“Drums”: a Middleware-Aware Distributed Robot Monitoring System

Valiaallah (Mani) Monajjemi, Jens Wawerla and Richard Vaughan
School of Computing Science
Simon Fraser University
Burnaby, BC, Canada
{mmonajje, jwawerla, 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 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 worked examples 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

I. INTRODUCTION

This paper describes Drums, a new tool for monitoring and debugging distributed robot systems. We describe the need for such a tool as a complement to robot middleware systems, and show examples of its use in the real-world scenarios that motivated us to develop it. Our group develops distributed multi-robot systems, and Drums provides us an important component for testing, debugging and run-time quality-of-service monitoring that was previously missing. The name “Drums” is nearly an acronym of Distributed Robot Monitoring System. It is freely available from http://autonomylab.org/drums/.

A. Background 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 [1], 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.

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], [3]. 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. We argue below that no system existed with the required functionality in a convenient form.

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 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.

B. Observing Faults in Robot Systems

The general approach to detecting and diagnosing faults is to observe (monitor) components of the system and detect deviations from expected behavior. Deviations can be found using quantitative or qualitative models (model based approach) or by comparing observations with data collected during the system’s nominal operating conditions (data driven approach).

Various types of observations have been proposed: Verma et al. [4] used sensor measurements as observations and particle filters as estimators to track the robot state and detect faults in the drive system of a simulated Mars rover. Steinbauer et al. monitored [5] the frequency of method invocations and the order and timing of service calls in a mobile robot’s control software using model based reasoning [6]. Using the same diagnosis technique, Kleiner et al. [7] observed communication patterns between software components of a robot navigation software to detect faults. Similarly, Golombek et al. [8] used kernel density estimation to fit a probabilistic model on temporal communication patterns among components of a robot systems under normal operating conditions. During runtime, the fitness of new observations is assessed according the model to detect faults.

While promising, these methods are not in wide use. One issue is that these ad-hoc implementations are not easily portable to new robot systems. Drums provides a portable substrate on which to base observers for fault detection or other introspection systems. Data collected by Drums can be used directly as observations, or indirectly as basis to design sophisticated observers. Zaman et al. [9] addressed this portability issue by designing a set of observers for ROS to monitor hardware drivers, the state of nodes, publication frequency, and the resource usage of hosts. The latter two are similar in concept to our monitoring system. However in its current implementation, Zaman’s system does not support individual process monitoring and dynamic discovery of the underlying computation graph (as defined below). Furthermore, in contrast to our passive socket monitoring (Sec. III-B), Zaman’s topic observers gather statistics actively by subscribing to corresponding publishers in the ROS network. In the worst case this increases the number of edges in the computation graph (section II-A) by the factor of 2.
between monitoring physical network sockets and monitoring data exchange over shared memory.

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.

B. Monitoring the Computation Graph

Distributed monitoring software is widely used to monitor distributed computing platforms such as IT infrastructures [10], computing clusters [11], grids [12] and clouds [13], [14]. However, none of the available distributed monitoring architectures support the process and socket granularity level required to monitor the proposed computation graph. The problem with most of these systems is that the lowest supported level of granularity is at the host level and the network connections between hosts. The latter is important because robot control software is often a data-intensive network connections between hosts. The former is important to observe the health of the distributed application [15] and monitoring the connections because robot control software is often a data-intensive network connections between hosts. The latter is important supported level of granularity is at the host level and the problem with most of these systems is that the lowest required to monitor the proposed computation graph. The architectures support the process and socket granularity level [14]. However, none of the available distributed monitoring [10], computing clusters [11], grids [12] and clouds [13], distributed computing platforms such as IT infrastructures [10], computing clusters [11], grids [12] and clouds [13], [14]. However, none of the available distributed monitoring architectures support the process and socket granularity level required to monitor the proposed computation graph. The problem with most of these systems is that the lowest supported level of granularity is at the host level and the network connections between hosts. The latter is important because robot control software is often a data-intensive network connections between hosts. The former is important to observe the health of the data flow. For extensible systems such as Ganglia [11], Nagios [10] or Collectd2, it is possible to develop custom monitoring plugins for finer levels of granularity. However monitoring the computation graph for robot systems has one characteristic that is not met by these systems: the graph created by the middleware is different between each execution, and may change during runtime. For example a publish/subscribe or an event channel managed by the middleware may use different TCP/UDP port numbers between successive executions. New nodes may be spawned or new communication channels may be created during the execution by the middleware. Previously mentioned systems rely on static configuration files that are not designed to be configurable during runtime.

