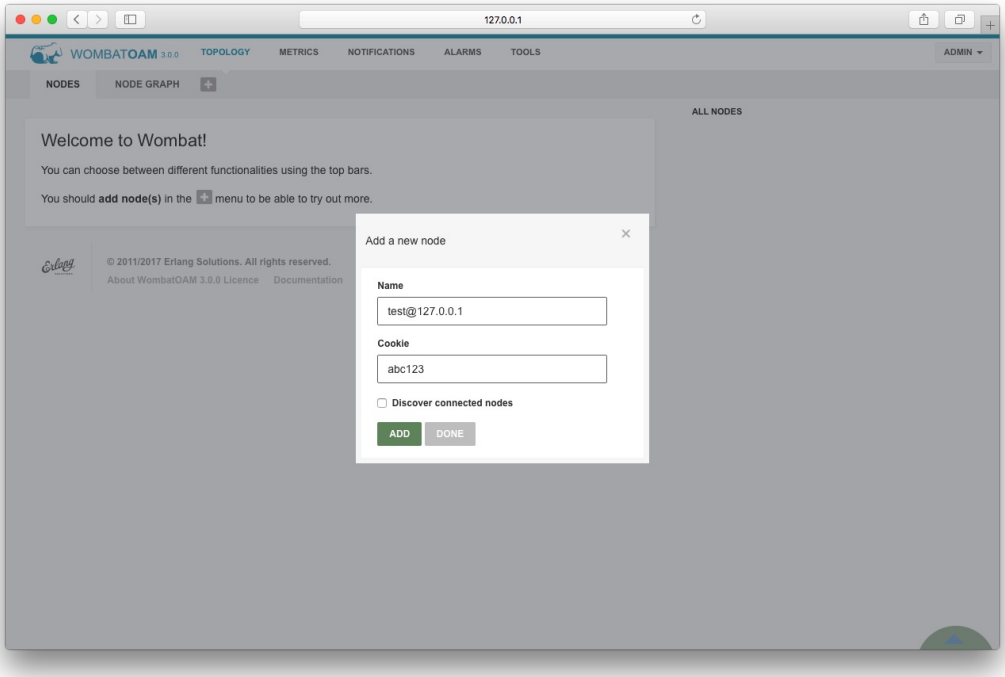
Exploring and understanding WombatOAM

"I have installed WombatOAM, now what?"

This chapter gives you a walkthrough of WombatOAM, demonstrating some of the issues you might encounter in your Erlang and Elixir based systems. It highlights how WombatOAM provides full visibility of what is happening, helping you with pre-emptive support in order to avoid outages, and also with post-mortem debugging to quickly and efficiently find and fix problems, ensuring they do not happen again. Besides, WombatOAM is handy for troubleshooting, online diagnosing and resolving problems and incidents on the fly. This walkthrough should take you about 30 minutes, and is recommended for anyone evaluating or trying to get the most out of WombatOAM.

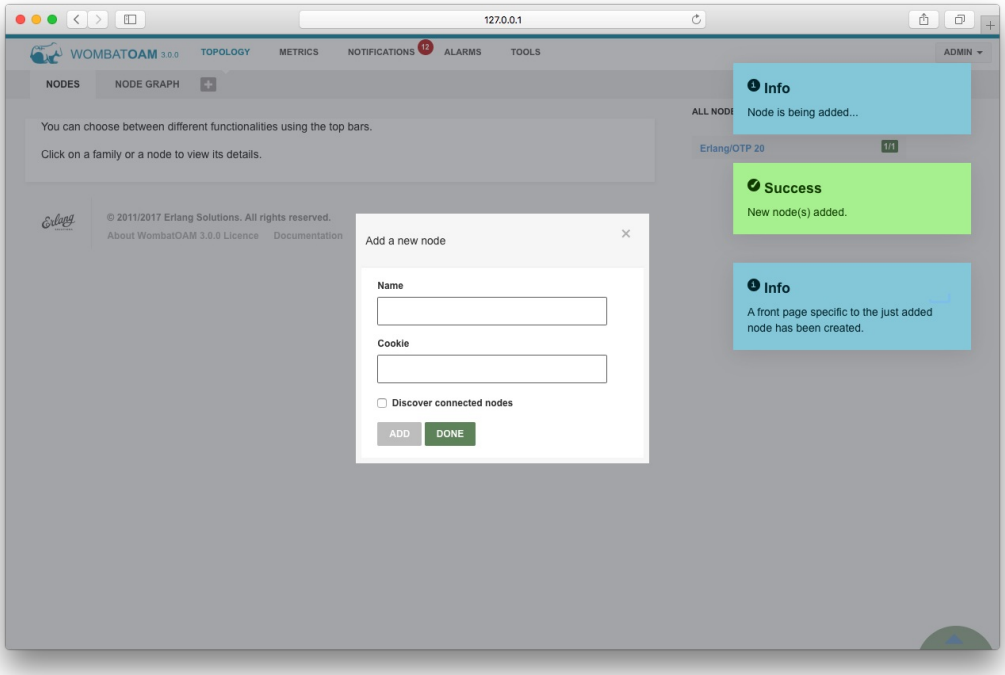
Basics

To get started, go to the WombatOAM dashboard and click **Topology**. Click **Add node**, and enter a node name and cookie.



If you are using distributed Erlang in your cluster, select **Discover connected nodes** to automatically add nodes meshed with the one you are adding, to save you the effort of doing it later.

When you have entered the node name and cookie, click the **Add node** button. This will connect WombatOAM to your cluster, and allow it to start monitoring the nodes.



If you are using multiple cookies, add a node from each cluster using the specific cookie. Note that WombatOAM allows you to connect to nodes with long and short names.

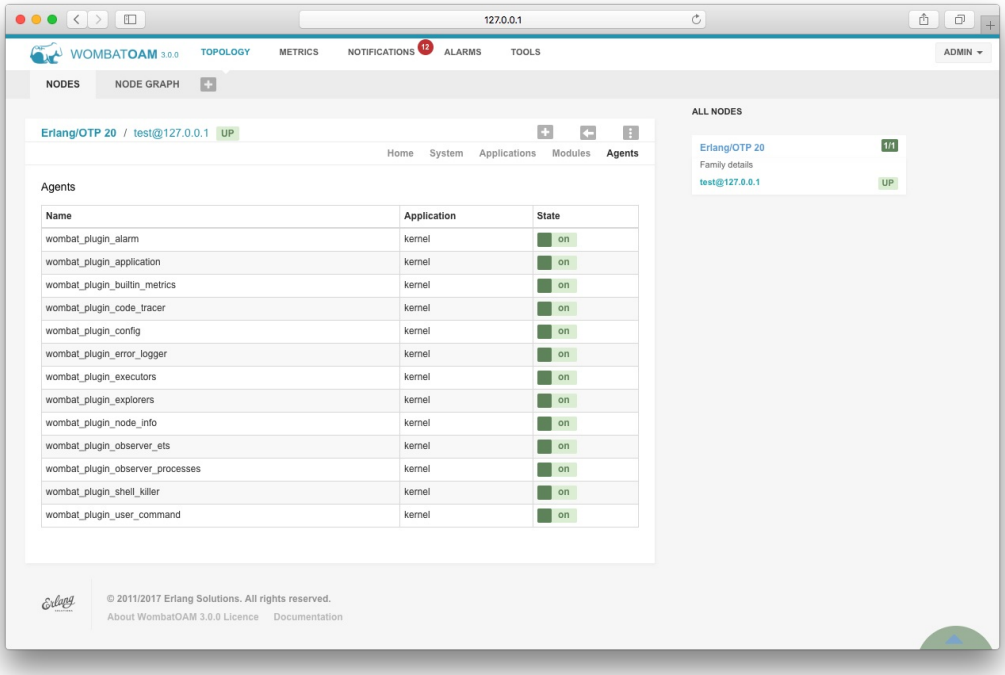
During this walkthrough, you will experiment and simulate a few system issues. If you are connected to a production system and are worried about the impact of this, you can also start and connect to a standalone node. To do this, enter the following command:

```
erl -sname test -setcookie abc123
```

Once you have started the node, add it to WombatOAM (the node name is "test" and the cookie is "abc123"). Ensure that you remain connected to the node in the shell. If you need to stop and restart the node later, you can use the same command provided above.

Depending on what applications you are running, WombatOAM will start agents that collect metrics, alarms, notifications, other important information about your nodes and also provide tools. Tools are not metrics, notifications or alarms, but aid devops with their daily tasks. Have a look at what these agents do, as you can easily write your own plugins introducing new agents.

Select the node to see information about the system, modules, applications and the agents that are running.



The agents that come with WombatOAM do the work of collecting and generating metrics, notifications and alarms. Also, they enable retrieving live information about running systems (such as the state of a gen_server), recover from outages for instance by forcing the GC to free up memory, or fix misconfiguration issues by changing configuration parameters. A basic installation (excluding the application-specific agents listed below) will include:

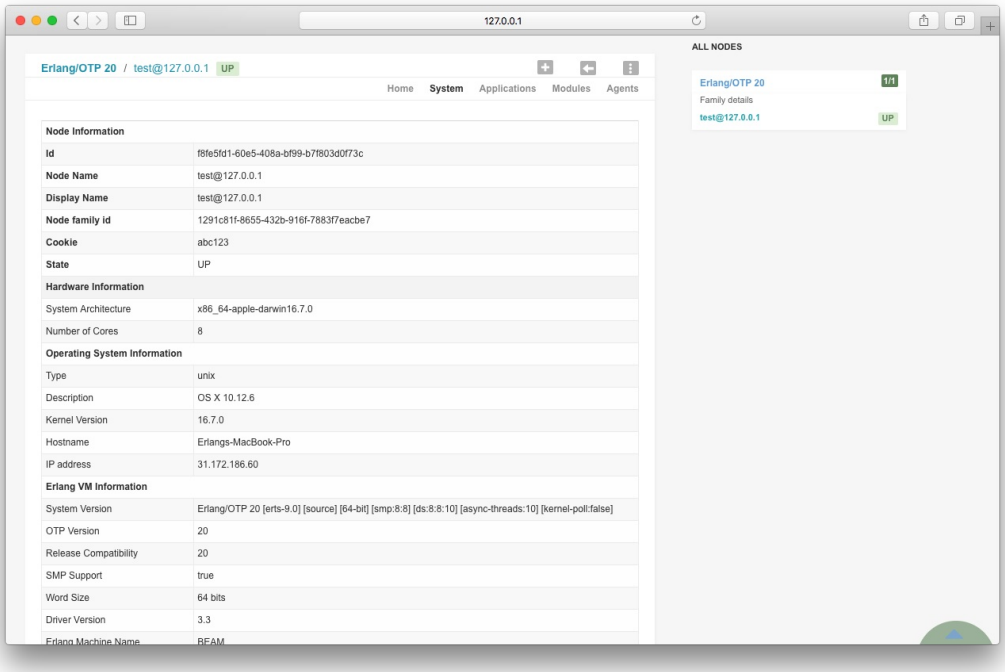
- More than 100 metrics and the automatic collection of Exometer and Folsom metrics
- Automatic collection of Lager and SASL logs
- Over 25 built-in alarms and automatic collection of elarm and SASL alarms
- Configuration management
- Etop-like process listing
- ETS table viewer
- Various process inspectors
- Executor for user commands and other actions such as initiating garbage collection

We currently have agents supporting the following applications:

- Kernel
- OS_Mon
- SASL
- Lager
- Exometer
- Folsom
- Cowboy (versions 0.x, 1.x)
- Phoenix
- Ecto
- Poolboy
- MongooseIM
- RabbitMQ
- Mnesia
- Riak Core
- Riak KV
- Riak Multi-Datacenter Replication
- Yokozuna
- SumoDB

- Yaws

Explore some more by clicking the **Topology** tab. Here you can view comprehensive system, application and module information.



This information can be useful for troubleshooting your system. For example:

- Module information: one of your nodes may be using an older version of a module
- Application information: an application running on one of your nodes may be using a wrong value (such as an old IP address)
- System information: check which release is currently used by the node

To see how WombatOAM monitors your systems, let's use the shell to simulate problems that might occur. If you have a node in your system that you can allow to crash, connect to it. Otherwise, use a standalone test node.

If you have started a separate node, add it to WombatOAM. Ensure that you are connected to the node in the shell, then enter the following command:

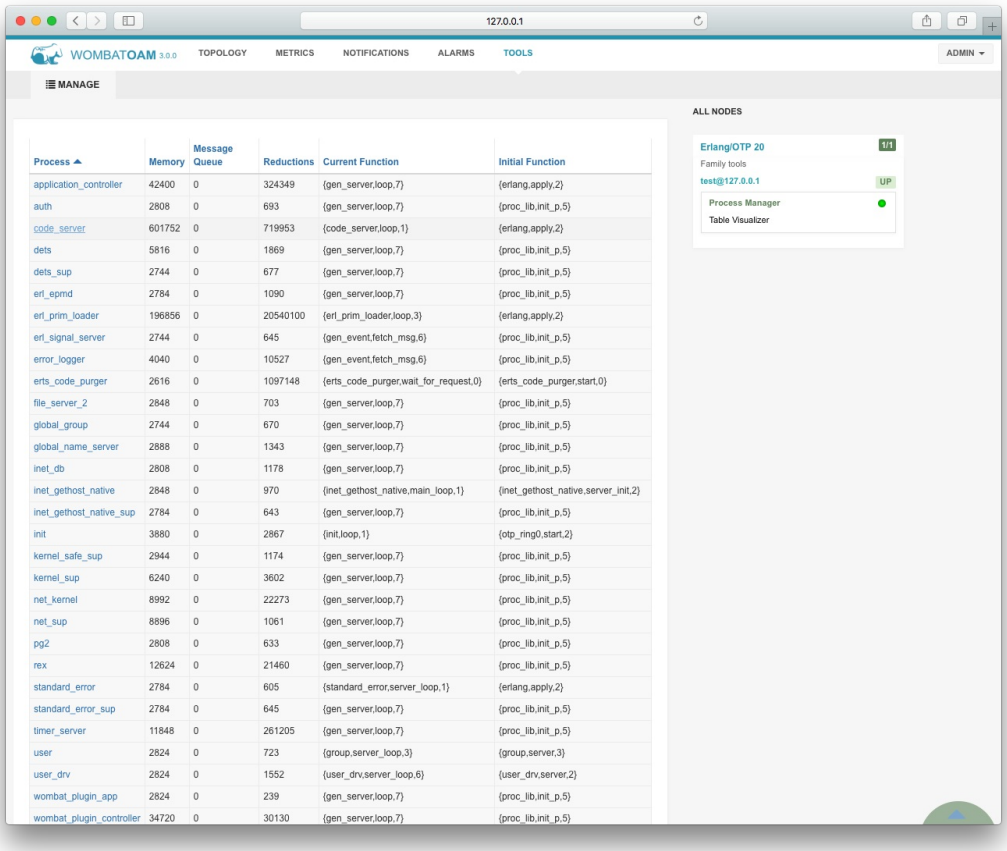
```
1  put(sample_key, value),
2  [ self() ! {count, Int} || Int <- lists:seq(1,100000) ].
```

This will associate `value` with `sample_key` in the shell process's local store called the "Process Dictionary" and then send 100,000 messages to the shell process, simulating producers sending requests at a faster rate than the consumers can handle. Long message queues are often a symptom of a bottleneck and result in the degradation of throughput, with the system eventually running out of memory.

