



The yaq project: Standardized software enabling flexible instrumentation

Cite as: Rev. Sci. Instrum. 94, 044707 (2023); doi: 10.1063/5.0135255

Submitted: 17 November 2022 • Accepted: 28 March 2023 •

Published Online: 12 April 2023



Kyle F. Sunden,  Daniel D. Kohler,  Kent A. Meyer,  Peter L. Cruz Parrilla,  John C. Wright, 
and Blaise J. Thompson^{a)} 

AFFILIATIONS

University of Wisconsin–Madison, Madison, Wisconsin 53706, USA

^{a)} Author to whom correspondence should be addressed: blaise.thompson@wisc.edu

ABSTRACT

Modern instrumentation development often involves the incorporation of many dissimilar hardware peripherals into a single unified instrument. The increasing availability of modular hardware has brought greater instrument complexity to small research groups. This complexity stretches the capability of traditional, monolithic orchestration software. In many cases, a lack of software flexibility leads creative researchers to feel frustrated, unable to perform experiments they envision. Herein, we describe Yet Another acQuisition (yaq), a software project defining a new standardized way of communicating with diverse hardware peripherals. yaq encourages a highly modular approach to experimental software development that is well suited to address the experimental flexibility needs of complex instruments. yaq is designed to overcome hardware communication barriers that challenge typical experimental software. A large number of hardware peripherals are already supported, with tooling available to expand support. The yaq standard enables collaboration among multiple research groups, increasing code quality while lowering development effort.

Published under an exclusive license by AIP Publishing. <https://doi.org/10.1063/5.0135255>

I. INTRODUCTION

Instrumentation development is a key part of the scientific enterprise. Novel instruments are typically constructed of many individual components that are both purchased and home-built. Orchestration software must communicate with each hardware component in the course of a scientific experiment. Orchestration can involve the utilization of many interfaces: NI DAQmx,¹ SCPI,² ModBus,³ PICam,⁴ and Thorlabs APT,⁵ among many others. The challenge of integrating all of these interfaces is a frustrating piece of the modern instrument development process. Weeks can be spent just integrating one new component into an existing project. In small academic labs, these software interfaces are typically created by student researchers without software development experience. Student researchers rarely focus on software reusability, and a lack of maintenance and documentation can make such software more difficult to use as time goes on. Scientists may struggle to rapidly innovate their experimental design when each hardware addition requires major software development.

Some large user facilities have addressed interface complexity via the adoption of unified standards, such as EPICS⁶ or TANGO.⁷ The unified standards define a network interface for any hardware component. Orchestration software can target these unified standards for reading and writing hardware states. Small background services are written to translate the myriad component interfaces into the standard EPICS IOCs and TANGO Devices. These programs are performant, open source, and have huge libraries of existing hardware interface support but require expert management to set up and provide descriptions via a separate server program. In our experience, EPICS and TANGO do not scale well to single-investigator lab environments.

As smaller research labs have grown in experimental complexity, many individual labs have created domain-specific orchestration software. In the last few years, several open-source projects by-and-for small-scale experimentalists have grown in popularity.^{8–15} Although this growth is encouraging, many of these are limited by their focus on particular types of hardware or particular experimental domains. More work needs to be done to catalog, compare, and

contrast the large number of open-source projects that now exist. Most small custom research instrumentation continues to rely on monolithic software that has hard-coded interface support for each particular connected device. These monolithic applications tend to be inflexible and difficult to develop.

We have created a new network-based communication standard for scientific instrumentation, *yaq* (Yet Another acQuisition).¹⁶ This standard borrows the most important ideas from established projects used by large user facilities while retaining the simplicity appropriate for small research labs. We have built this standard to be self-describing, portable, and reusable wherever possible. Via the *yaq* interface, we provide easy-to-use hardware support for scientists to use with their choice of experiment orchestration software.

Here, we discuss the design of the *yaq* standard in the context of challenges facing instrument designers. First, we discuss how particularly challenging hardware interfaces can become seemingly insurmountable barriers to software control. Next, we discuss how inflexible orchestration software can limit experimental creativity. Then, we focus on challenges that arise when enhancing or modifying existing instruments with new hardware. Finally, we discuss the heavy software maintenance burden that many instrument designers face. In each case, we will highlight how the *yaq* project is designed to alleviate that challenge. Several case studies provide a view into the flexible ways that *yaq* can be applied to perform different scientific experiments. This paper provides an overview of the concepts and motivations behind *yaq*. Refer to the *yaq* website for a formal specification of the *yaq* standard.¹⁶

II. HARDWARE INTERFACE CHALLENGES

In this section, we describe the architecture of the *yaq* framework in light of three major barriers that we have encountered in scientific instrumentation development. First barrier: Multiple interfaces are used to communicate with each component of the system. A fully automated system must be able to use all of these interfaces, a daunting task for scientists who do not specialize in software development. Second barrier: Certain specialty hardware has inconvenient interface requirements. A camera will only work with an obsolete interface card and drivers for Windows XP. A data acquisition manufacturer provides an Application Programmer Interface (API) that only works in Python 3.7. A graduate student wishes to drive several stepper motors using a Raspberry Pi. Third barrier: Some hardware interfaces are blocking. A graphical user interface stalls while waiting for a camera to collect data. Custom orchestration software needs to be closed before the manufacturer's configuration software can be used. A graduate student finds themselves needing to master advanced concepts in concurrency to orchestrate many motors performantly.

