"Drums": a Middleware-Aware Distributed Robot Monitoring System

Mani Monajjemi
School of Computing Science,
Simon Fraser University
Burnaby, BC, Canada
mmonajje@sfu.ca

Jens Wawerla
School of Computing Science,
Simon Fraser University
Burnaby, BC, Canada
jwawerla@sfu.ca

Richard Vaughan
School of Computing Science,
Simon Fraser University
Burnaby, BC, Canada
vaughan@sfu.ca

ABSTRACT

We introduce `Drums`, a new tool for monitoring and debugging distributed robot systems, and a complement to robot middleware systems. `Drums` provides online time-series monitoring of the underlying resources that are partially abstracted away by middleware like the Robot Operating System (ROS). Interfacing with the middleware, `Drums` provides de-abstraction and de-multiplexing of middleware services to reveal the system-level interactions of your controller code, the middleware, OS and the robot(s) environment. We show a worked example of `Drums`' utility for debugging realistic problems, and propose it as a tool for quality of service monitoring and introspection for robust autonomous systems.

Keywords

Robot Monitoring System; Distributed Monitoring; Fault Detection and Diagnosis

1. SCOPE AND MOTIVATION

The last decade has seen the rise of robot middleware. Many or most researchers and robot developers now take for granted the existence of a few well-known platforms, exemplified by ROS[7], for rapidly assembling robot systems based on mature, well-designed interfaces and a catalog of high-quality Open Source components. No doubt this has increased the productivity of the research community, and there is a current effort to transfer these benefits to industrial robotics[1].

Much of the benefit of these middleware systems is obtained from the abstractions they provide. For example in ROS, communication between components is by logical publish-subscribe channels called "topics". Internal to ROS, topics are usually implemented using pairs of TCP sockets and a central directory service (the "master") for establishing connections. Assuming the ROS platform is working properly, the user does not need to think about the details of networking: it just works transparently.

However these usually-useful abstractions have an important disadvantage, in that failures and resource constraints in the underlying mechanisms are not apparent. An abstraction layer that hides the existence of a bundle of underlying TCP socket pairs is not well suited to letting you know when one socket pair is suffering lots of dropped packets.

Whether robots are experimental or intended for deployment, failures and glitches in the underlying systems are a reality [2, 8]. In the lab experimental setting, such failures are often the result of misconfiguration of a component, an unplugged cable, or a bad wireless network connection. One aim of `Drums` is to help you find these bugs more quickly.

In a more long term vision, robust robot controllers should be able to reason about the state of underlying resources and modify their behavior accordingly. `Drums` aims to provide easy-to-use and low-running-cost infrastructure for resource introspection in distributed robot systems.

2. ACHIEVEMENTS

`Drums` is designed to integrate with and work alongside your robot middleware, to make apparent to the user what the interaction of user code, middleware, robot devices and the environment is actually doing to your networked computer system.

`Drums` in its current form provides these key functionalities:

- monitoring of the computation graph created by robot middleware, including run-time changes to the graph
- de-abstraction/de-multiplexing of abstract services and communication channels into native processes and network channels
- dynamic monitoring of these native resources, plus per-host resources such as CPU load, free RAM and disk space
- low-cost aggregation of these data into a central time-series database

The output from `Drums` can readily be visualized and mined with third-party tools based on queries on the time-series database. This helps to increase the operational awareness of the human operator, supervisor or engineer of the robot system. In addition, `Drums` can be used as a low-cost data collection layer for fault detection and diagnosis systems.

