Towards a flexible and reactive diagnostics tool for ROS

Vladimir Golubev
Applied Mathematics and Informatics
Faculty of Computer Science and Technology
Saint Petersburg Electrotechnical University
Email: trevorismay@gmail.com

***Abstract*—Software diganostics can save a significant amount of programmers debug hours in case it's detailed and vivid. ROS is a robotics middleware with mature ecosystem and large amount of debugging utilities. Besides that, it provides a number of advanced tools for analyzing environment and individual node state (e.g. *ros_runtime_monitor*, *rqt_console*, *rosbag*, etc.). But it still lacks a standard system and process diagnostics tool. This paper attempts to present a reusable solution to fill this gap and highlight the benefits of such tool for ROS community.**

***Index Terms*—diagnostics, system monitoring, Linux, ROS, publisher-subscriber, reactivity**

I. Introduction

Every more or less complex system possesses the means to report its state changes. Linux, for example, has *syslog* - a common log sink [1]. And */proc/kmsg* [2], [3] in conjunction with *dmesg* [4] is a good example of flexible and loosely coupled kernel feedback mechanism. Those mechanisms provide the interface to inspect system and process logs, in the aggregate - which is extremely illustrative. Late versions of *dmesg* also have the ability to highlight its output. Logs are, by their nature, reactive - they are generated by an actor (which is in our case a process or an individual thread) at the correct moment of a certain event. This property is the main reason why logs are so useful for debug and observation purposes - the collection of logs from different sources is being constructed in the correct chronological order. For the sake of deferred examination *syslog*, */proc/kmsg* and its clients employ timestamps [1], which is a common practice for mechanisms of this sort.

Not all system information stack can equally provide reactivity, but this can be emulated (poorly sometimes). For example, Linux provides the ability for user to poll current memory state through */proc* pseudo-file system [2], [5], but there are currently no events, which reflect memory consumption jumps or high increase in page swapping. The reason for this is that the techniques for monitoring memory state vary greatly from user to user. For example, there is no absolute **very high** value of memory consumption increase - this depends on a concrete Linux setup. In other words, this is a high-level functionality, a delicately tuneable one.

ROS is a robotics middleware, which encapsulates underlying operating system abstractions and provides mature development ecosystem. It implements several different styles of communication, one of which can be characterized as message passing, publisher-subscriber paradigm [6]. This workflow is based on topics (discussed below), and in this paper will later be referred to as reactive. This style of communication is of interest to this work, because, as was mentioned earlier, message-based diagnostics is intuitive and descriptive. The main unit in ROS computaion graph is *node*, which is represented by a system process [7]. *Topic* is a typical tool for observing another node changes. It is a unidirectional named transport, based on standard TCP or UDP stack [8]. As will be shown later in this text, there is currently no decent reactive system monitor for ROS - a tool which will be flexible and vivid enough to cover valuable system and node changes altogether.

This paper attempts to describe the requirements for such a set of tools and denotes the way of building this system. The main purpose of this work is to present a flexible and responsive tool for monitoring system and individual node state in ROS.

II. Requirements

A. Low profile

Diagnostics system must be lightweight. There are two basic principles:

- Avoid excessive resource consumption.
- Don't interfere in other process business.

A proper counterexample would be a process, which polls (and reports) current CPU load in a tight loop. This code would itself generate unnecessary load, and take precious userspace/system time from other participants, which effectively goes against both of the discussed points.

Ideally, diagnostics system would subscribe to kernel and process events and filter out non-significant changes, but this is usually not possible due to system API specifics, discussed in IV. Which means that system call frequency matters.

To follow ROS reactive workflow, synchronious polling approach must be turned to asynchronous on the interface level, which requires some internal state, to report swap space usage difference, for example. Diagnostics structure size may seem insignificant, but to watch over its bloat is a good guideline nevertheless.

The implementation preformance should be continuously challenged. On a certain stage of development is would be possible to perform reflective monitoring, a "meta diagnostics".

B. Reactivity

Reactivity is a key point in designing diagnostics tool. The reported events reflect system state changes. Discussed in I, this property is derived from standard ROS publisher-subscriber workflow. Although the interface should obviously be implemented using ROS *msg* stack, it's still not clear how to effectively organize message topics and diagnostics nodes. For a system of this sort it's important to be able to report nontrivial changes such as:

- Unexpected CPU load growth.
- Sudden memory consumption jumps.
- Excessive file descriptor usage for individual processes.
- High system call frequency (ergo high contention over kernel stuctures).

and etc.

Those events provide a valuable insight into system state, but are challenging to build correctly. Not on the implementation level, but because for such topics the events of interest vary from one system setup to other. This also leads us to the question of flexibility and configuration.