Figure 1 diagrams the *yaq* architecture. Here, we show three different computers connected via an Ethernet network. The top and bottom computers are connected to monitors for interactive use, whereas the middle computer is only accessible via the networks. This diagram might represent a complex scientific instrument involving several operator terminals as well as embedded computers. At the top, a single computer is connected to four hardware peripherals through RS232, TTL, USB, and PCI as indicated by the colored lines. That same computer is running four

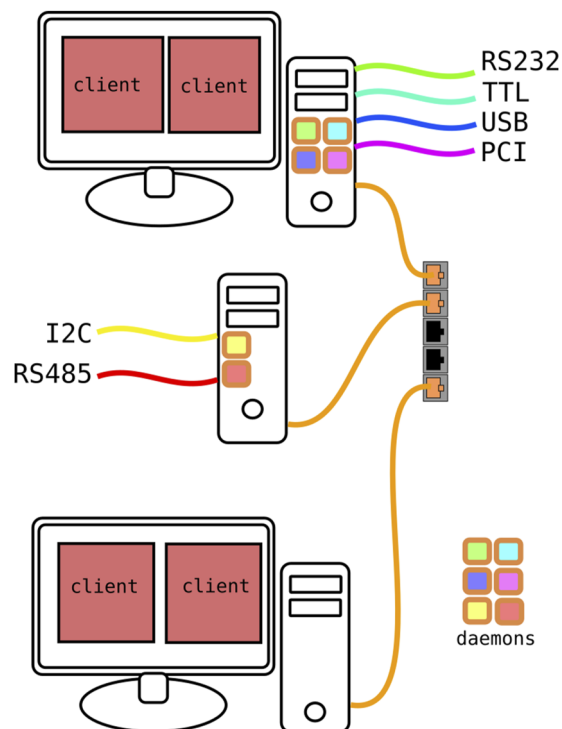


FIG. 1. A simplified network diagram for a hypothetical instrument. Three computers are connected via orange Ethernet cables and a network switch (middle right). Some of the computers are also connected to hardware peripherals via one of the six interfaces (colored lines). For each interface, the computer runs a *yaq* daemon (small colored squares). The top and bottom computers are also running *yaq* client applications.

separate programs, one for each peripheral. These small, targeted, programs are managed by the operating system and run in the background. It is conventional to call such programs “daemons.”¹⁷ The middle computer is connected to two additional peripherals and runs daemons for each. Besides communicating with the hardware peripheral, each daemon can communicate with other programs, “clients,” through the network. The four client programs shown in Fig. 1 can each communicate with all six hardware peripherals shown. As an example, a client running on the bottom computer could communicate with the RS232 peripheral shown in green via the following path: client ↔ network switch ↔ top computer ↔ daemon ↔ hardware peripheral. This powerful architecture can be used on a single computer or used across many networked computers, including fully remote operator interfaces. This client-server architecture offers similar network capabilities to EPICS and TANGO. As we will show, the usage of standards and the creation of tooling make this architecture accessible to instrument builders outside of large facilities.

In *yaq*, communication between daemons and clients is performed over TCP/IP using Apache Avro RPC.¹⁸ Avro provides an agreed-on standard for efficient serialization of data and method calls from a remote (client) process. Practically, the *yaq* interface looks like a collection of methods or functions, which Avro

calls “messages.” Each message has defined input parameters and output return types. A sensor might implement a message called “get_measured,” which takes no parameters and returns a dictionary mapping channel names to the numeric or array data. A motor would implement a pair of messages for setting and reading back the motor position: “set_position(float position) → null” and “get_position() → float.” Messages make up the lowest level functionality of the *yaq* interface. Each daemon supports a collection of messages for its unique functionality, called the “protocol.” When a client first connects to a daemon over TCP/IP, the daemon provides a complete description of its own protocol. We, therefore, refer to the daemons as “self-describing.”