[1]http://rosindustrial.org/

Middleware Doctoral Symposium '15, December 07-11 2015, Vancouver, BC, Canada

© 2015 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-3728-1/15/12...$15.00

DOI: http://dx.doi.org/10.1145/2843966.2843974

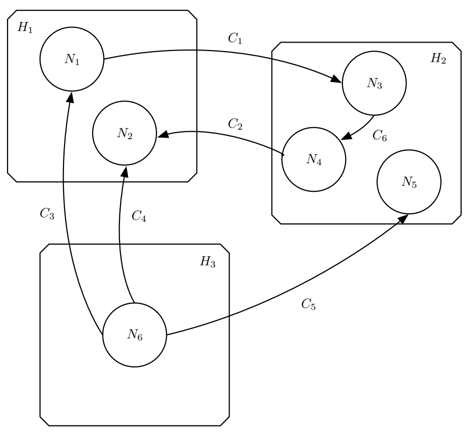

Figure 1: Computation graph extracted from cooperating middleware

A paper describing Drums in details and demonstrating some of its applications was presented at the 2014 Canadian conference on Computer and Robot Vision [5]. This report provides a short overview of that paper. The source code of Drums is freely available at http://autonomylab.org/drums/.

3. APPROACH

3.1 Common Representation Model

We first define a generic model of robot systems independent of the choice of middleware. The model maps closely to native OS resources, and our distributed monitoring infrastructure will gather information from various sources to describe the state of the model over time.

We define the common representation model by removing the abstraction imposed by the middleware architecture, then define a computation graph consisting of operating entities only. The computation graph G consists of a set of n processing nodes $N = \{N_1, N_2, \cdots, N_n\}$ and a set of k directed connections between nodes, $C = \{C_1, C_2, \cdots, C_k\}$ (Fig. 1). Nodes can vary from low level hardware interfaces to high level decision making units and from individual software modules to entire processes. Each connection C_i is a communication link (asynchronous or synchronous, memory based or network based) between two or multiple nodes. Nodes are hosted on a set of m computation units $H = \{H_1, H_2, \cdots, H_m\}$. Each node is hosted by exactly one host. The granularity level of node definition and the type of connections between them defines the type and level of instrumentation needed to monitor the graph.

The granularity level of the computation graph thus depends on the design specifications of the monitoring tool. For Drums these specifications are: 1) Low overhead in terms of resource consumption 2) Ease of deployment 3) No instrumentation beyond what the host operating system and target middleware can provide. To satisfy (3) we chose to limit the granularity level of nodes to operating system processes and network sockets.

To best of our knowledge none of the current robotic middleware platforms provide a distributed way to monitor resource usage of their computation graphs. Some robotic middleware provide ad-hoc monitoring tools for a subset of their computation graph elements. However these tools are either not distributed or do not cover all elements of the computation graph. Examples are ROS's internal statistics about topics and sockets, ROS's third party single computer resource monitoring tool [2] and Urbi's [1] object resource utilization observer.

3.2 Architecture

Drums consists of a statistics collector process and a client library for aggregation. The collector process runs on each host of the computation graph and collects statistics about elements of the graph accessible from the host. It provides an HTTP based interface for runtime configuration of monitoring jobs. In addition, the collector pushes the collected data to the client library for aggregation over a publish/subscribe channel. To utilize Drums, an adapter needs to be written for each target middleware to translate the state of the middleware into a computation graph. Fig. 2(a) depicts the architecture. Drums is written mostly in Python and tested on Linux. Some performance critical tasks such as socket monitoring are implemented as C libraries.

3.2.1 Drums Collector

The Drums collector process is a Python daemon application that collects statistics from elements of the computation graph running on each host. The data are collected using multiple monitoring modules. Each monitoring module runs in a separate process with a configurable sampling interval. Currently there are four types of monitoring modules implemented. New monitoring modules can be written as plugins. The existing modules are: **Process Monitor** which collect information about specific operating system processes running on the host, **Host Monitor** that aggregates resource utilization data about the host computer, **Socket Monitor** which monitors quality of service for each specified socket and **Latency Monitor** which is a multi-target `ping` program to measure host-to-host latency. The collector embeds a server that provides an HTTP based API to register or remove monitoring tasks with a well-defined API for accessing monitoring data both synchronously and asynchronously (through ZeroMQ [4] publish/subscribe sockets).

