Supporting Information

The yaq Project: Standardized Software Enabling Flexible Instrumentation

*Kyle F. Sunden, Daniel D. Kohler, Kent A. Meyer, Peter L. Cruz Parrilla, John C. Wright, and Blaise J. Thompson**

Department of Chemistry, University of Wisconsin–Madison
1101 University Ave., Madison, Wisconsin 53706

*Corresponding Author
email: blaise.thompson@wisc.edu
phone: (608) 263-2573

Contents

1 Timing and interface responsiveness

Each message call over the interface returns rapidly, ensuring that client applications are not blocked. Some mes- sages initiate long-running operations, e.g. motor motion and sensor measurement. Separate messages are provided to re- trieve results from e.g. sensor measurement.

1.1 `is_busy` for orchestrating order of operations

In order to know how long to wait, all daemons provide a message called "is_busy", which returns "true" while the long running action is not complete, and "false" once it is finished. Additionally, multiple clients can communicate with the same daemon si- multaneously. A complex instrument may involve multiple operators watching sensor data in real time, while one pro- gram is orchestrating the hardware and recording the data.

1.2 Scaling of large messages

While most messages are intended to be short and quick to respond, large single messages will take appreciable time to transfer data from daemon to client over TCP. One common usecase where a user may wish to transfer a large message over TCP would be camera data. Cameras can have large arrays which contain the scientifically interesting data.

While is flexible enough to represent such arrays, it was not specifically designed with large cameras as the primary usecase. As such, performance does suffer above about 1 Megapixel. Figure S1 shows the relationship between number of pixels and time for a message to read the array. Below about 2^{18} (250,000) pixels, there is virtually no dependence on size, with the standard overhead of a message dominating the time for transfer. remains usable for up to approximately 2^{20} (1,000,000) pixels. "Usable" is a relative term which will depend greatly on context. Here we generally mean "seems responsive to a user trying to have feedback on human timescales". Applications which require high speed, high throughput cameras are unlikely to be suitable for even with relatively small cameras. This test was conducted using 32-bit integer arrays.

Common camera sizes (512, 512), (1024, 1024), and (1024, 4096) are highlighted in S1 to provide context.

There are some strategies that could be used to mitigate the performance problems of large cameras, but they have not been implemented because large arrays remain an edge case among current users. Such strategies could include enabling compression, using transport layers other than TCP, using regions-of-interest (ROIs) to limit array sizes, and writing arrays locally to disk and providing mechanisms for retrieval out of band. The Avro Specification [1] specifically provides for compression codecs in the case of storing to disk, but there is nothing preventing the RPC pipe from similarly encoding the data, provided both ends of the communication support it. This would allow larger arrays to be transported using fewer bytes over TCP, which would likely improve performance for large arrays where TCP transport is the bottleneck. While TCP is the only currently supported transport layer for , this limitation could be lifted in the future. TCP was chosen as the preferred starting point because it is ubiquitous, being available on every desired platform and implementation language. One alternative that would be relatively easy to support would be Unix domain sockets (UDS). UDS has an extremely similar interface to TCP sockets, and are handled in Python by the same standard library module. UDS is potentially more performant, but as the name suggests are limited to Unix-like systems (i.e. it will not work on Windows). ROIs are implemented on the handful of cameras currently in the ecosystem [2, 3], however we are still in the experimentation phase and have not standardized ROI configuration into a trait. When only a subset of the total camera area is interesting, this is one way to limit array size over the interface. Finally daemons could be implemented to write arrays locally to disk, rather than transferring over the interface directly. This has not