First, retrieve live information about your running processes. In

WombatOAM, go to **Tools**. Select your test node and then open the **Process Manager** panel.

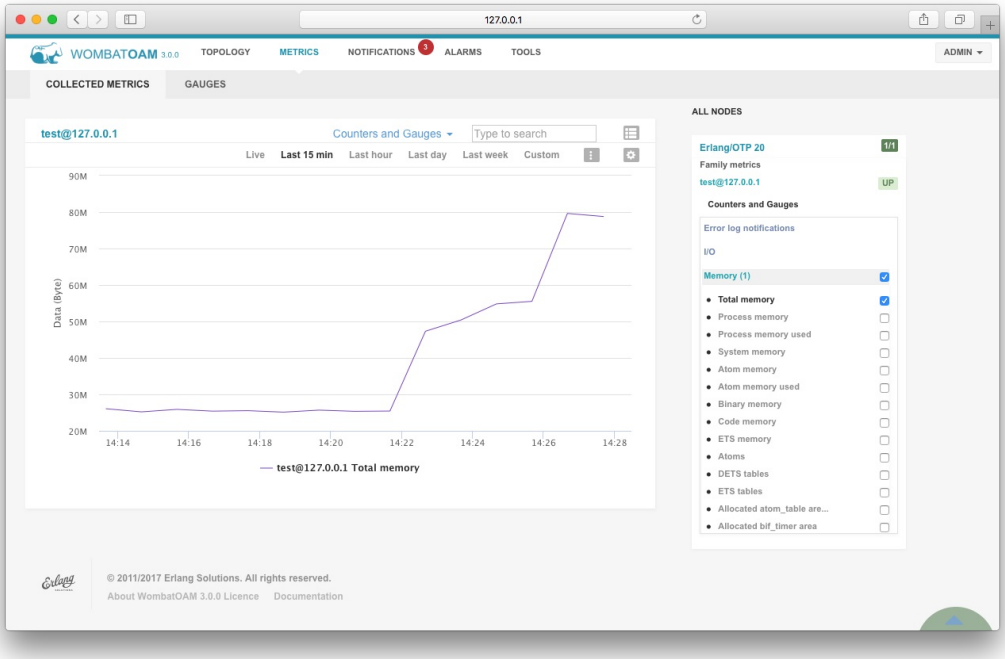
Notice that the shell process has the largest message queue.



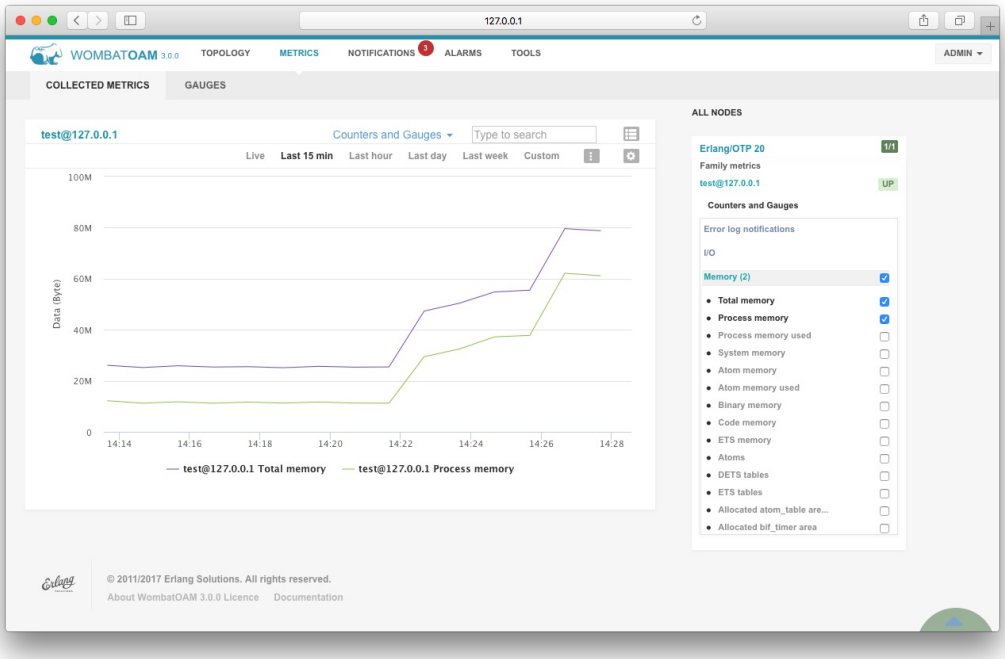
By clicking on the Pid of the shell process, various information, such as process info, process messages, process dictionary, process state, process stack trace) can be retrieved about this process. These come in handy when your system doesn't behave as expected and you want to find the root cause.

Next, go to **Metrics**, select the node to which you issued the command, select **Memory**, and then select **Total memory** to view that metric as a graph. (Note that metrics are collected once a minute, so you may have to wait briefly for the graph to update.)

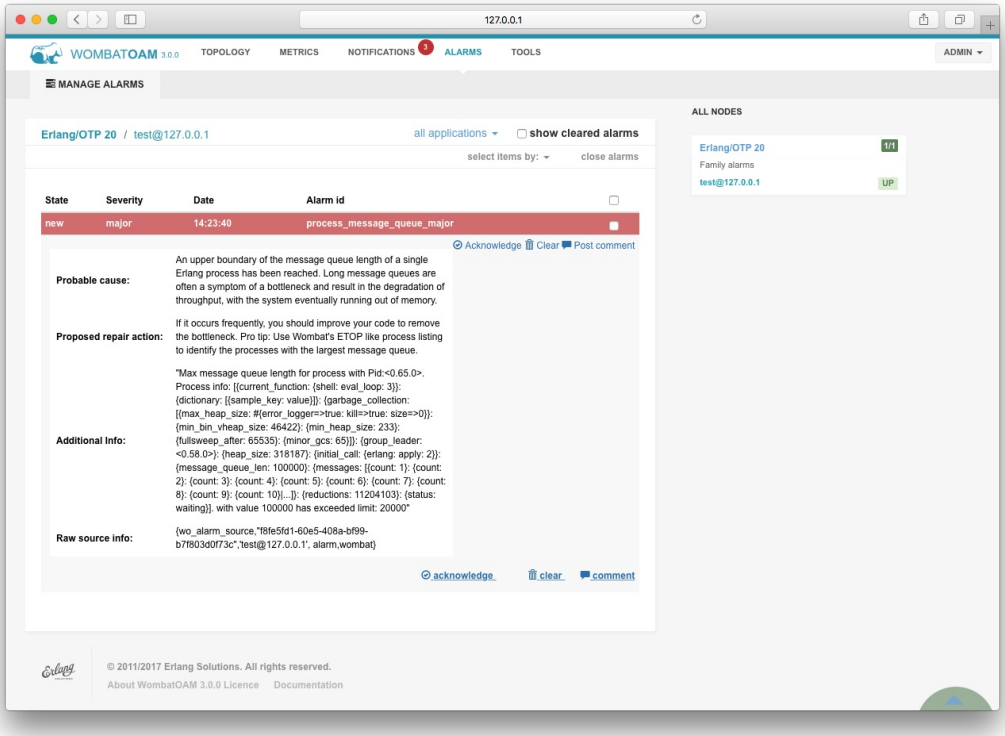
Notice how sending the messages affected the total memory increase.



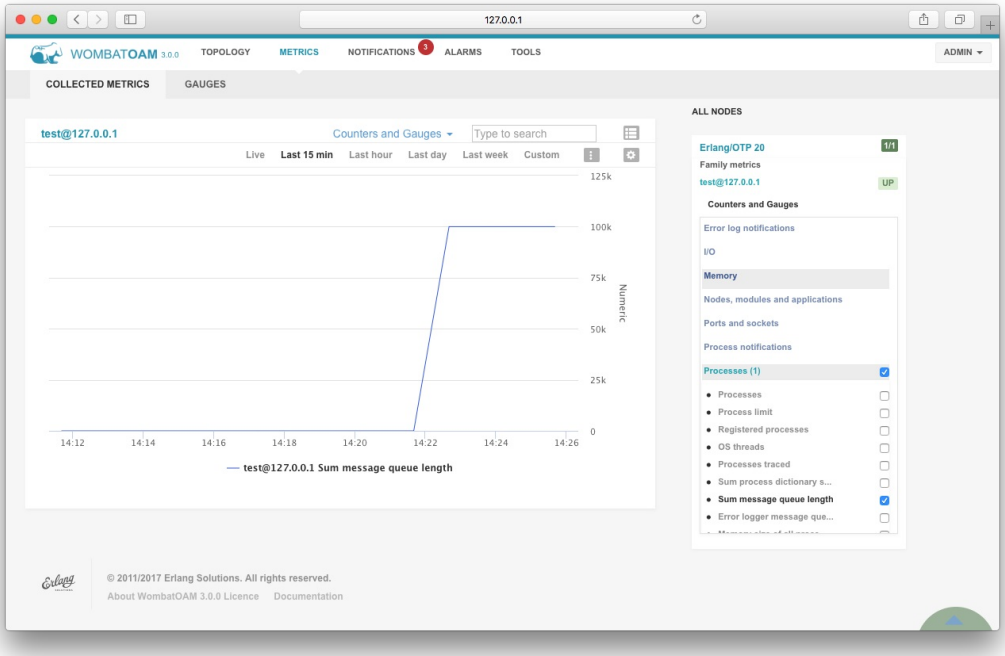
Select **Process memory** used to see how process memory affected the total memory.



Select the **Alarms** tab. Note the new **process_message_queue_major** alarm with severity **major** that will appear.



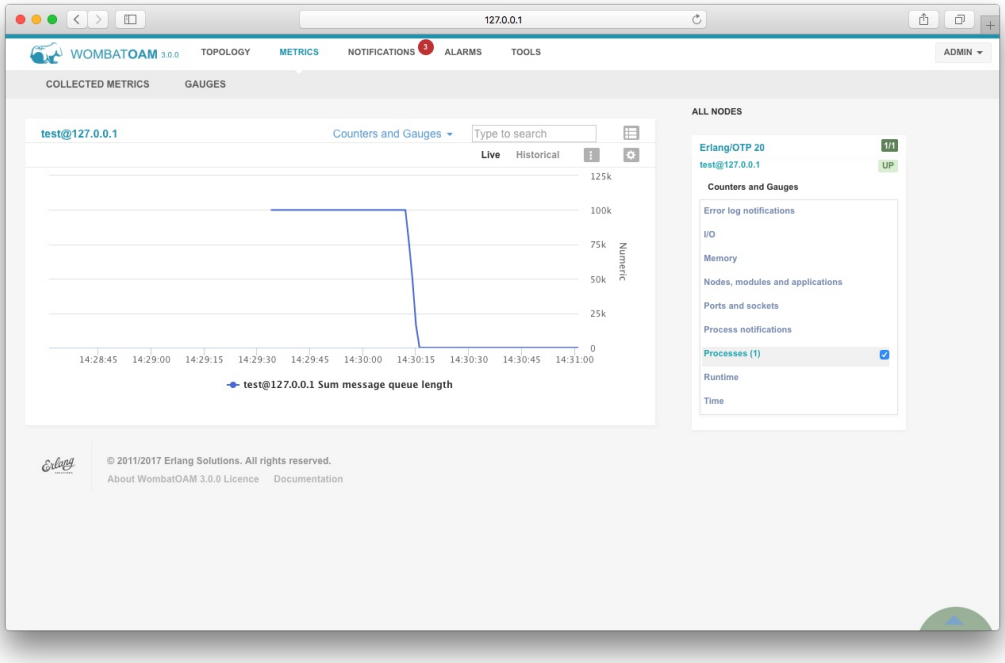
Go back to Metrics and clear all the metrics (click on the "trash" button at the upper right of the graph). Select the test node and click **Processes → Sum message queue length** to see all the messages you sent to the shell.