3.2.2 Drums Client Library

The Drums client library is a Python library that streamlines the process of registering tasks with multiple collectors over the network and subscribe to their publish channels. The client library provides an API to dispatch monitoring jobs to multiple collectors over the network. It aggregates the data collected and published by collectors and relays that data back to its corresponding clients (Fig. 2(a)).

The client library also provides an API to export the aggregated data. Currently the aggregated data is exported to "Whisper"[3]. Whisper is a fixed-size time series database system with flexible and configurable retention policy. Each time series is referred to by a key. Keys are expressed hierarchically e.g `drums.host.process_name.get_cpu_percent`.

The collector process and the client library provide the infrastructure needed for monitoring a distributed computation graph. The last piece of the architecture is the middleware adapter. The middleware adapter is a program that monitors the state of the middleware (hosts, processes

[2]http://wiki.ros.org/rqt_top
[3]http://graphite.wikidot.com/whisper

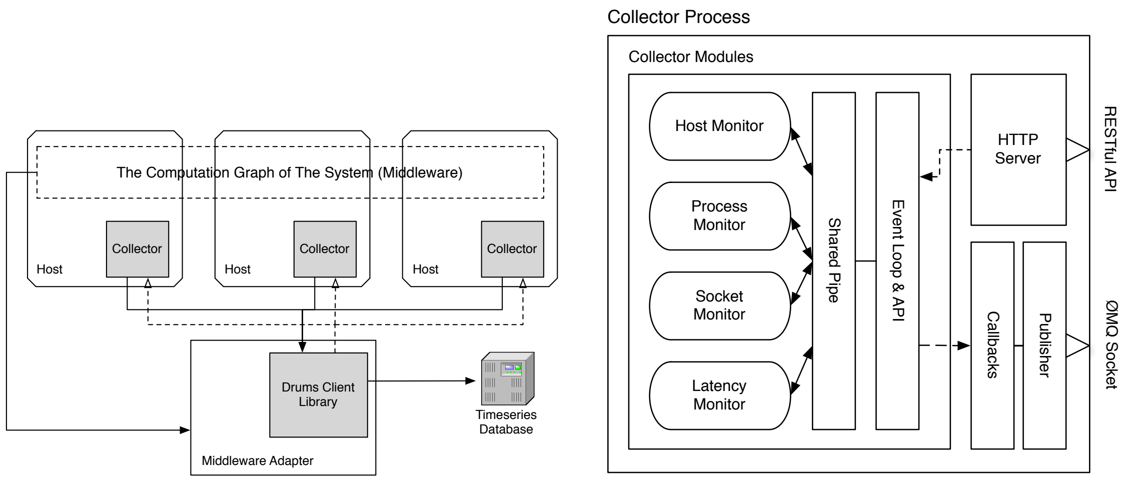

Figure 2: Overall architecture of Drums (left) Architecture of Collector Process (right)

and communication links) to maintain a computation graph. The adapter initiates/removes monitoring jobs for new/deleted elements of the graph by contacting Drums collector(s) that are local to that element of the graph. The adapter acts as a bridge between the middleware and Drums infrastructure. It translates the dynamic abstract state of the middleware into Drums monitoring jobs. For robot middleware with directory services such as ROS and YARP[3], data are obtained directly from the directory service (e.g ROS master or YARP name server). Optionally the adapter can make data collected by Drums available to the middleware, allowing system-level introspection from within the middleware. We developed an adapter for the popular robot middleware, ROS. The adapter wraps Drums client library and monitors the ROS computation graph periodically (by default every 15 seconds) by querying the master.

3.3 Visualization and Anomaly Detection

Since Drums produces too much raw data for users to easily apprehend, we must filter and visualize it before it becomes useful for debugging. For visualization we use a real-time time-series dashboard called "Graphite"[4]. Graphite is a web based front-end to the Whisper database that generates graphs based on customizable queries.