Each individual communication between the client and the daemon involves one message being requested by the client and the response returned from the daemon. If multiple messages are sent simultaneously, the daemon processes one message at a time. The protocol is preemptive: in the case of a conflict, the daemon will obey the last message to arrive. By being completely open to messages from multiple clients, *yaq* makes it possible to accidentally supersede a message. This has not become a problem for our usage where instruments are controlled by single scientists who can clearly see all of the clients interacting with their system. Typically, there is just one client sending “set_position,” and many clients polling “get_position” to record data or display real-time information. Users may choose to implement access control systems, but we do not foresee building access control into the *yaq* protocol. This is similar to how access control works, in practice, at some large facilities.

yaq introduces a concept called “traits,” which are collections of related messages that are shared among multiple protocols. Motors implement the “has-position” trait, which defines “set_position,” “get_position,” and “get_units.” Sensors would implement the “is-sensor” trait, which defines “get_measured,” “get_channel_names,” and “get_channel_units.” Protocols that implement a trait must support all of the messages from the trait. Traits are compositional: a given protocol may implement several traits at once. For example, a protocol representing a monochromator with several gratings might implement both “has-position” and “has-turret,” where the latter defines special messages for choosing which grating is used. Importantly, specific protocols can also implement arbitrary additional messages that are not defined by any trait.

We have carefully defined *yaq* traits to maximize interoperability while not excluding hardware with unusual features. New traits can be discussed by the community through the *yaq* enhancement proposal system. It is crucial that multiple hardware interface examples be considered when defining traits. Configuration that is unique to a single interface is best handled through protocol-specific messages. For example, our protocol for interfacing with a specific data acquisition card provides a message for setting the segment count. Unique messages are available through script-based and graphical clients, but the usage of such messages in scripts naturally limits portability.

We now describe how the *yaq* architecture addresses the three hardware interface barriers described at the beginning of this section. The first hardware interface barrier: Multiple incompatible interfaces are used to communicate with each component of the system. *yaq* provides a unified TCP/IP interface to all hardware peripherals based on the well-described Avro RPC protocol. The

trait system was introduced in pursuit of our primary goal of easing the client development process. Clients can trust that protocols that implement a given trait will behave in similar ways. The standardized *yaq* interface presents the same set of interactions for client-side scientific code, simplifying the experience of using hardware.

The second hardware interface barrier: Certain specialty hardware has inconvenient interface requirements. Since *yaq* enables multiple machines, any hardware requirements can be addressed by putting a machine for that specific hardware on the network. The Raspberry Pi, which drives several stepper motors, can be placed onto a private network to communicate with the primary instrument computer using *yaq*. Because each *yaq* daemon is running in its own process, the software environment can be tailored to its needs. A client running up-to-date Python 3.11 communicates seamlessly with a daemon running Python 3.7.

The third hardware interface barrier: Some hardware interfaces are blocking or slow. Experimental orchestration often means doing several things at once, e.g., simultaneously moving several motors. To accomplish this task performantly, monolithic orchestration software often necessitates separate threads for each component hardware interface, a fragile pattern in which small mistakes become both critical and elusive errors. When writing orchestration software using *yaq*, slow hardware interfaces are replaced by fast Avro-RPC. In our experience, the *yaq* interface is responsive enough to give good performance when orchestrating tens of motors using a singly threaded client. There is a limit where using TCP becomes a bottleneck for large responses, such as those from cameras or other high-throughput sensors. In our experience, this limit is ~1 megapixel. For larger sensors, a pure *yaq* approach may not be appropriate. There is a longer discussion of timing and order-of-operations control details using *yaq* in the [supplementary material](#).

III. EXPERIMENTAL FLEXIBILITY

Existing experimental orchestration software is often highly inflexible. An experimentalist will spend many hours in the lab manually repeating acquisitions because it is too challenging to add repetition functionality to their software. A laser lab needs to spend weeks on software development when introducing a single new step into its experimental procedure. Researchers are disappointed to realize that they are forced to start from scratch when developing software for a similar instrument built with trivially different hardware.

Unique experiments will always need custom orchestration and user experience. We believe that novel instrumentation development naturally and necessarily includes the creation of targeted software. Developing experimental software is an iterative process tied to the scientific goals of the instrument. Often experimentalists must apply their specialty scientific knowledge to develop this software.¹⁹

In our view, software inflexibility is a natural consequence of the typical software development practices used by custom instrument builders. Instrumental software is often built as one monolithic program that does everything from providing a graphical interface, through hardware interfacing, and writing data files. Such software is typically impossible to debug without access to real hardware, often requiring all of the hardware to be available to simply start the program. As such, instrumentation software development time

is in conflict with valuable data acquisition time. The hardware interfaces that these programs implement are typically made quickly and without regard to standardization with similar hardware. The orchestration routines are intimately tied to the particular hardware configuration of one instrument.