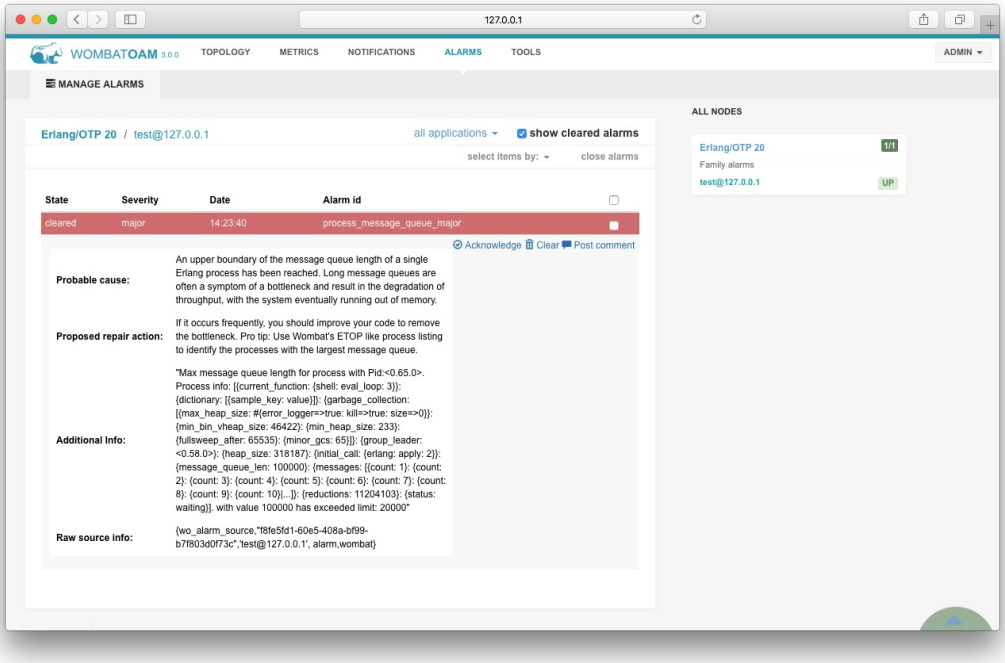
Next, take it a step forward, and click **Live Metrics**. Again select the **Sum message queue length** metric. While you keep monitoring the message queue length in real time, enter the following in the shell:

```
1  flush().
```

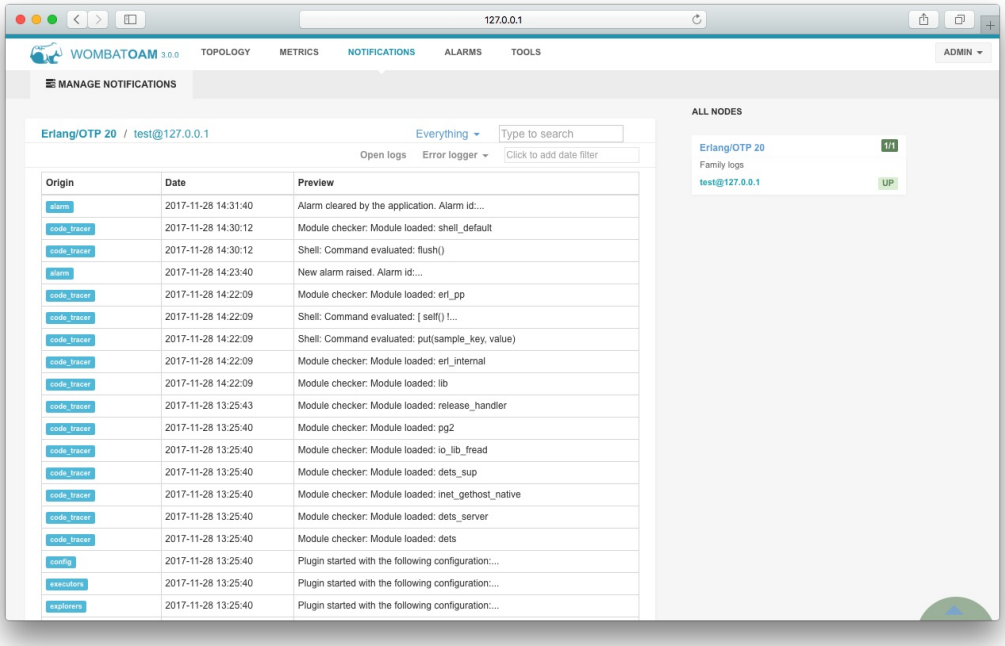
Notice how the graph drops as the queue is being cleared.



When the message queue is cleared, click on **Alarms**. You should see that the alarm has now been cleared. (You may need to give it some time to clear, as checks are made on average once a minute.)



Go to the **Notifications** tab, and you should see notifications from the code tracer and the alarm handler.



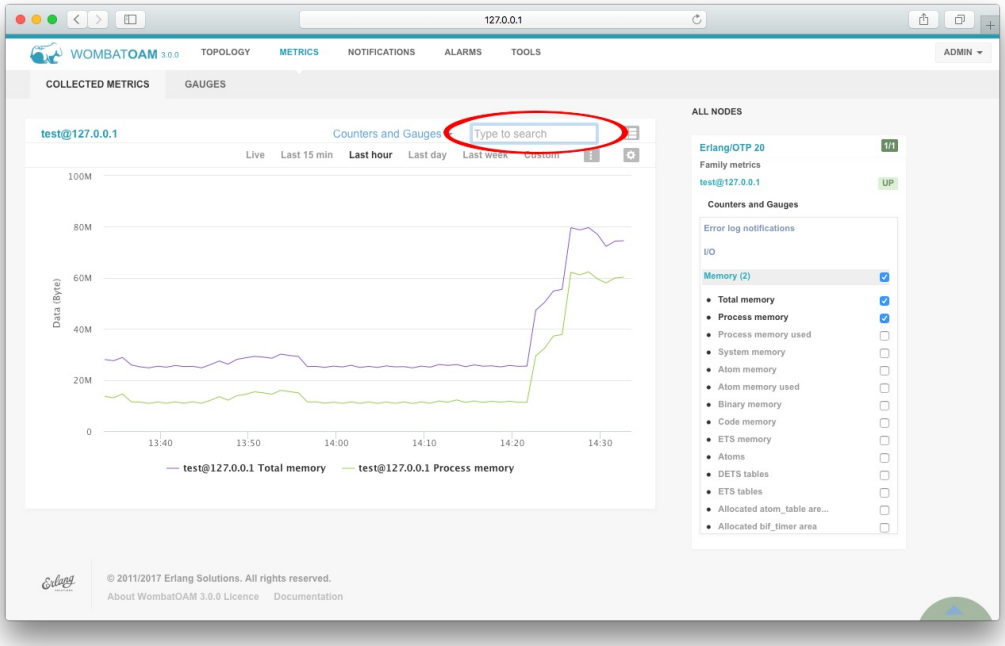
All alarms raised and cleared are logged under Notifications, along with all shell interactions, modules loaded and purged, and so forth. You might not be sitting in front of your monitor and miss an alarm being raised and cleared – maybe missing that a message queue grew to 100,000 before being handled! You should check alarm logs regularly, and address them if necessary. Next time, it might happen that the queue length goes far beyond 100,000, causing the node to slow down and eventually crash.

Following these steps should show you how using WombatOAM might let you uncover problems that would otherwise have remained unnoticed. The following story from a WombatOAM customer attests to the importance of this kind of pre-emptive support:

> "In one WombatOAM installation, all seemed to run as normal until we looked at the alarm logs. We noticed that twice in three days, three processes had reached a message queue length of 100,000, which got cleared without causing any issues. We narrowed down the cause to the process acting as a cache when the client connection was dropped for a longer time period. This became only visible in live systems, as client connectivity in the test plant worked well. Changes were immediately needed, as loss of network connectivity or a firewall misconfiguration between the perimeter network and that machine would have caused the connection to all clients to be dropped, resulting in the node running out of memory."

Further exploration

Below are some more things you can try out to get to know WombatOAM's functionality, focusing on alarms and notifications. Try to keep an eye on relevant metrics – you can easily search for corresponding metrics by typing into the search box in the list of metrics.



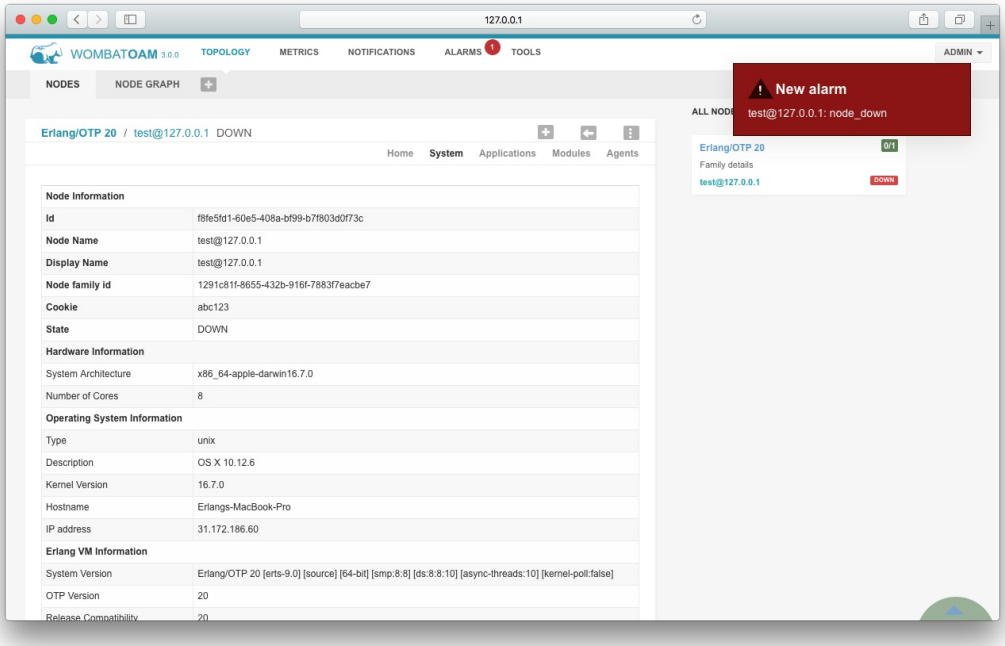
Also notice the badges on the Notifications and Alarms tabs – the numbers increment as new items are added. If new alarms are added while you are on the Alarms tab, or new notifications arrive while you are the Notifications tab, there will be a message above the list telling you that, with a link that you can click to show them. This prevents you losing your current view in cases where new entries arrive at a fast rate. There are navigation controls below the list to view older entries.

Alarms

Node down

The most basic alarm is **node_down**. To trigger this, you just need to stop the node that you are running and monitoring. If you started the node in the shell, you can press Ctrl-C to stop the node, or enter the following:

```
1  init:stop().
```



WombatOAM will show that the node is down, and also show an alarm (look out for the popup notification) after a brief moment. After this, restart the node. Again, WombatOAM will automatically notice (within 30 seconds) that it is up again, and the alarm will be cleared.

Hitting system limits

Filling the atom table

Converting values into atoms without any control can fill the atom table. To try this out, enter the following command in your Erlang shell:

```
1  [ list_to_atom("atom_" ++ integer_to_list(I)) || I <- lists
   :seq(1, 964689) ].
```

You should get the following alarm: **atom_limit_major**

This alarm is may be an indication of unsatisfactory code that dynamically turns strings into atoms. Since the atom table is not garbage-collected, the only way to recover from this situation is to restart the node.

Converting user inputs into atoms can also easily hit the system limit, since the atom table can only grow. Usually the table grows slowly, so the node doesn't crash immediately, but only after a long period of time (for instance, after two months).

You should restart the node after trying this out.

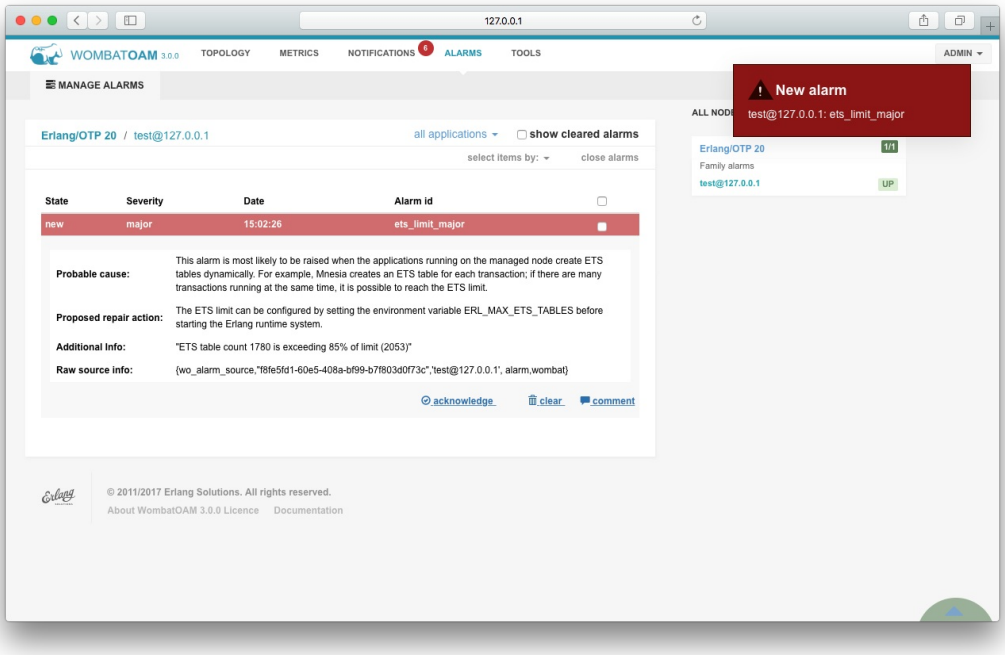
Creating too many ETS tables

Enter the following command in your Erlang shell:

```
1  [ ets:new(list_to_atom("_ets_table_" ++ integer_to_list(I))
2  ,
3            [public, named_table])
4    || I <- lists:seq(1, 1750) ],
```

```
ets:insert('_ets_table_1', {v1, v2, v3}).
```

You should get the following alarm: **ets_limit_major**



If you don't get the alarm, it may be because your system limits differ from the defaults. Try this alternative instead, as it first determines the system limits:

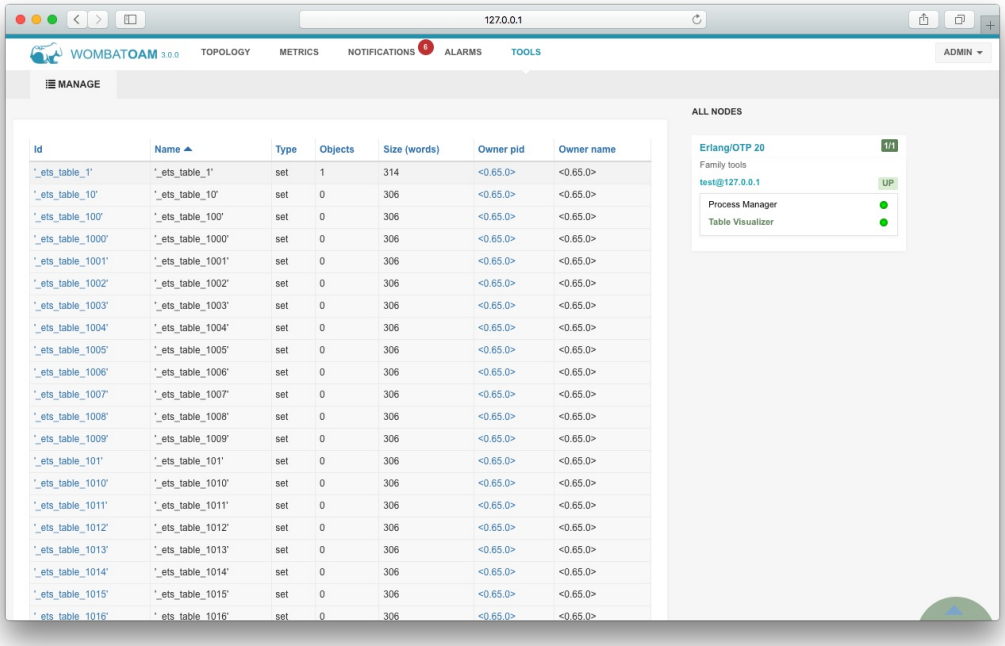
```
ETSLimit = try erlang:system_info(ets_limit) of
        Limit -> round(Limit * 0.92)
    catch
        error:badarg -> 1900
    end.

[ ets:new(list_to_atom("_ets_table_" ++ integer_to_list(I)
),
          [public, named_table])
  || I <- lists:seq(1, ETSLimit) ],
ets:insert('_ets_table_1', {v1, v2, v3}).
```

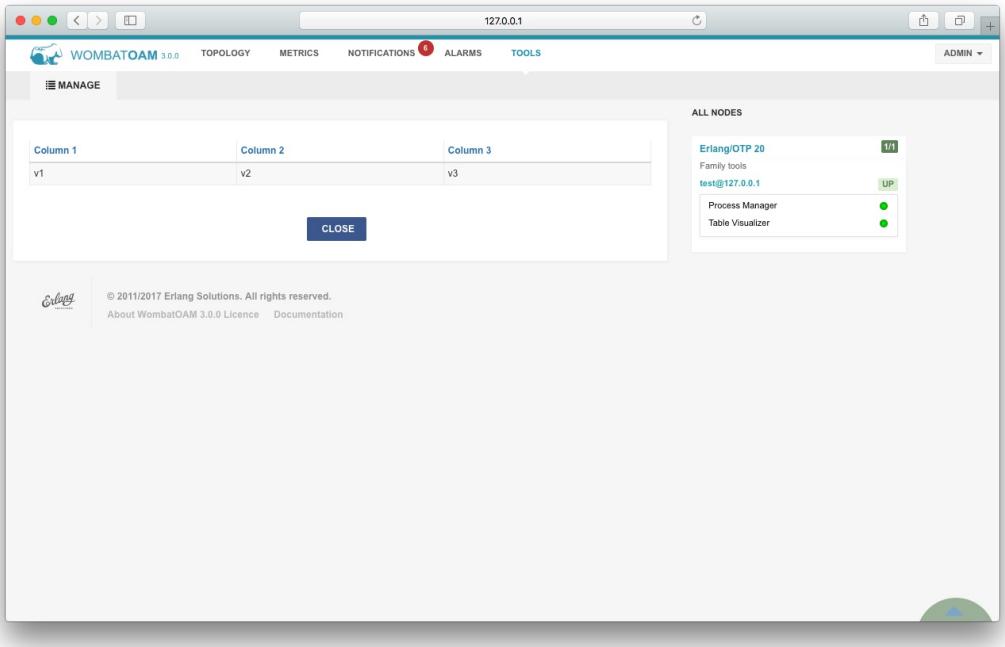
Hitting the limit of the number of ETS tables can happen quite easily, as there is no automatic garbage collection for tables. Even if there are no references to a table from any process, it will not automatically be destroyed unless the owner process terminates. Imagine a permanent server process creating new ETS tables for new sessions. If there is a spike in requests, you might hit the limit. This alarm is also seen when executing many simultaneous mnesia transactions, as each transaction creates a dynamic ETS table.

Now, it is a good time to explore another service provided by WombatOAM. Go to **Tools** and select the node to which you issued the command. Open the **Table Visualizer** panel, change the value of the *Order by column* input field to be *name* and submit the request.

A listings will appear showing the ETS tables with their properties.



Click the *'_ets_table_1'* link in the first row and view its content by clicking *View content* in the local menu that appeared. Notice that the displayed content is the same what we inserted into this table.



Go back to the **Table Visualizer** tab, and click the pid of the owner process which created these ETS tables. A local menu with the same content we saw at Etop has appeared. Click *Terminate process* that will kill your shell process and therefore all ETS tables created by this process will be deleted. Within a minute, the corresponding alarm will be cleared automatically.

Process limit

The maximum number of simultaneously alive Erlang processes is by default 32,768 (this limit can be configured at startup). To trigger an alarm where the number of processes approaches this limit, enter the

following commands into your shell:

```
Shell = self().

[ spawn(fun()-> receive after 120000 -> Shell ! I end end)
|| I <- lists:seq(1, 30147) ].
```

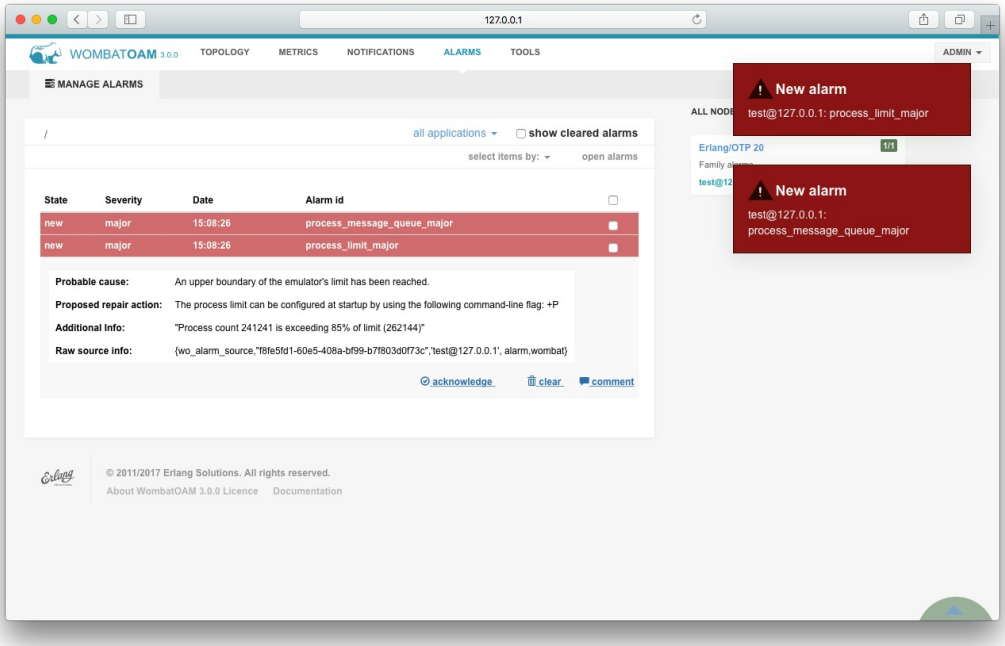
If you don't get the alarm, it may be because your system limits differ from the defaults. Try the following alternative, which determines the system limits first:

```
ProcessLimit = try erlang:system_info(process_limit) of
        Limit -> round(Limit * 0.92)
    catch
        error:badarg -> round(32768 * 0.92)
    end.

Shell = self().

[ spawn(fun()-> receive after 120000 -> Shell ! I end end)
|| I <- lists:seq(1, ProcessLimit) ].
```

Within about a minute, the following alarm should be raised: **process_limit_major**



Spawning a process is very easy in Erlang, and serving each request by a new process increases the system's throughput. However, the system limit will be reached if the processes can't terminate as they are waiting for a resource, or the number of arriving requests is much larger than expected.

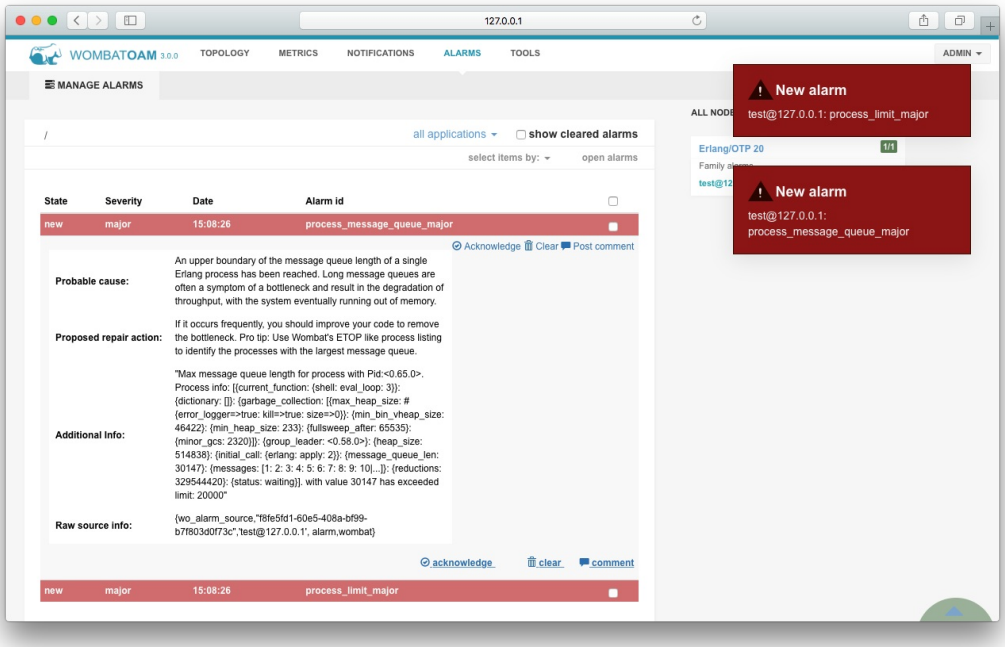
Had this alarm been monitored, we would probably have gotten an early warning that the following node's process limit had to be reconfigured:

> "We were supporting a really old installation of Erlang where the

processes limit was set to the default value of a few hundred thousand. A firewall misconfiguration caused all of the client connections to fail. The connectivity towards the node was down for five minutes, resulting in more and more users hitting the retry button. When the firewall configuration was fixed, one of the front-end nodes managing the connectivity was hit with so many requests, reaching the process limit and terminating the node. This caused all the clients to reattempt connecting to the remaining nodes, resulting in a cascading failure where all the nodes went down one by one."

Long message queue

After triggering the previous alarm (process_limit), wait 2–3 minutes. Another alarm should be raised: **process_message_queue_major**



This alarm means that an upper boundary of the message queue length of a single Erlang process has been reached. Large message queues are early warn signs. Generally speaking, a large message queue can indicate that a) your system is not totally well-balanced; b) new tasks arrive to a process in bursts; or c) a process is being stalled because it is waiting for something. If you see no reason for large message queues, or the alarm isn't cleared automatically, you should investigate the issue further to avoid possible outages.

After this, you should clear the message queue alarm. Enter the following in the shell:

```
1  flush().
```

Learning more

To find out more about memory and system limits, see the following official Erlang documentation: http://www.erlang.org/doc/efficiency_guide/advanced.html

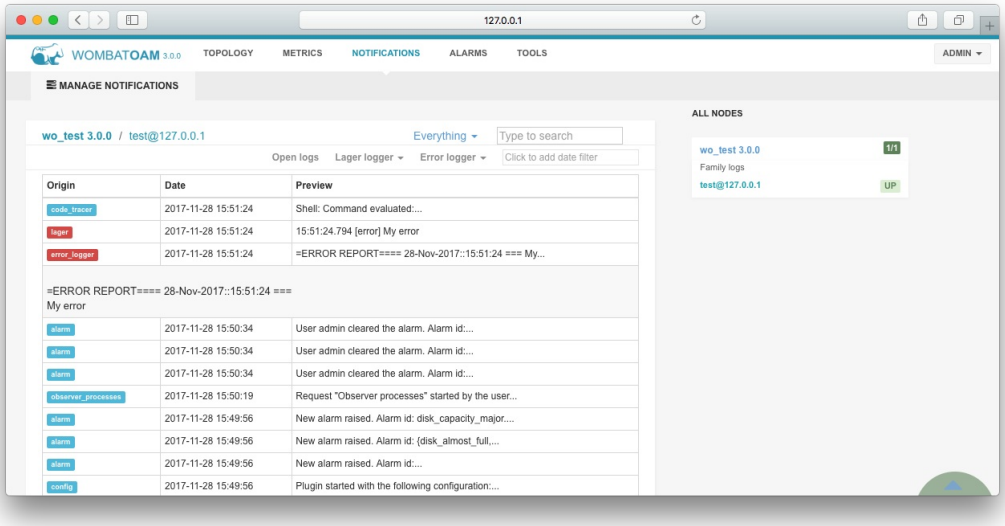
Notifications

Log entries

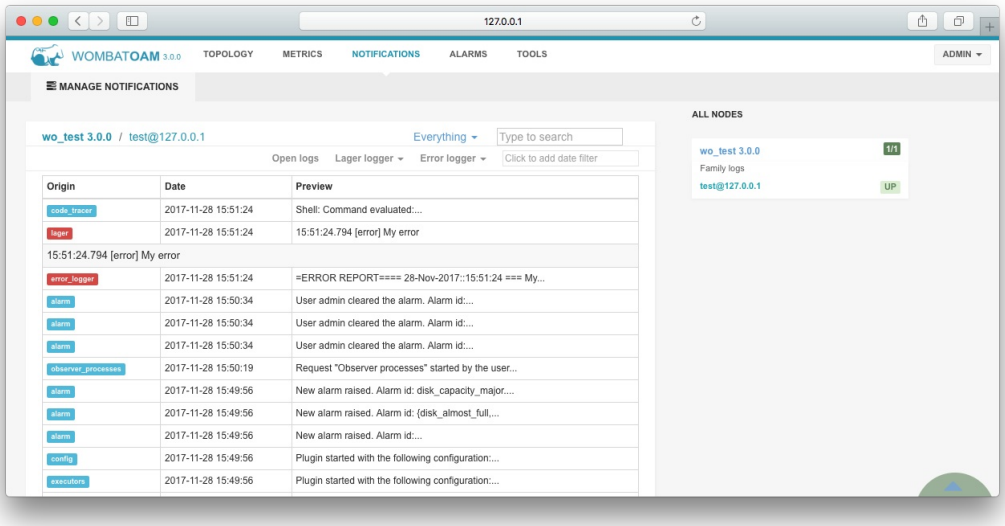
WombatOAM can aggregate errors and warnings coming from logging applications such as SASL or Lager. You can trigger SASL log entries to see how WombatOAM responds and shows you such events. In the node you are running, enter the following:

```
1  error_logger:error_msg("My error").
```