As we do not always know in advance which metrics are important, we employed the data-driven anomaly detection software "Skyline"[5] to watch all metrics at once. Skyline uses the consensus of multiple statistical tests to find discrepancies between recent data points and the recent history for each time series. For accumulative measurements such as counters (bytes, packets, errors) the derivative of the time-series is also fed into Skyline. For the following experiment, the collector process was configured to collect metrics at 1 second intervals. The anomaly detector was configured to run every 5 seconds.

4. FAULTY ROUTER EXPERIMENT

In this demonstration we use Drums to detect and isolate a fault in network equipment. This was an actual fault that interfered with a human-multi-robot interaction experiment [6]. Finding this fault without Drums required many hours of skilled debugging effort.

Three AR-Drone quadcopters were connected via Wifi (802.11n) to a wireless router. From this router, the network connection passed through gigabit Ethernet to a dedicated computer (Intel Core i7 CPU with 8GB of RAM) for each drone. On each computer, realtime vision software subscribed to a high definition video stream from one drone to detect a human and her gestures. In addition, software ran on each computer to control the behavior of the corresponding robot. During initial experiments we observed that the system was not scaling well from two to three robots. The symptom was unresponsiveness of the system when three hosts were running at the same time. A large debugging effort was required to track down the course of the problem. In this demonstration we perform the original experiment with the same hardware. In addition we use Drums to see if we can detect any anomaly and isolate the fault.

After all three robots are powered, we start the vision and control software on each host in turn with a 60 seconds delay in between. We terminate the software after 300 seconds of execution time. The total number of monitored performance parameters was 243. From these metrics we exclude 102 host related metrics such as host's CPU/RAM usage since these may be affected by entities outside the monitoring system and can cause false positives.

Fig. 3(a) shows the number of anomalies detected over the course of the trial. The metrics are broken down into three categories for CPU, memory and I/O related metrics at node (process) level.

There are six major peaks in Fig 3(a). The peaks at 13:41:09 and 13:42:09 correspond to the start of the execution of the second and third host's software respectively. The major peaks at 13:43:09, 13:44:09 and 13:45:09 correspond to the shutdown time of the first, second and third host's software respectively. However the major peak at time 13:42:29 is suspicious. The peak mainly consist of I/O related anomalies at process level. Checking the anomaly detector's log file for the period of 13:42:24 to 13:42:39 reveals