To best of our knowledge none of the current robotic middleware provide a distributed way to monitor resource usage of their computation graph. 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 3 and Urbi’s [16] object resource utilization observer.

The rest of this paper is organized as follows. We will present the distributed monitoring architecture and its technical details in section III. Section IV describes applications of Drums in two robot scenarios. Finally we discuss possible extensions and future directions in section V.

III. Method

A. 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 a 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 publisher/subscriber 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.

B. 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: The process monitor collects information about specific operating system processes running on the host. Information such as CPU and RAM utilization of the process and its spawned threads. The process monitor maintains a list of Process IDs that need to be monitored. This list is modifiable during runtime upon request from the client. The process monitor uses the cross platform psutil4 library to collect the data.

Host Monitor: The host monitor collects aggregate resource utilization data about the host computer. Information such as CPU, RAM, I/O and network utilization. Similar to process monitor, this data is collected using psutil.

Socket Monitor: The socket monitor is built on top of libpcap [17], a packet capture library. Monitoring jobs are specified by the client as tuples of \(<\text{protocol}, \text{direction}, \text{port}>\) (e.g. \(<\text{tcp}, \text{src}, 8000>\)) which will be converted to libpcap’s filters. Packets that match any of these filters are captured. The captured packet headers are then analyzed to retrieve the packet length. For each filter, the total number of packet lengths are stored and passed to client once per interval.

Latency Monitor: The latency monitor is a multi-target ping program. The process maintains a list of target hosts to monitor. Once per interval the latency monitor sends a

\(^2\)http://collectd.org/

\(^3\)http://wiki.ros.org/rqt_top

\(^4\)https://code.google.com/p/psutil/
number of ICMP packets to specified hosts to calculate the round-trip delay between the host and targets. The average, minimum and maximum of the measured ping times are stored for each target. The number of packets sent and the intervals between them are configurable.

The collector includes a server that provides an HTTP based API to register or remove monitoring tasks. For example clients can send a POST HTTP request to http://host:8001/drums/monitor/pid/344 to register a process monitoring task on host for the process with PID of 344. The returned HTTP response code determines if the task registration was successful. Sending a DELETE request to the same URI removes the monitoring job. There is a well defined API for all monitoring modules.

The collector provides both synchronous and asynchronous data retrieval methods. Clients can poll the daemon to retrieve the latest measurements by making HTTP GET requests at the URI associated with each job. The returned data is serialized into JavaScript Object Notation (JSON) format. The daemon also publishes data over a ZeroMQ publish socket. ZeroMQ [18] is a cross-platform, lightweight and high performance message passing middleware for distributed applications (See [19] for a performance comparison with other message passing middleware). Clients can connect to this publish socket to subscribe to particular measurement sources. Data is pushed to subscribers when it is received from the collector, serialized in “msgpack”\(^5\) binary serialization format. The address of the publisher socket and the key for each monitoring task is provided to clients via the HTTP response sent back by the server when a job is first initiated.

\(^5\)http://msgpack.org/

C. Drums Client Library

ZeroMQ and msgpack both have bindings for almost all modern programming languages and mature libraries are available for using HTTP APIs and JSON-encoded data. Hence, it is straightforward to develop applications that use Drums collector services. As a reference implementation, we developed a Python client 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”\(^6\), a time-series database. 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. Time-series are stored in Whisper for a configurable duration (e.g days).