You will see the **error_logger** notification in the list of notifications for your node:



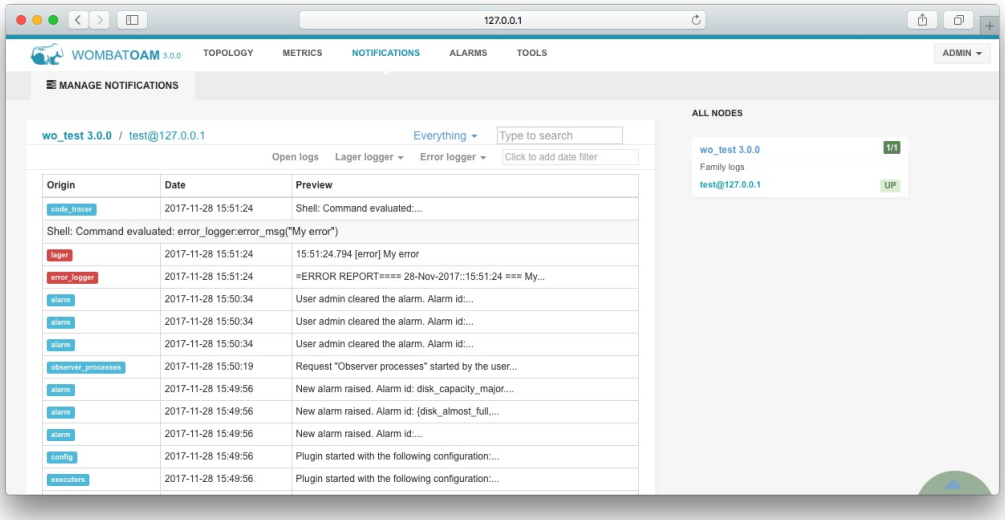
If Lager is also running on your node, you will see the **lager** notification, too.



Shell commands

As you try out the alarms and notifications suggested in this walkthrough, notice the commands executed in the shell are logged

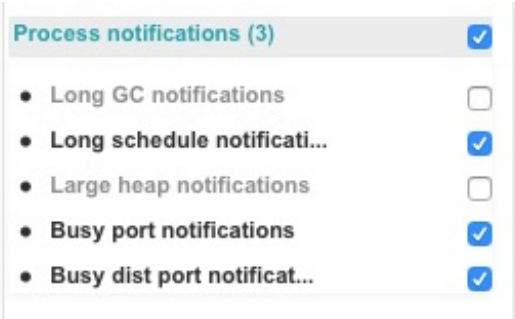
under Notifications. For example, the previous command (for the error_logger entry) resulted in the following:



This can be useful simply as a history of the commands you entered; it can also show you what other people may have done when you are troubleshooting anomalies.

Enabling System Monitor notifications

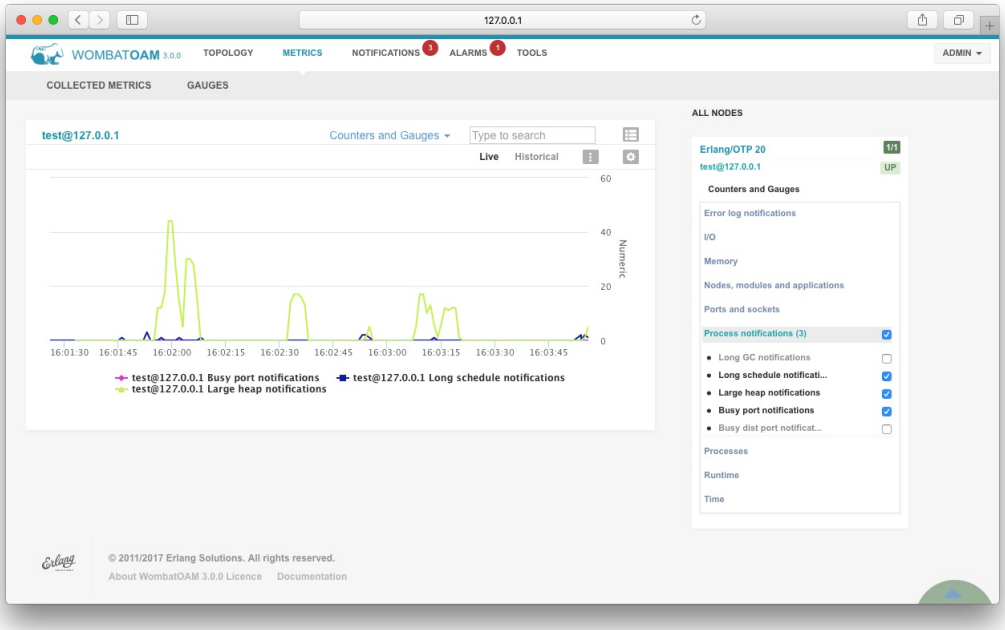
Go to **Live Metrics**, select **Process notifications**, and then select the **Busy port**, **Large heap** and **Long schedule** notifications. (You can select the others as well, if you want to.)



In the shell, type the following, and execute it 5 times:

```
[ self() ! [ random:uniform(25)+97 || _CharPerMsg <- lists:
seq(1,10000) ]
     || _NumOfMsgs <- lists:seq(1,200) ].
```

This sends 200 large strings to the shell process (and the shell will show an impressive block of random text characters). You should see the large heap and long schedule notification counters increase – these are triggered as a result of the process blocking the scheduler, meaning it has been running uninterrupted for a longer time than expected. This command will also increase the heap size of the shell.



System notifications are important to monitor, as they affect the soft real-time properties of your system. If a process spends too long holding a resource, be it the port or a scheduler, the counters are incremented. Counters exist for:

- **Large heap**: Triggered because a process is consuming a large amount of memory. A process on its own will probably not affect the system, but many of them running concurrently might.
- **Long GC**: Triggered because a process spent an unusually long time garbage collecting.
- **Long schedule**: Triggered because a process was not pre-empted, probably because it was running a BIF or NIF, giving it more CPU time than its peers. This disrupts the soft real-time properties of the system.
- **Busy dist port**: Triggered when the distributed Erlang port is kept busy by a process sending large volumes of data.
- **Busy port**: Triggered when the port is kept busy by a process sending large volumes of data.

Keep an eye on the above metrics. They will tell you if there is an issue with parts of your system, more specifically how you handle your data and memory. The metrics will not tell you where the issue is, just that there might be one. If you suspect that is the case, turn on and receive the system monitor notifications.

You do so by setting some configuration flags. Find the `wombat.config` file in the directory `rel/wombat/files`, and add the following line:

```
1  {set, wo_plugins, plugins, code_tracer, report_system_monit
   or_notifs, true}.
```

When you have done this, you need to stop and restart WombatOAM. On the command line, go to the WombatOAM root directory (you don't need to start a shell) and enter the following:

```
./stop.sh
```

Wait for "ok", and then enter the following:

```
./start.sh
```

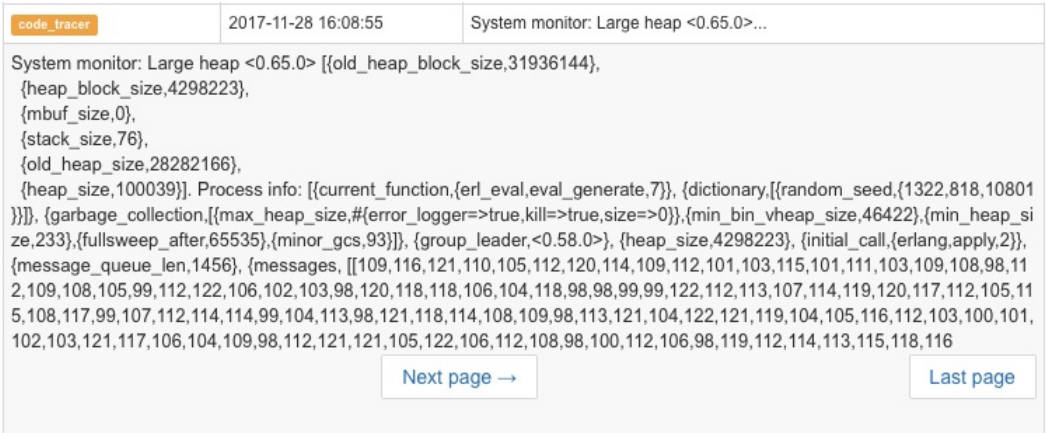
(You'll be able to do this from the dashboard in an upcoming version of WombatOAM.)

> **Note:** After completing your tests, you may want to turn these notifications off again. We have worked with systems that, under heavy load, generated millions of log entries per day! To turn of the notifications, remove the line that you added earlier from the `wombat.config` file, and then stop and restart WombatOAM again.

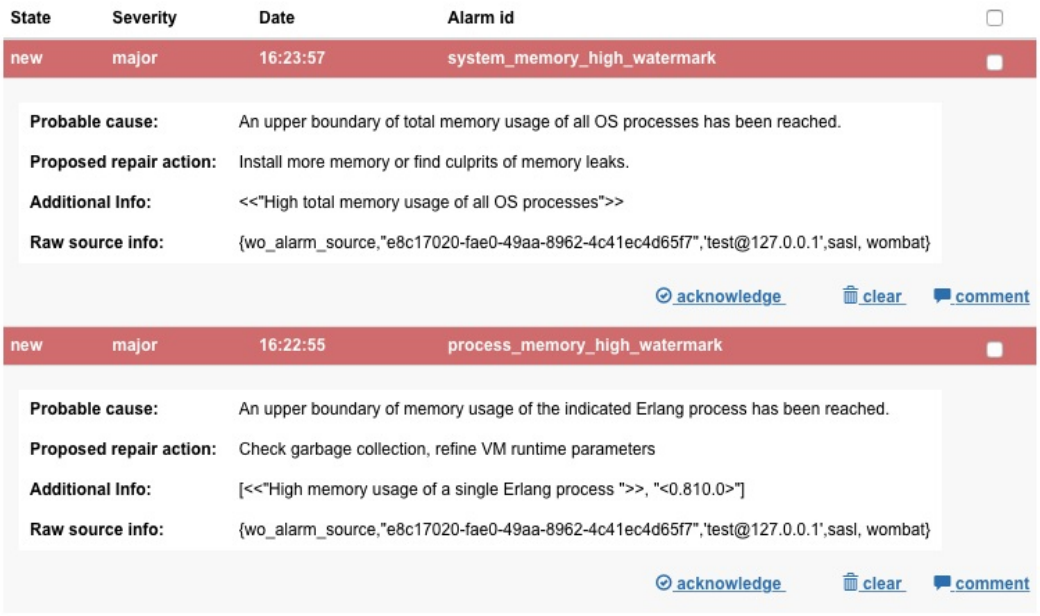
With the System Monitor notifications enabled, re-enter the following command in your shell:

```
[ self() ! [ random:uniform(25)+97 || _CharPerMsg <- lists:
seq(1,10000) ]
     || _NumOfMsgs <- lists:seq(1,1000) ].
```

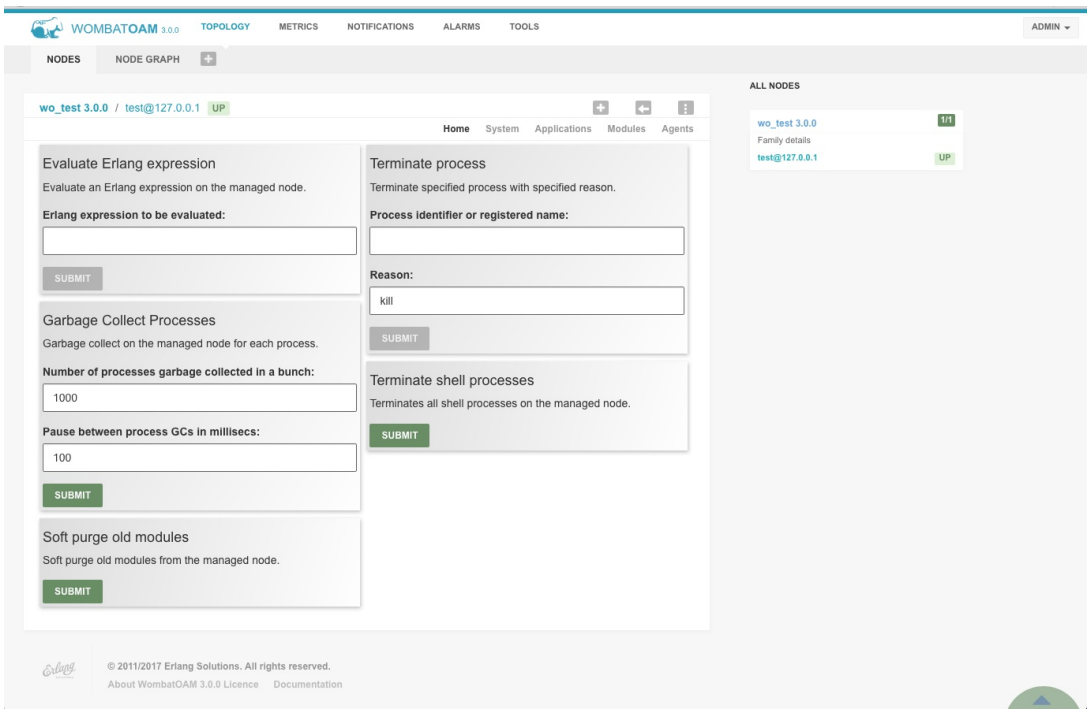
After your command has run for a while, you should see the following **code_tracer** notification: **System monitor: Large heap**