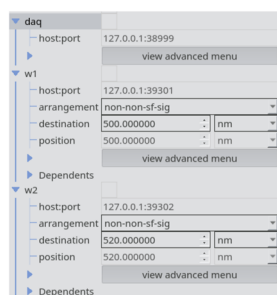
yaq is architected to encourage better software development practices when creating such programs. In a yaq context, orchestration code and graphical control interfaces are implemented as clients. These clients are automatically simpler because they only need to implement the yaq standard and do not need to include the vast array of hardware-specific communication interfaces. Beyond this, clients can use traits to interact with similar hardware identically. A client written to perform a two-dimensional fluorescence experiment using two Acton monochromators will also work with Horiba monochromators without any modification.

Figure 2 shows that yaq supports a diverse array of client types. At the top, we represent the most lightweight interface to yaq, Python scripting. yaqc²⁰ is a Python client that is excellent for using in scripts or any other Python program. The code shown creates three client objects that could be used directly through an interactive Python prompt or in a reusable script. Each client object provides Python methods for each Avro message specified by the associated protocol.

Scripts

```
import yaqc
w1 = yaqc.Client(39301)
w2 = yaqc.Client(39302)
daq = yaqc.Client(38999)
```

GUIs



Frameworks

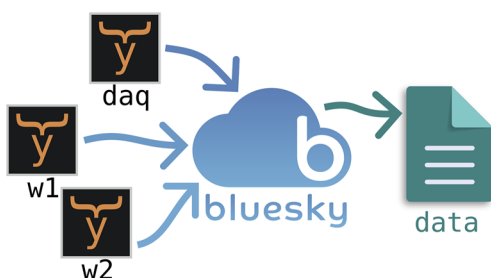


FIG. 2. Representations of three very different yaq client experiences. Top: lightweight scripting. Middle: graphical user interface. Bottom: integration with Bluesky for data collection. In all three cases, the same daemons are addressed.

In the middle, the same three hardware peripherals are represented in an interactive, graphical form. yaqc-qt²¹ is a graphical application that builds interactive controls based on traits for any conceivable yaq protocol. The self-describing yaq interface is used to provide graphical elements for the most commonly used messages. yaqc-qt²¹ is an invaluable tool that provides a “free” graphical user interface (GUI) to any daemon.

At the bottom, we graphically represent the integration between yaq and the Bluesky project.²² Bluesky provides a powerful orchestration layer for conducting and recording data for a wide variety of experimental procedures. Bluesky has no built-in hardware interface support, instead relying on packages to create hardware interfaces that are compatible with their Hardware Protocol definitions. yaqc-bluesky²³ is a specialized client that adapts any yaq protocol into Bluesky. Similar translation layers could be built for a variety of orchestration software such as PyMoDAQ,⁸ PyMeasure,¹³ or TRSpectrometer.¹⁰

All three types of clients represented in Fig. 2 have been shown addressing the same three hardware peripherals. All types of clients can be used simultaneously to interact with the same instrument in different modes. Client sophistication can be introduced naturally, as novel experiments are tested and refined. Different clients can specialize in different requirements of a custom instrument. For example, yaqc-qt²¹ can provide a quick interface for setting and viewing hardware positions, whereas Bluesky can focus on experimental data acquisition.

The Landis Group at UW-Madison is currently working on a new type of flow reactor: the Wisconsin Quench Kinetics Reactor (WiQK). This reactor incorporates several computer-controlled valves and syringe pumps as well as various sensors. The set of hardware peripherals is rapidly changing as researchers continue to test and refine their design. Only a few researchers are actively using the reactor during this prototyping stage. These researchers are experimentalists who have limited background in software development. The Landis Group has written basic Python scripts to orchestrate hardware for their reactor. These lightweight scripts can be extensively refactored by the experimentalists as the hardware and orchestration strategy changes dramatically during WiQK development. This approach ensures that the Landis Group is not slowed down by complex, inflexible orchestration software. Once the reactor is complete, more sophisticated graphical clients will be created to accommodate end users who were not involved as the reactor was built.

The Wright Group at UW-Madison needs to orchestrate a large variety of hardware in multidimensional scans for their complex spectroscopy experiments.^{24,25} This need for exquisite hardware control has resulted in several prior attempts at “home-built” orchestration software.^{26–30} Now, using yaq, the Wright Group has been able to move to Bluesky rather than inventing their own sophisticated control software “from scratch.” The Wright Group uses simulated hardware to enable client development away from active laboratory computers. Hardware simulation allows Wright Group researchers to create polished client interfaces without interrupting ongoing experiments. Clients developed by the Wright Group have proved flexible enough to be used on four laser systems, each with different complements of hardware. Moving forward, the Wright Group will spend less energy developing control software and more energy developing creative spectroscopy experiments.