D. Drums ROS Adapter

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 and communication links) to maintain a computation graph. The adapter
Table I
AVERAGE RESOURCE USAGE OF DRUMS COMPONENTS DURING THE DEMONSTRATIONS.

<table>
<thead>
<tr>
<th>Module</th>
<th>CPU Util.</th>
<th>Mem Util.</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collector API</td>
<td>&lt;1%</td>
<td>2MB</td>
<td>150-160 Kbps</td>
</tr>
<tr>
<td>Process Monitor</td>
<td>&lt;1%</td>
<td>13MB</td>
<td>-</td>
</tr>
<tr>
<td>Host Monitor</td>
<td>&lt;1%</td>
<td>13MB</td>
<td>-</td>
</tr>
<tr>
<td>Latency Monitor</td>
<td>&lt;1%</td>
<td>12MB</td>
<td>-</td>
</tr>
<tr>
<td>Socket Monitor</td>
<td>1%-2%</td>
<td>19MB</td>
<td>-</td>
</tr>
<tr>
<td>ROS Adapter</td>
<td>1%-2%</td>
<td>22MB</td>
<td>-</td>
</tr>
</tbody>
</table>

Resource Usage (392 metrics)

<table>
<thead>
<tr>
<th>Module</th>
<th>CPU Util.</th>
<th>Mem Util.</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collector API</td>
<td>&lt;1%</td>
<td>9MB</td>
<td>290-310 Kbps</td>
</tr>
<tr>
<td>Process Monitor</td>
<td>&lt;1%</td>
<td>12MB</td>
<td>-</td>
</tr>
<tr>
<td>Host Monitor</td>
<td>&lt;1%</td>
<td>11MB</td>
<td>-</td>
</tr>
<tr>
<td>Latency Monitor</td>
<td>&lt;1%</td>
<td>12MB</td>
<td>-</td>
</tr>
<tr>
<td>Socket Monitor</td>
<td>&lt;1%</td>
<td>14MB</td>
<td>-</td>
</tr>
<tr>
<td>ROS Adapter</td>
<td>4%-5%</td>
<td>24MB</td>
<td>-</td>
</tr>
</tbody>
</table>

initiates/removes the 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 [20], data are obtained directly from the directory service (e.g. ROS master or YARP name server). Optionally the adapter can make the 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. Processes (nodes) spawned by ROS and their host platforms are identified by querying the master. For each host its hostname and/or IP address is determined. This information is used to contact Drums collectors and to initiate host, process (node) and latency monitoring tasks. Information about the sockets used by ROS is obtained directly from the nodes using an extended ROS client API\(^7\). The aggregated information is published as hierarchical key-value pairs via ROS diagnostic topic which is monitored by ROS’s diagnostics infrastructure. The data can be viewed using the graphical user interface provided by ROS or be analyzed using plugins.

IV. DEMONSTRATION

In this section we show how Drums can assist in detecting problems in two ROS-based robot systems. In the first demonstration we show how Drums can be used to isolate a fault in network equipment. In the second, we demonstrate how to detect excessive resource usage caused by elements of the computation graph.

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 realtime time-series dashboard called “Graphite”\(^8\). Graphite is a web based front-end to the Whisper database that generates graphs based on customizable queries. Fig. 3 shows an example resource-monitoring dashboard, where important measurements are chosen manually for near-real-time display. The user can explore the data by entering queries in this GUI.

But we do not always know in advance which metrics are important, so we employed the data-driven anomaly detection software “Skyline”\(^9\) 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 all the following experiments, the collector process was configured to collect metrics at 1 second intervals. The anomaly detector was configured to run every 5 seconds.

A. Faulty Router

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 [21]. Finding this fault without Drums required many hours of skilled debugging effort.