[4]http://graphite.wikidot.com/
[5]https://github.com/etsy/skyline

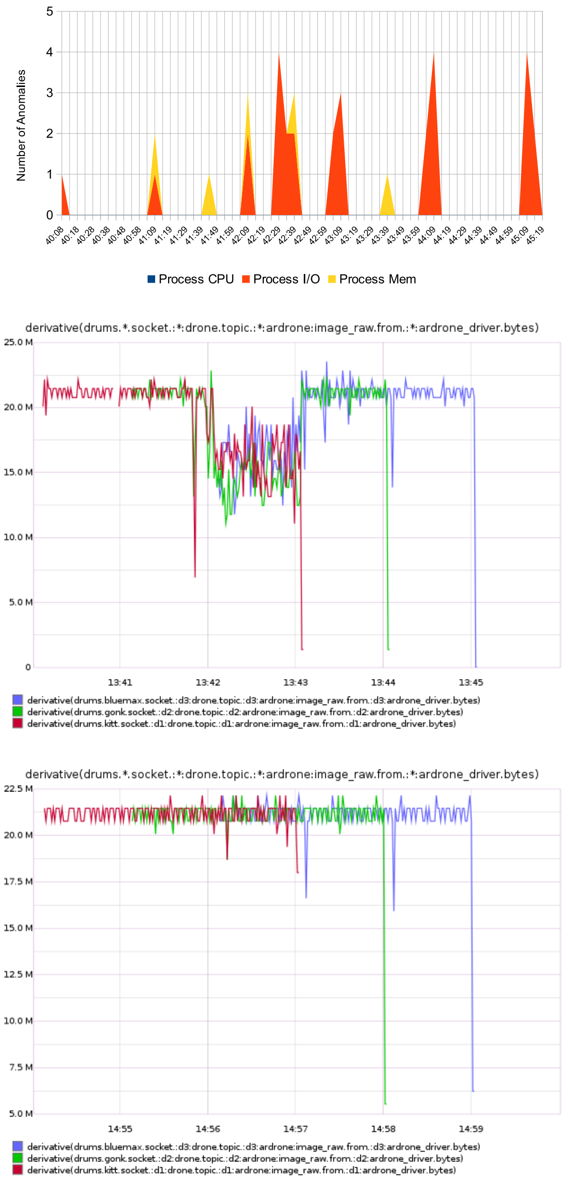

Figure 3: Faulty Router - Excerpt of `Drums` data from the faulty router demonstration. (top) Stacked graph of anomaly breakdown over time for the faulty router demonstration. Time is in minutes past 13:00. The major peaks at 41:09, 42:09, 43:09 and 44:09 are caused by adding and removing robots from the network. The peaks at 42:29 – 42:39 are suspicious. (middle) Shows the socket traffic with the defective router. (bottom) Shows the same data for the nominal router.

that there are multiple anomalies in the AR-Drone driver's image publishing sockets for all three hosts. Fig. 3(b) shows the bandwidth usage graph by these sockets generated by Graphite. The graph shows that when all three software stacks are running simultaneously the traffic of the image publisher's socket drops significantly for all three robots. With this knowledge we can narrow down the search for the root of the problem to two possible cases: wireless interference between robots or a problem with the router. We replaced the router with a similar model from a different manufacturer and repeated the experiment. Fig. 3(c) shows the resulting bandwidth utilization graph on image publishers' sockets for the new experiment. Comparing these two figures, we can conclude that the router was the source of the problem.

5. CONCLUSION AND FUTURE WORK

We introduced a lightweight distributed monitoring tool for robots and demonstrated its applications in a practical demonstrations. `Drums` unpacks the abstraction layer presented by the middleware and maintains a corresponding graph that maps into monitorable system entities. We used a generic data-driven anomaly detector to draw user's attention to fewer metrics. Of course, anomalous behavior of a metric is not necessarily due to a fault: false positives can occur. Furthermore, metrics with anomalies only provide clues about possible faults in the system. Future work includes developing a custom visualization and fault detectors for distributed robot systems that take full advantage of `Drums`. A long term interesting research direction is to use `Drums` as an introspection tool inside robot controllers, adapting robot behavior to internal system conditions that have previously been difficult to observe across the network.

6. REFERENCES

[1] J.-C. Baillie. Urbi: Towards a universal robotic low-level programming language. In *Intelligent Robots and Systems (IROS). IEEE/RSJ International Conference on*, pages 820–825. IEEE, 2005.

[2] J. Carlson and R. R. Murphy. How UGVs physically fail in the field. *Robotics, IEEE Transactions on*, 21(3):423–437, 2005.

[3] P. Fitzpatrick, G. Metta, and L. Natale. Towards long-lived robot genes. *Robotics and Autonomous Systems*, 56(1):29–45, Jan. 2008.

[4] P. Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly, 2013.

[5] V. Monajjemi, J. Wawerla, and R. Vaughan. Drums: A middleware-aware distributed robot monitoring system. In *Computer and Robot Vision (CRV), 2014 Canadian Conference on*, pages 211–218, May 2014.

[6] V. Monajjemi, J. Wawerla, R. Vaughan, and G. Mori. HRI in the sky: Creating and commanding teams of uavs with a vision-mediated gestural interface. In *Intelligent Robots and Systems (IROS), IEEE/RSJ International Conference on*, pages 617–623, 2013.

[7] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an open-source Robot Operating System. *ICRA workshop on open source software*, 3(3.2), 2009.

[8] G. Steinbauer. A Survey about Faults of Robots Used in RoboCup. In *16th Annual RoboCup International Symposium*, pages 344–355, 2013.