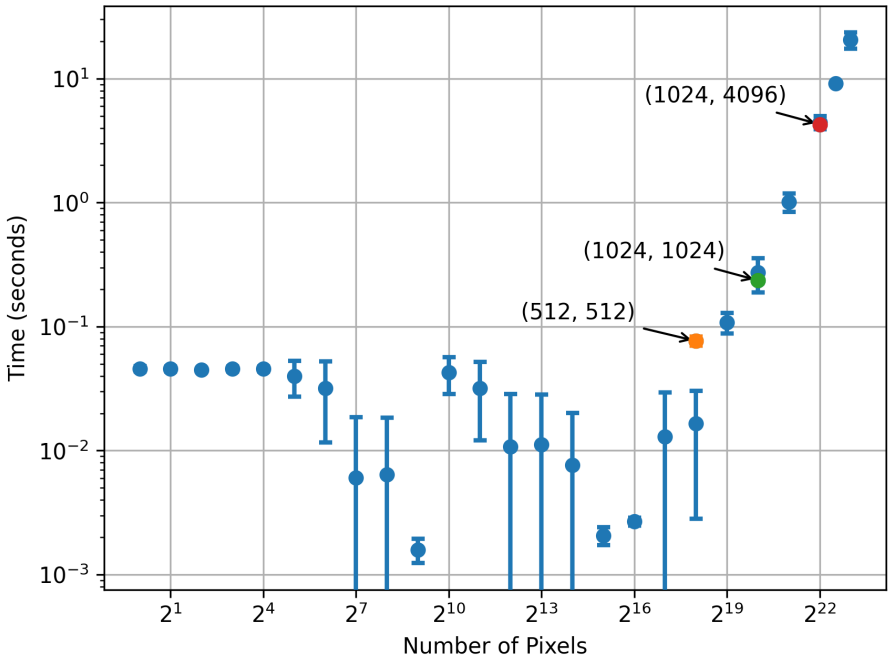

Figure S1: Scaling of transport time as a function of number of pixels for 32-bit integer arrays. Both the x- and y-axes are logarithmically scaled. Common camera sizes (512, 512), (1024, 1024), and (1024, 4096) are highlighted to provide context for camera users.

yet been implemented, and would require clients to have knowledge of how to retrieve and display the images collected, but there is nothing preventing this if throughput is a limitation. If such behavior became a standard feature desired for cameras, it should be encapsulated in a trait to provide a consistent interface.

To reproduce this figure, you will need the libraries specified in "figures/requirements.txt". The scripts for generating this figure, including the daemon, are located in "figures/scaling". To generate the data, run the daemon using `python scaling.py --config config.toml` from with in the "scaling" folder. Then run `python scaling-data-gen.py`, which will communicate with the running daemon and produce a CSV written to standard out. The result can be saved to `scaling.csv`. To visualize the results, run `python plot-scaling.py`, which will read the CSV and produce the image.

2 Package size analysis

The raw data for package size analysis was collected using Tokei[4], a command line tool for analysing line lengths of source code. It includes breaking down line length into "code", "comments", and "blank" lines. The package size data was curated into `figures/lines_histogram.txt`, which includes annotations as to the "type" and "class" for each file. The "type" annotation is one of "stub" (meaning that the bulk of the implementation is in another file), "normal" (an individual daemon with the bulk of its implementation), "protocol" (an implementation the provides for multiple daemons, such as those referred to by "stub"s, or "serial" (the implementation of a serial protocol rather than a full fledged daemon). The "class" refers to the primary function of the daemon and is one of the following values: "is-sensor" for detectors, "has-position" for setable hardware, "serial" (identical to "type" annotation), and "other" for daemons which do not fit in the above categories.

The plotting script `figures/lines_histogram.py` can be used to generate the figure from this text file. The figure uses the "code" lines to generate a histogram of file sizes in the python implementations. This is a measurable proxy for the amount of work that implementing an individual daemon entails, though of course cannot capture the work involved in learning the lower level interface.

References

[1] *Apache Avro Specification*. https://avro.apache.org/docs/1.11.1/specification/. Accessed: 2022-10-02. 2022.

[2] *yaqd-andor*. https://pypi.org/project/yaqd-andor. Accessed: 2023-02-05.

[3] *yaqd-pi*. https://github.com/yaq-project/yaqd-pi. Accessed: 2023-02-05.

[4] *Tokei*. https://github.com/XAMPPRocky/tokei. Accessed: 2023-02-05.