To conclude, reactivity can be achieved by reporting notable system change as close as possible to the moment of its exposure. The *msg* stack is especially suitable for this task.

C. Flexibility

The last topic raised in II-B leads to the important question: "How to implement a diagnostics system, which is equally usable for different setups?". The answer is configuration, but the choice of this mechanism is still up to us. Some standard configuration tool must be chosen.

It must be possible for user to switch features on and off depending on her demands to avoid unnecessary *sysfs* polling overhead, for example. The ability to tweak resource limits and timeouts of operations under observation is also important.

D. Portability

In this context portability is denoted by platform independence. This is somewhat unimportant, considering how highly ROS is Linux-dependent. However, [9] stands that it's preferrable to make ROS-independent package, which will later be wrapped using ROS-specific tools, for code reuse reasons. Considering this guideline, it's favourable to produce original (non-wrapped) package using platform-independent tools.

This goal can be achieved either by choosing already portable tools, or by wrapping abstractions uniformly. The former way is elegant and more preferrable - the portability property is simply inherited.

III. Alternatives

First of all, ROS ecosystem already provides a standard carcass for reporting system-wide evnts - the */diagnostics* topic [10]. But this is not enough, because there is no standard package for collecting generic system diagnostics. It was designed mostly for individual device control, but the current stack is suitable enough for monitoring system-wide events.

Current attempts to build the desired package are listed below:

psutil_ros [11]

- Strategy - Uses custom topics in global namespace with custom messages.
- Backend - Separate, *psutil*.
- Portability - Portable, inherited from *psutil*.
- Reactivity - Non-reactive.
- Coverage - Basic CPU, memory and network statistics.
- Flexibility - Non-configurable.

This package provides basic system events, which is enough for superficial overview. However, this tool is not reactive - **psutil_ros** merely reports current system state. This is a major disadvantage, since the diagnostics is not illustrative. Moreover, this package defines custom interface and is not meant to be integrated with */diagnostics* stack.

The important thing about this package is that this is a tiny *psutil* wrapper for ROS. This conforms to practces, described in [9].

ros-system-monitor [12]

- Strategy - Uses */diagnostics* topic with custom *DiagnosticArray* structure updaters.
- Backend - Embedded, custom *sysfs* and UNIX tools processing.
- Portability - Linux-dependent.
- Reactivity - Reactive.
- Coverage - Basic CPU, memory, drive, NTP and network statistics. Resource limit reactivity.
- Flexibility - Configurable through *roslaunch*, hard-coded limits and thresholds.

This package provides more broad system coverage than *psutil_ros*. In includes CPU load, times and temperature, detailed memory overview, network and IO counters and NTP statistics. Another important aspect is that *ros-system-monitor* uses standard */diagnostics* stack, which was described earlier. Not only that, but it can be configured using *Parameter Server*, e.g. high CPU load - **cpu_load_warn**, high HDD temperature - **hdd_temp_warn**, NTP time offset tolerance - **offset_tolerance**.

There are several disadvantages. It does not report sudden changes - for example, memory consumption jumps - only some absolute limit exceedance. This makes diagnostics less vivid. *ros-system-monitor* also lacks separate backend. Currently custom *sysfs* processing is used, which is embedded into reporting scripts. This is not portable and does not conform to [9].

All the described packages provide global system diagnostics, lacking per-process statistics. There is currently no tool to monitor and report node state in ROS.

IV. Implementation

A. Strategy

Ideally, a tool would subscribe to the desired set of system events and report those accordingly, following ROS reactive workflow. However, this is challenging, because not all kernel

changes may be observed in an asynchronous way due to their implementation factor. In particular, kernel provides read-only access to some of its high-level data structures through *sysfs*, propagating standard UNIX file workflow [5]. One would use *ionotify* API [13] to observe filesystem changes in an asynchronous way, yet this approach would not work on *sysfs*, because *sysfs* entries are lazy - there are no changes, until client requests current state [3]. This leads to the conclusion, that most kernel changes can be observed only by *sysfs* polling.

B. Interfaces

First of all, as was already discussed in III, ROS ecosystem provides a standard topic for system diagnostics - the */diagnostics* topic [10]. All the high-level events would appear there. Thore are:

- CPU load
- CPU times
- Memory/swap usage
- Network counters
- Drive IO counters
- Resource-hungry processes

It goes without saying, that messages, pushed to diagnostics topic, must be packaged in standard *DiagnosticArray* type to be usable, for example, for *rqt_runtime_monitor*.
Individual process monitors should provide separate interfaces, because pushing process events to */diagnostics* topic would pollute it and make operator unable to detect valuable system changes in a timely manner. Those are:

- CPU load
- CPU times
- Heap/system memory usage
- Network counters
- Drive IO counters
- Context switches
- Thread stats
- File descriptors
- System call frequency

Per-process events are to be organized in one separate topic, much like */diagnostics* aggregates system-wide events. It would be reasonable to use *DiagnosticStatus* structure for this in the virtue of code reuse, but field semantics is a bit different - it's nice for interfaces to speak for themselves. It goes as follows:

```
ProcessStatus:
    byte level
    uint32 pid
    string message
    KeyValue[] values
```

The level is enumerated in *DiagnosticStatus*. *uint32* is chosen for pid, because, according to [2], pid values must wrap around (which denotes their unsigned nature), and `PID_MAX_LIMIT` is 2^{22}.