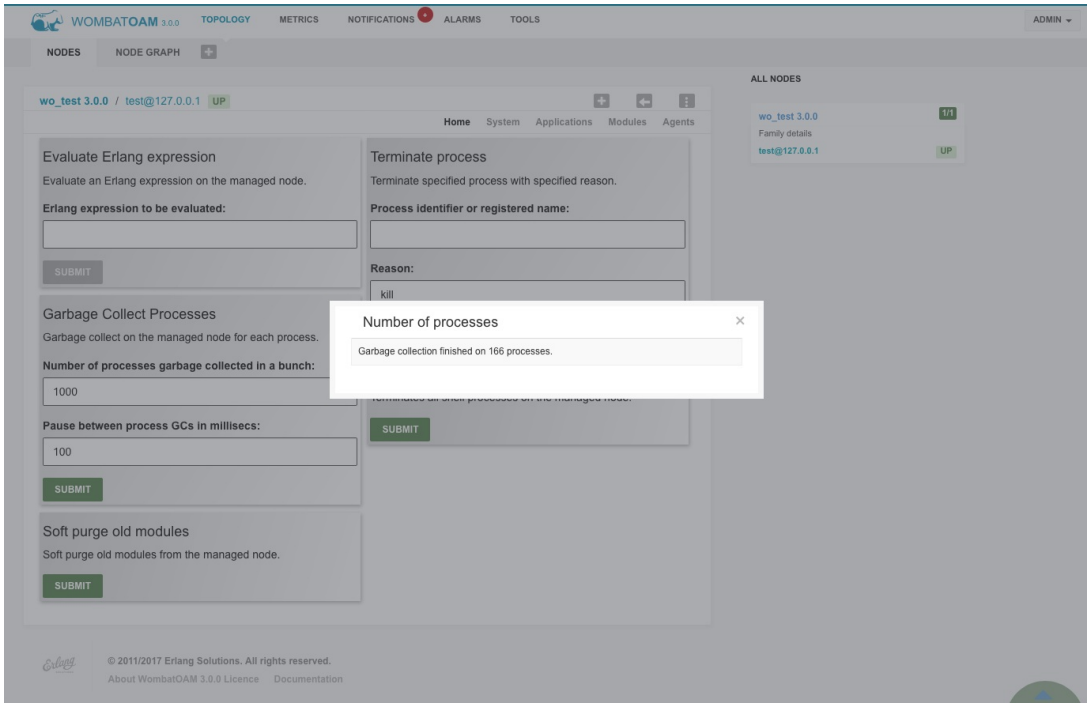
Also, two new alarms have appeared, namely, system_memory_high_watermark and process_memory_high_watermark, telling you that your system uses too much memory.



A possible solution for such a problem is to force garbage collection on the node. To do so, go to **Topology** → select the node with high memory usage → **Home** . Customise the garbage collection in **Garbage Collect Processes** panel and submit the request.



The garbage collection will be carried out on all processes of the node freeing up memory.



Now as the node is using less memory both alarms should have been cleared.

After completing this test, enter `flush().` to clear the messages sent to the shell.

"We once spent three months soak testing a system which, contractually, had to run for 24 hours handling 15,000 operations per second sustained. Each operation consisted on average of 4 http requests, 7 ETS reads, 3 ETS writes and about 10 log entries to file, alongside all of the business logic. The system was running at 40% CPU with plenty of memory left over on each node. After a random number of hours, nodes would crash without any warning, having run out of memory. None of the memory graphs we had showed any leaks, and even if we were polling at 10-second intervals, about 400 MB of memory was still available in the last poll right before the crash. We suspected memory leaks in the VM, runaway non-tail recursive functions, reviewed all the code and ended up wasting a month before discovering system monitors (which at the time were hidden at the bottom of a module description in the documentation). Eventually we turned on the system monitor and noted that a few seconds right before the crash, an unusually high number of long garbage collection and large heap trace events were generated. These were connected to the creation of a session, where an XML file sent back to us with session data caused a huge memory spike when parsed. We were seeing memory spikes when plotting our graphs, but did not think much about them because they were contained. What happened in the run-up to every crash was a surge in session initialization requests, causing these spikes to pool together and create a spike in memory usage which caused the VM to run out. We eventually discovered that the more cores we used increased the probability of this monster wave happening. In less than half a second, this memory surge used up all available memory and caused the node to crash. We estimate this particular issue in the 3 months of soak testing (as each crash took up to 20 hours to reproduce) and kept two people

busy for a whole month trying to figure out what happened. Had the system monitor been there from day one, we would have saved three months of trouble shooting and optimizing, and solved the problem in no time."

Loading a new module

Loading a module will produce a notification. You can try this out by creating a simple module (that doesn't do anything!) and loading it using the shell. In the directory where your node is running, create a text file named `my_mod.erl` with the following content:

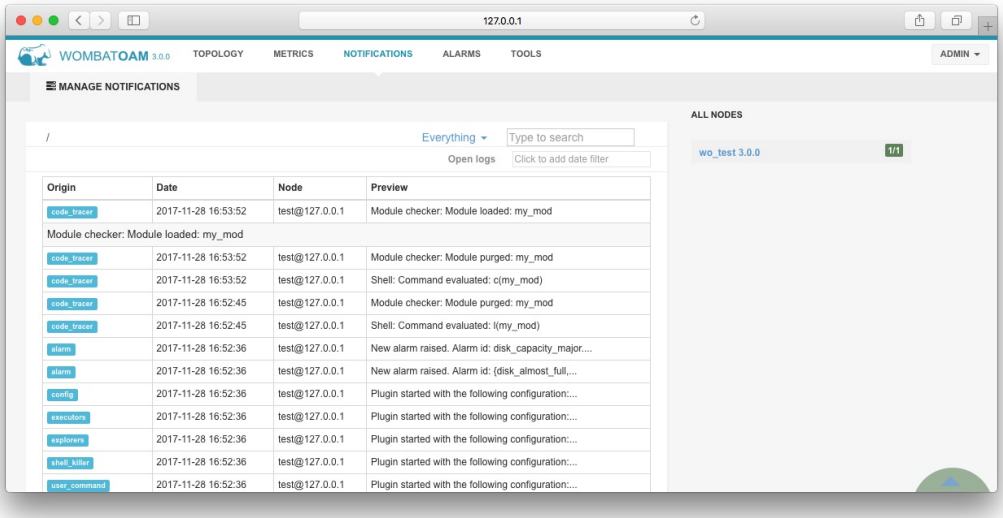
```
-module(my_mod).
-export([f/0]).

f() ->
    ok.
```

In the shell, enter the following, which both compiles and loads the module:

```
c(my_mod).
```

This will trigger the following **code_tracer** notification: **Module checker: Module loaded: my_mod**



Bonus: old_code alarm

Following the preceding steps you took to compile and load a module, you can trigger an **old_code** alarm. Note that the code_tracer configuration should still be enabled for this.

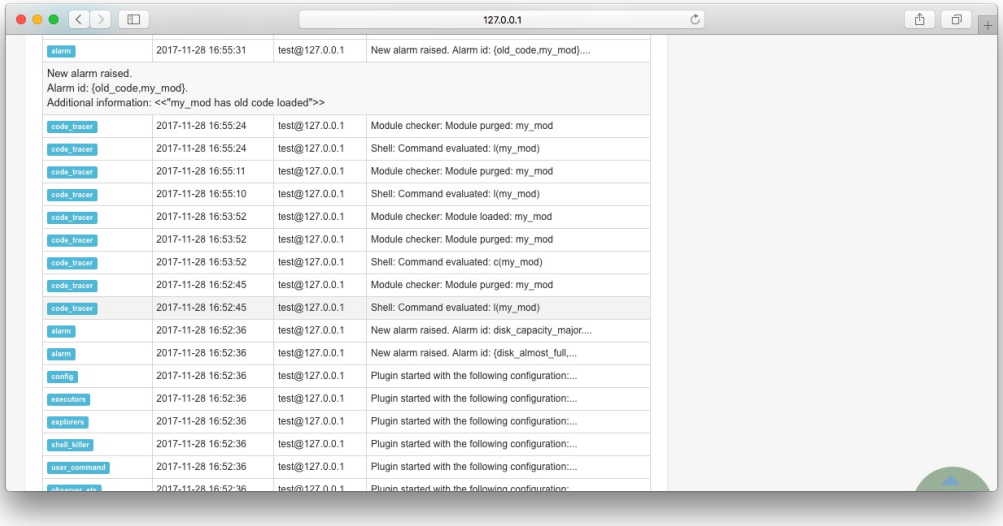
Stop your test node and start it again. Enter the following to **load** the module once more:

```
l(my_mod).
```

Then enter the following to **compile and load** it again:

```
1  c(my_mod).
```

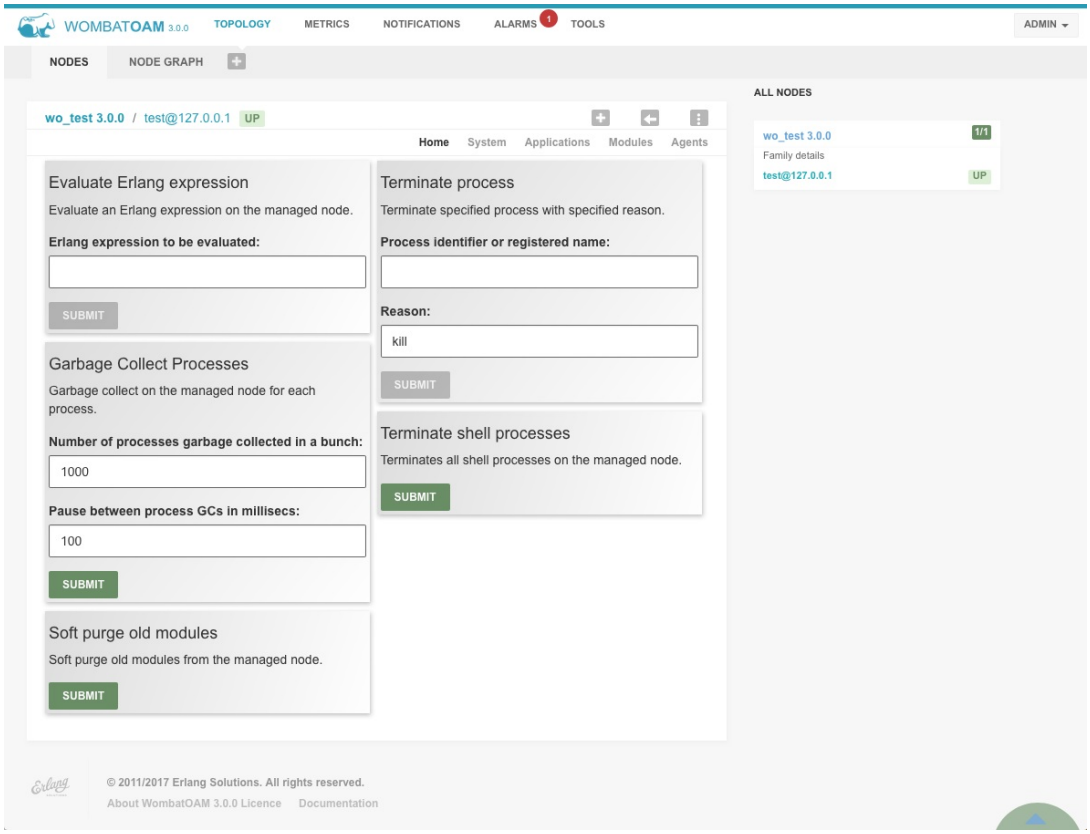
This should produce an **old_code** alarm, which indicates that a process or a user (via a shell) loaded a new version of a module on the node.



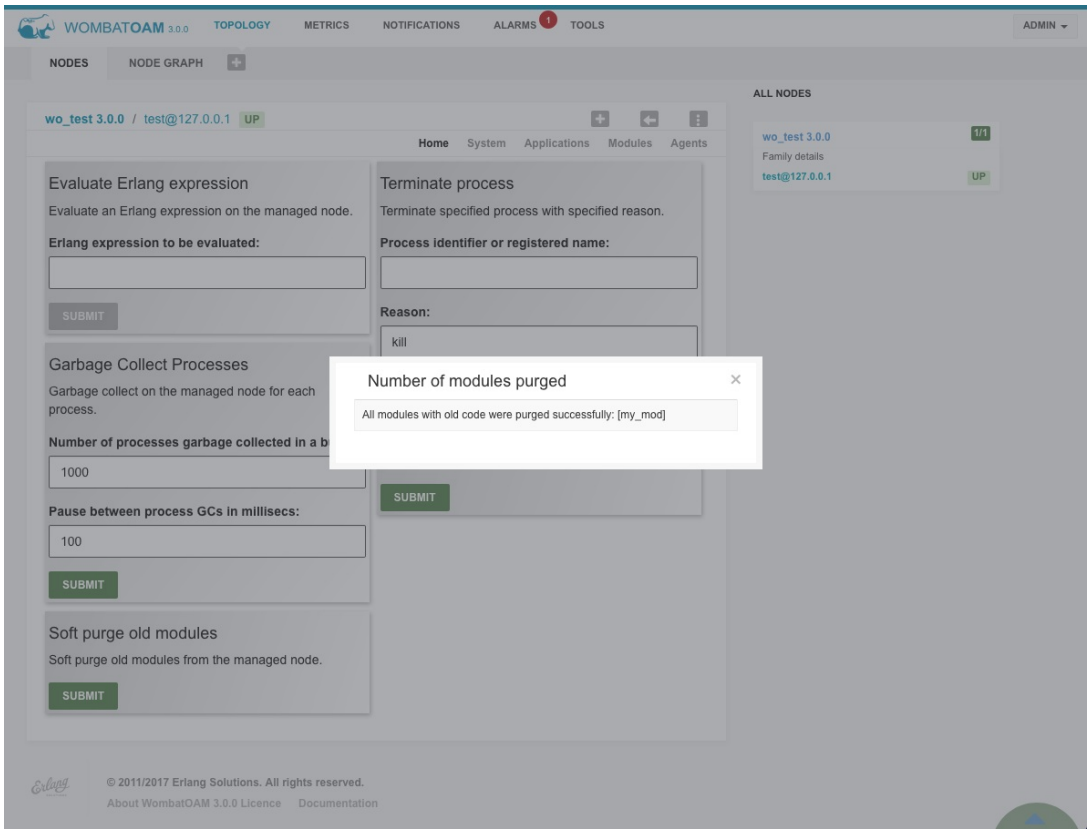
This is quite normal during a hot code upgrade or on a development node that is configured to recompile modules when the source code is changed, but in a production environment it is likely to indicate a problem either in configuration or in deployment management.