IV. INCORPORATING NEW HARDWARE

In a *yaq* context, new hardware can be incorporated into an instrument through the addition of a new daemon. The *yaq* architecture simplifies hardware interface development in several ways. First, because daemons are separate and portable programs, the development effort can be spread across the community of *yaq* users. Often, researchers can download an existing daemon rather than writing a new one. Second, *yaq* daemon development can be performed separately from the particulars of any individual client. Often, this separation allows initial hardware enablement work to be done on a researcher's personal machine before the new hardware peripheral is installed in the instrument. Separability, portability, and distributed development are advantages common to many open-source hardware interface projects. Third, when developing trait-compliant protocols, it becomes easy to design and fully test your hardware interface. Traits are unambiguous and well described, making an obvious target for development. Tooling exists to verify full trait compliance, for example, you can use *yaqc-qtpy* to provide a graphical program to interact with your hardware immediately. Finally, as discussed in Sec. II, *yaq* provides options to design using remote hardware or unusual interfaces when necessary.

We have created several tools to aid in daemon development. First, a Python library, *yaqd-core*,³¹ which implements shared functionality. Second, *yaq-traits*³² is a command line application that allows the description of messages provided by a *yaq* protocol to be written in a human-readable fashion and translated into a more fully described machine-readable format. The format it generates is an important part of how *yaq* protocols are self-describing. This shields developers from the details of Apache Avro, which can be somewhat esoteric.

Figure 3 shows the distribution of unique lines of Python code written to implement each of the daemons in our current ecosystem. Although lines of code are an imperfect metric, we use it here to represent the amount of work required to create a daemon for

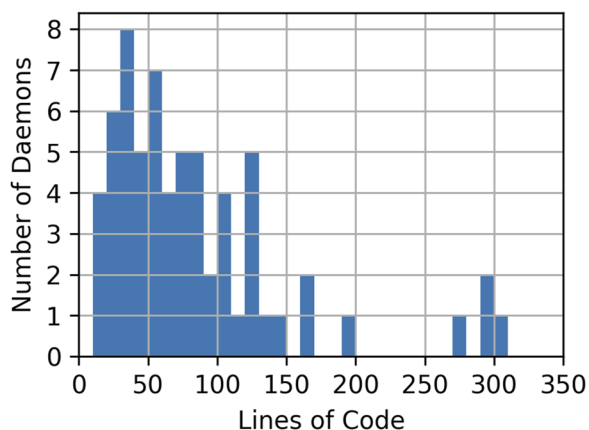


FIG. 3. Histogram of the number of lines for each implemented daemon. Some daemons are implemented in ways that share code, resulting in apparent line counts of less than 10. For example, the Thorlabs APT⁵ motor implementation supports at least eight different daemons with each specifying only a handful of constants. These are extreme examples that are not representative of most hardware interfaces; hence, we omit them here.

a new hardware peripheral. Most interfaces, such as Brooks MFC,³³ have been implemented in fewer than 100 lines of Python code. Even the most complicated daemons are implemented in about 300 lines. Implementing *yaq* daemons using Python is enabled through our own tooling mentioned above and the large and growing ecosystem of hardware interface libraries that Python now provides.^{34–37} In our experience, the process of creating a new daemon involves about a day of work after mastering communication with the hardware. There are currently 72 daemons in the *yaq* project supporting at least 66 types of hardware, noting that some daemons support the same hardware and others are software only. Because *yaq* is standards-based, anyone can design and publish new daemons extending our hardware support. A living list of all daemons and supported hardware can be found on the *yaq* website.³⁸

The Stahl Group at UW-Madison created a custom reactor that monitors gasses being produced or consumed in the reaction head space.³⁹ This reactor incorporates a collection of sensitive pressure transducers and a single-heating process value under computer control. *yaq* daemons are used to interface with each sensor and the heater controller. Recently, experimentalists have been attempting reactions involving smaller, slower, pressure changes. A fundamental flaw in the initial analog-to-digital converter board was revealed by these attempts. As a result, a new digitizer has been purchased. This new digitizer will be incorporated into the existing reactor without modifying the existing graphical user interface and data recording program, minimizing downtime.

V. TECHNICAL DEBT

Years after the original researchers leave, large monolithic acquisition programs become unknowable, undocumented, and unmaintained. A graduate student discovers a hard-coded conversion factor that is incorrect years after implementation. Scientists resort to sourcing an exact replacement for an old, broken oscilloscope due to their software's reliance on that particular interface—newer, cheaper oscilloscopes are readily available. A graduate student is forced to meticulously reverse engineer the LabVIEW codebase that they inherited to understand the details of their experiment. Software developers refer to the extra effort required to modify or fix large unmaintained codebases as “technical debt.”⁴⁰ Technical debt grows especially fast in academic environments where graduate students are involved in projects for a limited time.

The *yaq* approach favors many small single-purpose applications above the large monolithic ones. For daemons, the purpose of each application is obvious and unambiguous. There is a strict, well-defined interface that explicitly limits the kinds of interactions that are provided to the hardware, thus limiting the opportunity for unintended consequences. The lack of hardware interface code makes *yaq* clients much simpler and easier to describe and maintain. Tools like *yaqd-fakes*⁴¹ allow clients to be tested and improved outside of their instrument, including the possibility of fully automated testing. Simple, script-based clients written using the expressiveness of Python can be read and understood in hours rather than weeks. Integrations with communities like Bluesky offer powerful features that are actively maintained across many institutions.