C. Configuration

Polling method requires some user-defined value of idle time interval. This value is setup-specific, so some common configuration mechanism must be chosen. Luckily, ROS ecosystem provides standard configuration mechanism - *launch* files, *Parameter Server* and the *roslaunch* tool. For configuration, the default values will be chosen, where possible. Those may be overriden by *roslaunch*. Poll interval is not the only configurable value. Critical memory level, swap consumption rate, high file descriptor count are among the others.

D. Grouping

The events are to be grouped by nodes based on their subject. Those are:

- CPU monitor
- Memory monitor
- Network monitor
- Drive IO monitor
- System-wide process monitor

E. Language and backend

The language of choice would be *Python*, and, in case its ecosystem would fail to provide ready-made solutions, suitable for specific subject monitoring, the implementation would fallback to *C++* through *Boost.Python* [14] (or to a pure *C++* node).
On the higher level, the implementation uses *psutil* framework. It containts the utilities suitable to cover the basic needs (both system-wide and process-specific) [15]. The implementation portability property is also derived from *psutil*, which is favourable.

F. ROS-specific tools

Since the */diagnostics* stack is used, it's reasonable to integrate with *diagnostics_updater* package, described in [10] and http://wiki.ros.org/diagnostic_updater. This package encapsulates all the boilerplate, associated with filling out *DiagnosticArray* structures and pushing data to */diagnostics*.

V. Conclusion

It's important to have reactive and flexible diagnostics system in hand, especially in a potentially complex ecosystem such as ROS. Reactivity is a major challenge - such a system must monitor a various number of activities, while being vivid and concise.
There is currently no satisfactory diagnostics package for ROS, the one up to the tasks stated in II. None of them provides process monitoring functionality.
The overall strategy, in most cases, is to poll *sysfs*, as Linux system API is not reactive. It's convinient to use current */diagnostics* ROS stack for high-level system monitoring, because it's mature and easy to integrate. Process-specific events are meant to be pushed to separate topic to avoid */diagnostics* topic pollution.
Current implementation is already able to provide basic

system-wide service, covering the most of the *ros-system-monitor* functionality. Unlike *ros-system-monitor*, it's portable and has a separate *psutil* backend, which is favourable. A certain amount of custom events is implemented - e.g. suden memory consumption jumps, or sudden CPU load growth. Future work is to enhance system-wide statistics and develop process-specific monitoring.

REFERENCES

[1] "The syslog protocol," Internet Request For Comments, RFC 5424, 03 2009. [Online]. Available: https://tools.ietf.org/html/rfc5424

[2] *A man-page for /proc pseudo-file system*. [Online]. Available: http://man7.org/linux/man-pages/man5/proc.5.html

[3] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linix Device Drivers*. OReilly Media, 2005.

[4] *A man-page for dmesg tool*. [Online]. Available: http://man7.org/linux/man-pages/man1/dmesg.1.html

[5] M. Kerrisk, *The Linux Programming Interface*. No Starch Press, 2010.

[6] Ros/concepts. [Online]. Available: http://wiki.ros.org/ROS/Concepts

[7] Ros/nodes. [Online]. Available: http://wiki.ros.org/ROS/Nodes

[8] Ros/topics. [Online]. Available: http://wiki.ros.org/ROS/Topics

[9] Ros best practices. [Online]. Available: https://github.com/ethz-asl/ros_best_practices/wiki

[10] "Diagnostic system for robots running ros," REP 107, 10 2010. [Online]. Available: http://www.ros.org/reps/rep-0107.html

[11] psutil_ros repository. [Online]. Available: https://github.com/brNX/psutil_ros

[12] ros-system-monitor repository. [Online]. Available: https://github.com/ethz-asl/ros-system-monitor

[13] *A man-page for ionotify API*. [Online]. Available: http://man7.org/linux/man-pages/man7/inotify.7.html

[14] *Boost.Python manual*, 08 2017. [Online]. Available: http://www.boost.org/doc/libs/1_65_1/libs/python/doc/html/reference/index.html

[15] *psutil maunal*, 09 2017. [Online]. Available: https://media.readthedocs.org/pdf/giamptest/latest/giamptest.pdf