While it useful to apply patches manually to keep control of the process, automating system deployments by using scripts is less prone to errors. Nonetheless, scripts can also fail, so if you suspect that some nodes are using different versions of the same module, check the MD5 hashes or the used compilation options hold for the module loaded on those nodes. You can use WombatOAM to easily retrieve this information: Under **Topology**, select the node or node family, click **Modules**. WombatOAM shows detailed information about all the modules loaded on your system. You can find a specific module by using the search box at the top.

When the old version of the module is no longer in use, there is no need to keep it loaded in. To get rid of such old versions, go to **Topology** and select the node having old code → **Home**. In the **Soft purge modules** panel submit the request. Executing this request is safe, as only the old version of those modules are purged that aren't referenced by any process.



As the result, the old version of my_mod was successfully purged, which cleared the **old_code** alarm automatically.



different_application_versions alarm

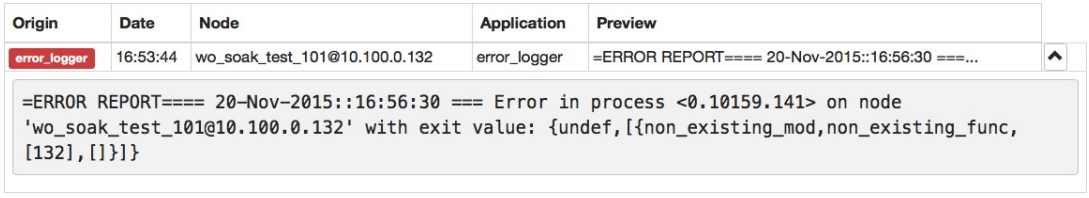
It isn't only the presence of different module versions that can lead to partial service degradations. If different application versions are running on some nodes of a cluster, users are also likely to experience problems.

"I was visiting a customer trialling WombatOAM to show how they can totally exploit what WombatOAM provides. First, I noticed that they have an active alarm. Going into the Alarms page, I saw the different_application_versions alarm relevant for their production Riak cluster had been raised. Having described the alarm, which is raised if some nodes belonging to the same cluster use different version of a certain application, they immediately questioned the devops. The devops were saying for sure that it couldn't happen to their production cluster, but they left us alone to double check. While the devops were away we were exploring the Notification's features. Browsing the collected log entries from the Riak cluster we noticed that crash reports about calling an undefined function were repetitively generated on some Riak nodes. These nodes were exactly the ones that were running the old version according to the alarm! At that time the devops returned and admitted that they had forgotten to upgrade those Riak nodes and just completed the upgrade process. In evidence, the active alarm got automatically cleared by WombatOAM confirming that the anomaly disappeared."

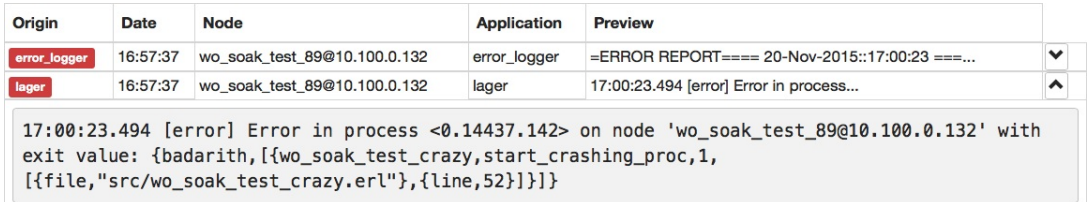
Crash reports

WombatOAM shows crash reports under Notifications. If you have a few hundred nodes in production, you can conveniently aggregate crashes in one place, or write handlers to forward them to a third-party tool, email them, or automatically raise tickets.

Below are two examples, encountered during heavy soak testing. The first one was collected from SASL:



The following crash report was collected from the **lager** handler:



Consider the following story from a WombatOAM customer:

"We had a client with 200 nodes in production. The only way for them to detect that a process had crashed was to log on to a machine, into the node shell and filter in the SASL report browser. WombatOAM retrieves all of the SASL and Lager reports for you in one place, so you can notice crashes and address them."

Tools

Functionalities under Tools are several standalone features, which are not metrics, notifications or alarms, but aid devops with their daily tasks. It is handy for troubleshooting, online diagnosing and resolving problems and incidents on the fly.

You have seen how you can exploit the Etop-like process listing, the ETS table viewer, the various process inspectors.

To experiment with these features, first start two Erlang nodes and add them to WombatOAM.

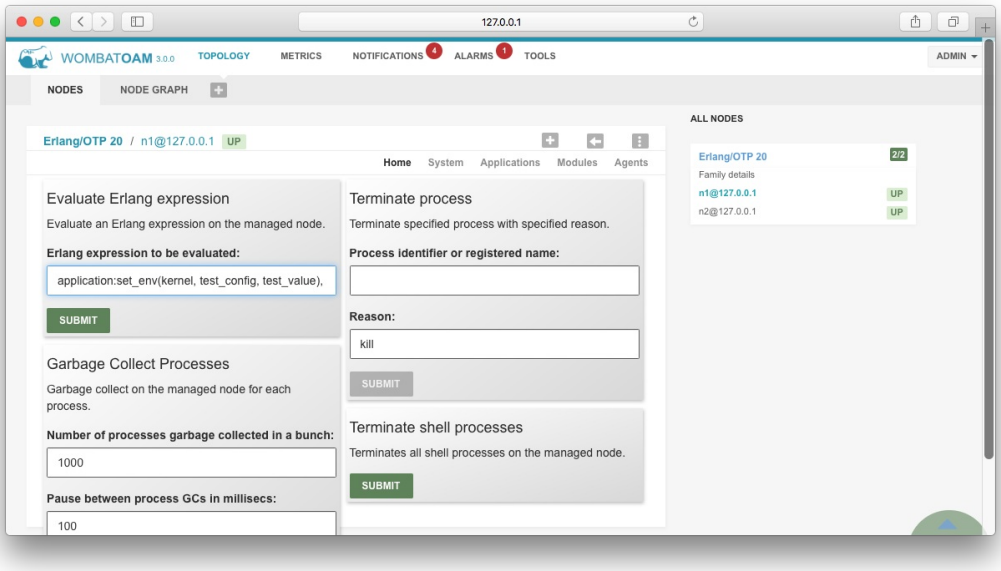
```
erl -name n1@127.0.0.1 -setcookie abc123
erl -name n2@127.0.0.1 -setcookie abc123
```

User command executor

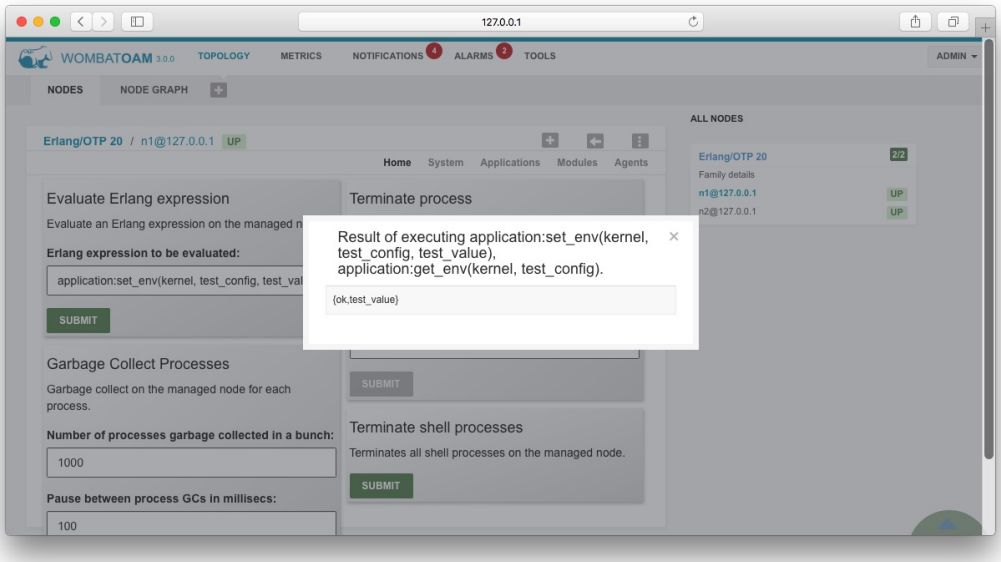
This service allows users to execute arbitrary Erlang expressions on the nodes and the result of the execution is shown to the users.

Go to **Topology** → select the `n1@127.0.0.1` node **Home**, and in the **Evaluate Erlang/Elixir expressions** panel. Copy the following Erlang expression into the input field and start the request.

```
application:set_env(kernel, test_config, test_value),
application:get_env(kernel, test_config).
```



This will associate `test_value` with the `kernel` application's `test_config` parameter, and then will return the current value of the same parameter, which will be the result of the request.



Use this service to retrieve information specific to your business logic or adjust your system to the current requirements. However, if there are commands you often use, you may consider adding them as new services. Implementing your own service is easy and will speed up your maintenance processes.

Configuration management

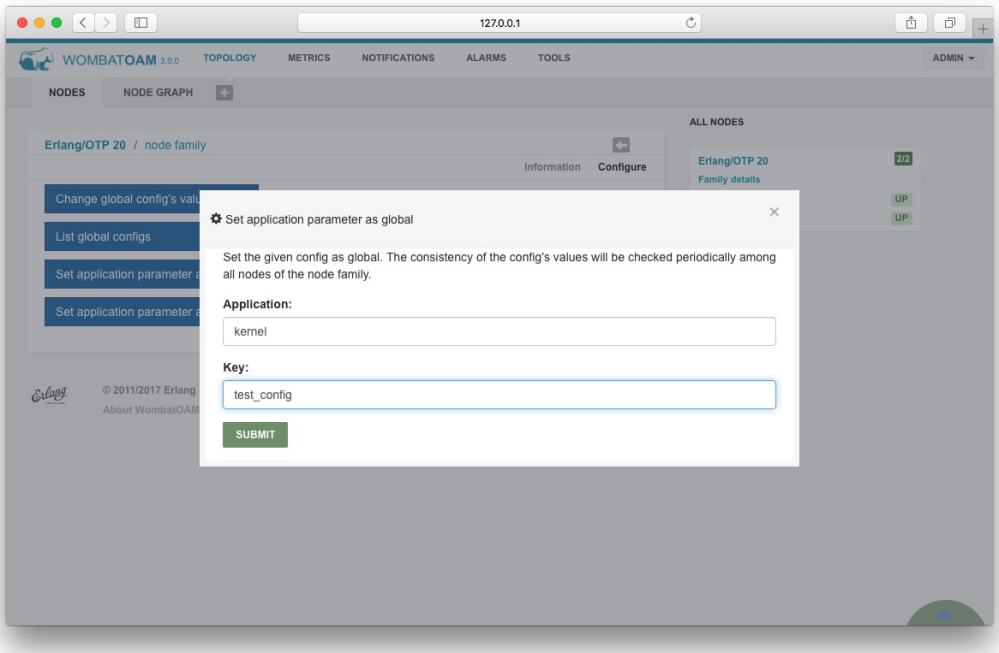
Tailoring an application to fit a certain system is achieved by configuring the application. An application can be configured using its *configuration parameters*, which are often referred to as environmental variables or simply envs.

There are two kinds of configuration parameters, namely, a configuration parameter can be local or global. *Local configuration parameters* are specific to Erlang nodes, their values can vary among the nodes. The data's root directory of an Erlang node, the HTTP port on which the Erlang node listens are good examples for local configuration parameters. On the other hand, *global configuration parameters* must have the same value set on all the nodes belonging to the same node family as they describe properties of the overall system. As examples, consider the IP address of the load balancer or the lifetime of sessions.

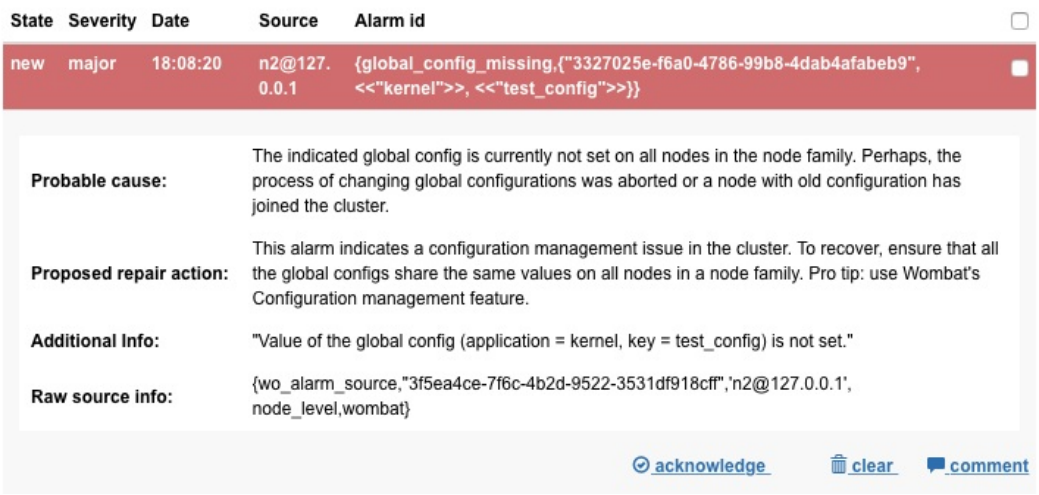
WombatOAM allows you to manage your configs both on node and on cluster level. For instance, what we did in the previous example can be done using the node level services.