In *yaq*, each component of an instrument can be developed and distributed separately. For example, two different instruments might

happen to use the same temperature sensor. Because the temperature sensor daemon is its own independent program, both instruments can benefit from the same daemon. The growing “ecosystem” of *yaq* daemons makes future instruments easier and easier to develop. Growing this ecosystem is a collaborative effort where many *yaq* users create portable daemons that they need and share them with the community where they will be used and improved. *yaq* components can be incorporated seamlessly into existing software projects, including languages other than Python. For example, an existing LabVIEW project may provide additional hardware support via *yaq*. In this way, a more modular instrument can be built gradually without needing to overhaul all of the software simultaneously.

Software built by scientists often has incomplete and inadequate documentation.⁴² *yaq* attempts to automate daemon documentation as much as possible. Our website, <https://yaq.fyi>, automatically builds generated reference pages for all known protocols. These pages are automatically updated when new versions are published. Our website also contains written and video tutorials on *yaq* usage and development.

We have designed *yaq* to be easier to deploy and maintain when compared with EPICS and TANGO. *yaq* daemons are Python packages that can be installed on any platform using pip or conda. There is no need for centralized management servers or databases when using *yaq*. The Avro RPC standard unambiguously describes the message signatures for any protocol; hence, users are always aware of the capabilities their hardware supports. Traits enable interchangeability where possible. *yaq* can easily be used alongside other experiment control software. We provide tooling (*yaqd-control*) that makes it easy to configure, list, and manage daemons as background services on Windows, MacOS, or Linux (see [supplementary material](#) for more details).

yaq is open-source software. Anyone can view, install, edit, and suggest changes to our growing collection of daemons and clients. Furthermore, anyone can create their own totally custom client or daemon software separately by following the specified *yaq* standard. Thirteen individuals have contributed code to the development of *yaq*. Open-source development can be a powerful approach for research communities looking to share software development and maintenance burdens.⁴³ It is our hope that a vibrant open-source community will form around *yaq*. Although open-source development is not a panacea,⁴⁴ we hope that by maintaining a distributed development strategy with a strict focus on only hardware interfaces, the *yaq* project might prove sustainable.

VI. CONCLUSION

The *yaq* project defines a new general-purpose standard for hardware control in the context of scientific instrumentation. This standard has some of the powerful features of facility-scale standards while remaining simple enough for feasible implementation and maintenance in small research labs. We have shown how this approach alleviates common problems through discussion and case studies. Designing around self-describing protocols is a productive approach that has great promise in scientific software development.

SUPPLEMENTARY MATERIAL

See [supplementary material](#) for full set of line-count data and script used to produce Fig. 3. More detailed examples of

orchestration using *yaq*, including scripts and graphical user interfaces. Detailed description of *yaqd-control*. Discussion and quantitative analysis of the *yaq* interface's performance with large arrays.

See [supplementary material](#) for a full description of the name “*yaq*.”

ACKNOWLEDGMENTS

The authors would like to thank all *yaq* users and contributors. We would also like to acknowledge the developers of the broader open-source software community on which this project rests.

This work was supported by the National Science Foundation under Grant No. CHE-1709060.

AUTHOR DECLARATIONS

Conflict of Interest

The authors have no conflicts to disclose.

Author Contributions

Kyle F. Sunden: Conceptualization (equal); Data curation (lead); Software (lead); Writing – original draft (equal); Writing – review & editing (equal). **Daniel D. Kohler:** Conceptualization (equal); Software (supporting); Writing – original draft (supporting); Writing – review & editing (supporting). **Kent A. Meyer:** Conceptualization (equal); Software (supporting); Writing – original draft (supporting); Writing – review & editing (supporting). **Peter L. Cruz Parrilla:** Conceptualization (equal); Software (supporting); Writing – original draft (supporting); Writing – review & editing (supporting). **John C. Wright:** Conceptualization (equal); Funding acquisition (lead); Supervision (supporting); Writing – original draft (supporting); Writing – review & editing (supporting). **Blaise J. Thompson:** Conceptualization (equal); Software (equal); Writing – original draft (equal); Writing – review & editing (equal).

DATA AVAILABILITY

The data that support the findings of this study are available within the article and its [supplementary material](#).