Three AR-Drone quadrocopters 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 every computer, realtime vision software subscribed to a high definition video stream of one drone to detect a human and her gestures. In addition, software

\(^{7}\) We extended the ROS client API to include low level socket information. The extended API has been scheduled to be included in the next ROS release.

\(^{8}\) http://graphite.wikidot.com/

\(^{9}\) https://github.com/etsy/skyline
was running on each computer to control the behavior of the corresponding robot. During the 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 one by one 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 from fault detection. The reason for that is host related metrics such as host’s CPU/RAM usage may be affected by entities outside the monitoring system, therefore they may cause false positives.

Fig. 4 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 4. 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 that there are multiple anomalies in the AR-Drone driver’s image publishing sockets for all three hosts. Fig. 5(a) 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. 5(b) 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.

Summarizing the utility of Drums in this scenario: Drums automatically enables monitoring of every process, network connection and host involved in maintaining the ROS abstraction of networked services communicating over topics. Due to the robustness of the ROS design, the topics continued to work when the router faults were encountered and ROS reported no errors. Yet the underlying network performance changed on individual socket pairs, and this anomaly was automatically detected by Skyline watching Drums data, drawing the user’s attention to the relevant system parameters.

We also measured the overhead of different components of the monitoring system during this experiment. Table I includes the average CPU, memory and bandwidth utilization of various components of the monitoring system. We can see that Drums adds only marginal overhead.

B. Monitoring Resource Usage

In the second demonstration, we show how data collected by Drums can be used to detect excessive and anomalous resource consumption. The testbed is a Husky A200 ground robot equipped with camera, inertial measurement unit, GPS and a laser scanner. The task of the robot is to detect a human and follow her while doing safe navigation. We injected a memory leak bug in the person following code to simulate a software bug that degrades the performance of the whole system by excessive memory consumption. We also added a number of logging statements in a tight control loop of the emergency stop routine. In ROS, log messages aggregate in
a central process called `rosout`. Although log messages are helpful tools for debugging, unnecessary log messages or log messages with high publication frequency cause unnecessary network overhead.

Figure 6 shows the breakdown of detected anomalies over the course of a human following scenario. The robot is initially in “waiting for human” state until the human enters the field of view at time 16:53:08. This triggers a change in the robot’s behavior and it starts the “follow the human” subroutine. The robot follows the human for 20 seconds until the human stops walking and the robot comes too close at time 16:53:28. This triggers the “emergency stop” mode. As shown in Fig. 6, the number of anomalies start to rise as the robot enters the “follow the human” state. There are two major process specific anomalies during this state. The first one at 16:53:13 is mostly memory related. Log files reveal anomalies in memory usage of the `people_follow` node. Figure 7(a) shows the graph for the `people_follow` resident memory usage which clearly shows a memory leak during the “follow the human” period. The second peak at 16:53:23 shows I/O related anomalies. Investigating the log files shows that anomalies were caused mainly by nodes involved in safe navigation of the robot. There is also a major process I/O peak during the “emergency stop” state at time 16:53:38. Anomalies logged for that time period include anomalies related to messages published from `people_follow` node to `rosout` such as `socket.rosout.topic.rosout.from.people_follow` (shown in Fig. 7(b)).

The total number of recorded metrics in this demonstration was 392. From these we exclude 68 host level metrics. All nodes except the ROS adapter were running on the
V. CONCLUSION AND FUTURE WORK

In this paper we introduced a lightweight distributed monitoring tool for robots and demonstrated its applications in two practical demonstrations. Drums unpacks the abstraction layer presented by the middleware and maintains a corresponding graph that maps into monitorable system entities. We showed that by monitoring the computation graph we can detect excessive resource usage, anomalies and faults in robot systems. We used a generic data-driven anomaly detector to draw the user’s attention to fewer than 10 suspicious time series of around 300 recorded. 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.

VI. ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council (NSERC) through the NSERC Canadian Field Robotics Network (NCFRN).

REFERENCES