Also, WombatOAM periodically scans for anomalies among the nodes that should share the values of global configs. It will raise an alarm whenever different values are set to any global configuration or whenever any global configuration is missing from any of the nodes.

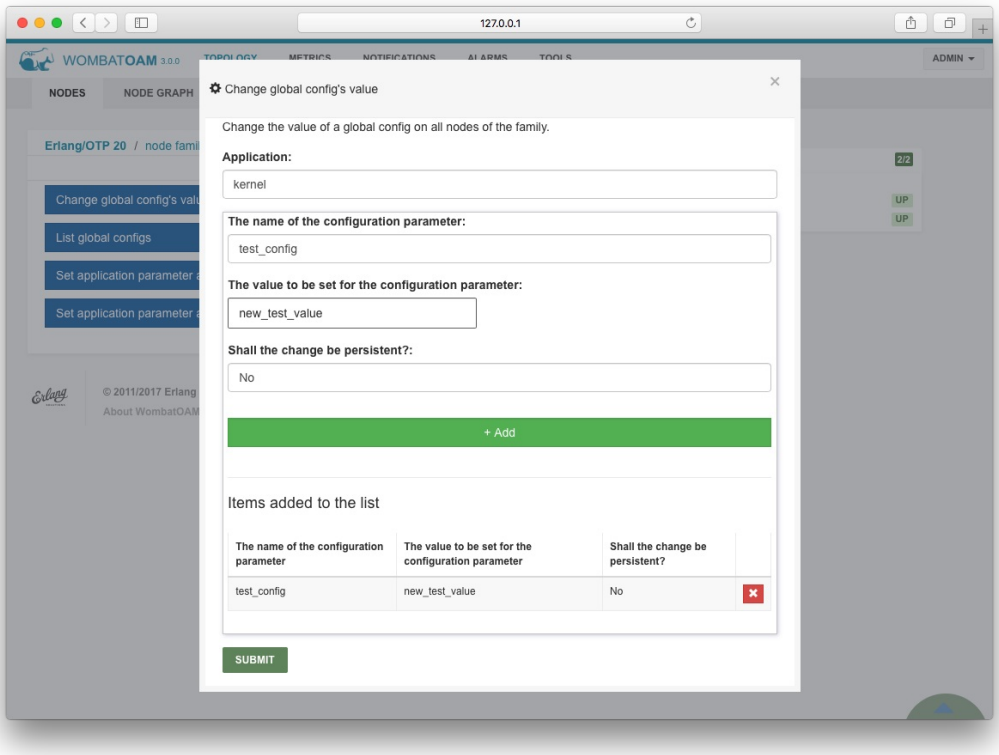
To explore this feature, go to **Topology** and select the *node family* to which both test nodes belong → **Configure** . Then, open the **Set application parameter as global** panel, choose *kernel* as the *Application* and choose *test_config* as the *Key* and then submit the request.



This will mark the *test_config* parameter as global on both test nodes. As previously we changed the *test_config* parameter only on one node, a new alarm will be raised because this config is missing from the other node.

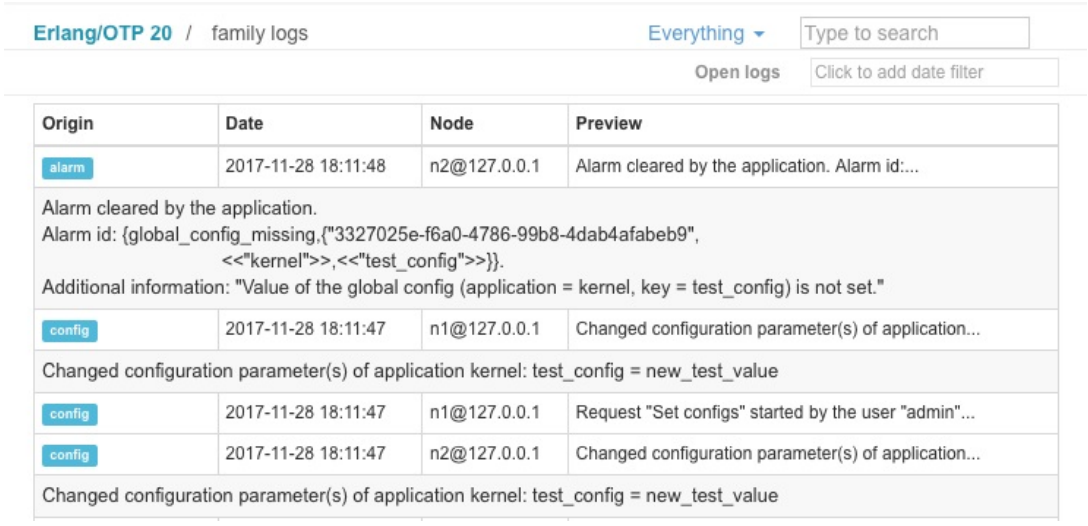


To resolve this issue, go to **Topology** → and select the *node family* again → **Configure** . Then, open the **Change global config's value** panel, choose *kernel* as the *Application* , *test_config* as the parameter name, *new_test_value* into the input field and then click **Add** button. Finally, submit the request.



This will change the config's value on both nodes.

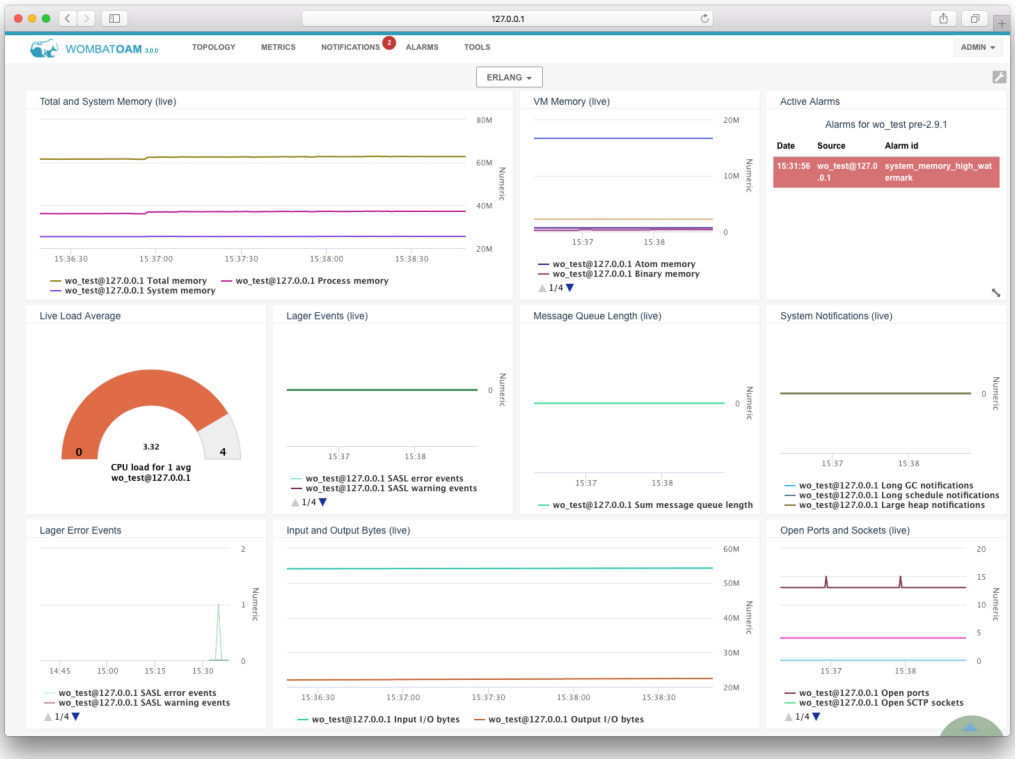
As we managed to set the same value for the config on all the nodes, the values of the config are consistent with each other, thus the alarm got cleared by WombatOAM. Also, notice that all the changes performed by WombatOAM are always recorded as notifications, providing a history available for all team members.



Overview page

WombatOAM's Overview page gives a brief summary about the current state of your system. By default it shows the most helpful metrics, the active alarms and general statistics. It is easily customisable, e.g. graphs with any groups of metrics can be added to the page.

Click the WombatOAM logo in the top left corner to explore the Overview page.



More stories from the front line

In the following stories, we'll see how WombatOAM addresses real-life problems that customers have encountered.

From a major mobile network operator

> "I had a node crashing and restarting over a 3 month period at a customer site. Some refactoring meant they were not handling the EXIT signal from the ports we were using to parse the XML. Yaws recycled processes, so every process ended up having a few thousand EXIT messages from previous requests which had to be traversed before the new request could be handled. About once a week, the node ran out of memory and was restarted by heart. The mailboxes were cleared. Our customers complained that at times, the system was slow. We blamed it on them using Windows NT. We saw the availability drop from 100% to 99.999% (The external probes running their request right during the crash or when the node was restarting) about every 4 weeks. We rarely caught this issue as external probes sent a request a minute, took half a second to process, whilst the node took 3 seconds to restart. So we blamed it on operations messing with firewall configurations. With triple redundancy, it was only when operations happened to notice that one of the machines running at 100% CPU that we got called in. Many requests going through the system, I thought, but it was only 10 requests per second. Had we monitored the message queues, we would have picked this up immediately. Had we had notifications on nodes crashing, we would have picked up after the event."

WombatOAM would have raised three alarms as a result of the above

issue: an alarm when the message queue of a process exceeded certain thresholds, another when it reached a high memory utilisation, as well as one when the node was down. Even if alarms were cleared, following up on them would have lead to the memory metrics, showing the increase in memory usage; more specifically, in the process memory usage. This would have led us to look at the other process related metrics, spotting that the message queue was growing out of hand.

From an enterprise Riak user

> "A customer having a Riak cluster with 5 nodes reported that one Riak node crashed without any early warning signs. The customer reported that no maintenance activities had been performed on that node before the crash occurred. After long hours spent on investigating the issue, ESL pointed out that the root cause was a recent configuration change that increased the number of concurrent hand-offs 50 times larger than its default on the bad node. This change allowed the node to accept so many transactions that made the node totally overloaded. After reverting this change the cluster started functioning properly again."

Here, WombatOAM would have logged a notification and raised an alarm: a notification about the configuration change, and an alarm about the inconsistency of the global config's values. Using the configuration management service, the misconfiguration would have been fixed quickly.

Runaway modules

> "Someone tried to patch a module, loaded it, saw that it did not fix the bug and deleted the beam file, not knowing he had to purge the module as well. An engineer wasted a whole week figuring that one. That is why we have an Alarm if nodes of the same type run different modules."

If more than one node of the same type existed, WombatOAM would have raised an alarm that the two nodes were running different versions of a module with the same name. To find out how that came about, you would have looked at all of the code related notifications on that node, and finally, narrowed it down to the shell command.

File sanity check

> "There was a software upgrade in real-time which required upgrades of the Erlang Environment variables. Operations upgraded the sys.config file, copying and pasting from a word document, invisible control characters included. Months later, a power outage caused the nodes to be rebooted. But because of the corrupt sys.config file, they would not start. The error messages in this case were so cryptic and the control characters in the sys.config file not visible, it took us a few hours to find out what the issue was and restore the

service."

WombatOAM regularly checks the sanity of all of the files needed at startup. This includes the boot, app, config and others. If any of them are corrupt and will prevent your system from restarting correctly, WombatOAM will raise an alarm.

Conclusion

This walkthrough shows how WombatOAM collects many metrics, notifications and alarms from managed Erlang nodes. The dashboard helps to visualise this information, with graphs that can be placed on top of each other, and notifications and alarms that can be filtered. When the granularity of the collected information is not enough, you can zoom in and use Live Metrics to see the value of metrics at each second, or the explorer services to online debug your system.

WombatOAM monitors the managed nodes for signs of possible future problems – processes having long message queues, having too many ETS tables, reaching the process limit, filling the atom table, the Erlang node being down, and so forth – and generates alarms from these. It also generates notifications for events that happen within a node, such as commands typed in the Erlang Shell, modules that were loaded, and processes that crashed, thereby giving a history of events on the node. Alarms and error log entries raised by the application will also appear as alarms and error notifications in WombatOAM. All are useful to improve your system by pointing out the weakest part, and then Services can help you to further narrow the case. Also, when there is an on-going outage Services will help you to recover from it immediately.

While Services allow you to inspect and change your system, metrics, alarms and notifications are useful both for detecting early signs of errors that might cause future outages, and investigating past incidents so that you can make sure they don't happen again.

What's next

WombatOAM has a plugin system that lets **you write your own plugins**, which can collect metrics, notifications and alarms specific to your business logic, and provide new services best fitting to your daily routine. For example, if an application has its own API for serving metrics, a plugin can use this API and provide the metric values to WombatOAM. Or if the user wants to generate a notification each time a certain function is called, a plugin can subscribe to calls of this function and generate the appropriate notifications.

WombatOAM has integration capabilities with other OAM systems, meaning that the information collected by WombatOAM can be pushed into these tools. Supported OAM systems include Graphite, Grafana, Cacti, Graylog, Splunk, Zabbix, Datadog, Logstash, Nagios, AppDynamics and PagerDuty, and because WombatOAM exposes the collected information via its REST API, integration with other OAM systems is also possible. If you already use OAM systems, then **hooking WombatOAM into your existing infrastructure** is a logical next step, since it would allow you to view the new information via the usual channels.

After a few days of having a system monitored by WombatOAM, it is worth **analysing the information collected**:

- Look at the active alarms and try to uncover their reasons. Do the same with notifications and cleared alarms.
- Look at the error and warnings notifications, including crash logs.
- Go through the metrics to see if there is anything suspicious, such as strange spikes in memory metrics and the message queue length metric. Going through the metrics will also help to establish

a baseline level for them, so that anomalies can be spotted in the future.
- Check the Process notification metrics, such as Busy port and Busy dist port. Numbers significantly higher than 0 indicate possible performance problems.

You may be surprised at what you discover.

Getting in touch

To learn more about WombatOAM, or get in touch with any questions, visit our website:

- **WombatOAM product page:**
  https://www.erlang-solutions.com/products/wombat-oam.html
- **Contact Erlang Solutions:**
  https://www.erlang-solutions.com/contact.html