REFERENCES

- ¹ See <https://www.ni.com/en-us/support/documentation/supplemental/06/getting-started-with-ni-daqmx-main-page.html> for Getting started with NI-DAQmx; accessed: 02 October 2022, 2022.
- ² Standard Commands for Programmable Instruments (SCPI), SCPI Consortium, 1999.
- ³ MODBUS Application Protocol Specification, Modbus, 2012.
- ⁴ PICam 5.x Programmer's Manual, Teledyne Princeton Instruments, 2021.
- ⁵ Thorlabs Motion Controllers Host-Controller Communications Protocol, Thorlabs, 2022.
- ⁶ L. R. Dalesio, M. R. Kraimer, and A. J. Kozubal, “EPICS architecture,” in ICALEPCS (1991), Vol. 91.
- ⁷ J.-M. Chaize, A. Götz, J. Meyer, M. Pérez, and E. Taurel, “TANGO—An object oriented control system based on CORBA,” in ICALEPCS, 1999.
- ⁸ S. J. Weber, “PyMoDAQ: An open-source python-based software for modular data acquisition,” *Rev. Sci. Instrum.* **92**, 045104 (2021).

- ⁹N. Bogdanowicz, C. Rogers, S. Weber, Zakv, S. Pelissier, J. Wheeler, Cxz, F. Marazzi, Eedm, and I. Galinskiy, (2022) "Instrumental: A python-based library for controlling lab hardware," Zenodo. <https://doi.org/10.5281/zenodo.6591764>
- ¹⁰P. Tapping, "TRspectrumeter documentation," <https://trspectrumeter.readthedocs.io/>; accessed 02 October 2022, 2021.
- ¹¹L. Koerner, "Instrbuilder: A Python package for electrical instrument control," *J. Open Source Softw.* **4**, 1172 (2019).
- ¹²L. Campagnola, M. B. Kratz, and P. B. Manis, "ACQ4: An open-source software platform for data acquisition and analysis in neurophysiology research," *Front. Neuroinform.* **8**, 3 (2014).
- ¹³See <https://pymasure.readthedocs.io/> for PyMeasure; accessed: 02 October 2022, 2022.
- ¹⁴G. Giesbrecht, A. Amsellem, T. Bauer, B. Mak, B. Wynne, Z. Qin, and A. Persaud, "Hardware-control: Instrument control and automation package," *J. Open Source Softw.* **7**, 2688 (2022).
- ¹⁵A. Shkarin (2022). "pyLabLib," Zenodo. <https://doi.org/10.5281/zenodo.7324876>
- ¹⁶The yaq protocol is fully specified by the set of accepted yaq Enhancement Proposals at <https://yeps.yaq.fyi/>.
- ¹⁷E. S. Raymond, *The New Hacker's Dictionary*, 3rd ed. (MIT Press, Cambridge, MA, 1996).
- ¹⁸See <https://avro.apache.org/docs/1.11.1/specification/> for Apache avro specification; accessed: 02 October 2022, 2022.
- ¹⁹J. Segal, "When software engineers met research scientists: A case study," *Empir. Softw. Eng.* **10**, 517–536 (2005).
- ²⁰yaqc available on PyPI at <https://pypi.org/project/yaqc/>.
- ²¹yaqc-qtpy available on PyPI at <https://pypi.org/project/yaqc-qtpy/>.
- ²²D. Allan, T. Caswell, S. Campbell, and M. Rakitin, "Bluesky's ahead: A multi-facility collaboration for an a la carte software project for data acquisition and management," *Synchrotron Radiat. News* **32**, 19–22 (2019).
- ²³yaqc-bluesky available on PyPI at <https://pypi.org/project/yaqc-bluesky/>.
- ²⁴S. Mukamel, "Multidimensional femtosecond correlation spectroscopies of electronic and vibrational excitations," *Annu. Rev. Phys. Chem.* **51**, 691–729 (2000).
- ²⁵J. C. Wright, "Multiresonant coherent multidimensional spectroscopy," *Annu. Rev. Phys. Chem.* **62**, 209–230 (2011).
- ²⁶R. J. Carlson, "Quantitative aspects of high resolution, fully resonant, four-wave mixing spectroscopy for the analysis of vibronic mode coupling in molecules," Ph.D. thesis, University of Wisconsin-Madison, 1988.
- ²⁷K. A. Meyer, "Frequency-scanned ultrafast spectroscopic techniques applied to infrared four-wave mixing spectroscopy," Ph.D. thesis, University of Wisconsin-Madison, 2004.
- ²⁸S. Kain, "Transition of frequency-domain coherent multidimensional spectroscopic methods to the femtosecond time regime with applications to nanoscale semiconductors," Ph.D. thesis, University of Wisconsin-Madison, 2017.
- ²⁹B. J. Thompson, "Development of frequency domain multidimensional spectroscopy with applications in semiconductor photophysics," Ph.D. thesis, University of Wisconsin-Madison, 2018.
- ³⁰K. F. Sunden, "yaq: Yet Another Acquisition a modular approach to spectroscopy software and instrumentation," Ph.D. thesis, University of Wisconsin-Madison, 2022.
- ³¹yaqd-core-python available on PyPI at <https://pypi.org/project/yaqd-core>.
- ³²yaq-traits available on PyPI at <https://pypi.org/project/yaq-traits>.
- ³³yaqd-brooks-mfc-gf source code available on GitHub at <https://github.com/yaq-project/yaqd-brooks/>.
- ³⁴See <https://pypi.org/project/pyserial> for PySerial; accessed: 02 October 2022.
- ³⁵See <https://pypi.org/project/pyusb> for PyUSB; accessed: 06 November 2022.
- ³⁶See <https://pypi.org/project/pyvisa> for PyVISA; accessed: 06 November 2022.
- ³⁷See <https://pypi.org/project/pymodbus> for PyModbus; accessed: 06 November 2022.
- ³⁸See <https://yaq.fyi/daemons/> and <https://yaq.fyi/hardware/> for living lists of all yaq daemons and supported hardware, respectively.
- ³⁹C. A. Salazar, B. J. Thompson, S. M. M. Knapp, S. R. Myers, and S. S. Stahl, "Multichannel gas-uptake/evolution reactor for monitoring liquid-phase chemical reactions," *Rev. Sci. Instrum.* **92**, 044103 (2021).
- ⁴⁰E. Allman, "Managing technical debt," *Commun. ACM* **55**, 50–55 (2012).
- ⁴¹yaqd-fakes available on PyPI at <https://pypi.org/project/yaqd-fakes/>.
- ⁴²J. Segal, "Some problems of professional end user developers," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)* (IEEE, 2007).
- ⁴³J. Cohen, D. S. Katz, M. Barker, N. Chue Hong, R. Haines, and C. Jay, "The four pillars of research software engineering," *IEEE Softw.* **38**, 97–105 (2021).
- ⁴⁴A. Nowogrodzki, "How to support open-source software and stay sane," *Nature* **571**, 133–134 (2019).

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	Orchestration Examples	S2
1.1	busy for ensuring order of operations	S2
1.2	Raster Script	S3
1.3	Non-Blocking Script	S4
1.4	Landis	S5
1.5	Wright & Bluesky	S5
2	yaq transport layer	S9
2.1	History of yaq protocol format	S9
2.2	Scaling of large messages	S9
3	yaqd-control: tooling for managing yaq daemons	S12
3.1	installation	S12
3.2	Usage	S12
4	Package size analysis	S16
5	The name yaq	S17

1 Orchestration Examples

Here we elaborate with detailed information about what orchestration looks like for yaq. As we discussed in Section 3 of the manuscript, yaq is designed to accommodate a wide variety of orchestration tools and approaches.

1.1 busy for ensuring order of operations

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

In order to know how long to wait, all yaq 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 simultaneously. A complex instrument may involve multiple operators watching sensor data in real time, while one program is orchestrating the hardware and recording the data.

Script writers may simply wait for “busy” to return “false”.

```
import yaqc
import time
motor = yaqc.Client(port=38000)
motor.set_position(500) # initiate a long motor motion
while motor.busy():
    time.sleep(0.001)
print("done") # will only print when motion is complete
```

This same pattern exists for sensors and more complex peripherals.

We believe that Python and the Scientific Python ecosystem is a powerfully simple tool for defining experimental procedures. Lightweight scripts such as these are encouraged by the yaq project. The following two sections contain example orchestration scripts for common tasks.

1.2 Raster Script

For those unfamiliar with yaqc, the best orchestration example is a simple self-contained script.

```
import time
import yaqc
motor1 = yaqc.Client(port=38000)
motor2 = yaqc.Client(port=38001)
sensor = yaqc.Client(port=38002)
data = []
for m1 in range(-10, 10, 1):
    motor1.set_position(m1)
    for m2 in range(0, 300, 5):
        motor2.set_position(m2)
        for c in [motor1, motor2]:
            while c.busy():
                time.sleep(0.001)
        sensor.measure()
        while sensor.busy():
            time.sleep(0.001)
        reading = dict()
        reading["timestamp"] = time.time()
        reading["motor1"] = motor1.get_position()
        reading["motor2"] = motor2.get_position()
        reading.update(sensor.get_measured())
        data.append(reading)
```

Here we are doing a two dimensional motor scan while collecting data from a sensor. Motor one is stepping from -10 to 10, and motor two is scanning from 0 to 300. Two examples of polling while not busy are used: the first to ensure that motors have stopped moving before sensors acquire, and the second to ensure that sensors stop collecting before the next motor motion happens. This “tick tock” moving multiple motors, collecting sensor data, and repeating is a common pattern.

1.3 Non-Blocking Script

Sometimes you want to acquire data while a peripheral continuously moves, asynchronously with data acquisition. The following is an example where sample temperature is continuously ramped and data is collected every second.

```
import time
import yaqc
temp = yaqc.Client(port=38003)
sensor = yaqc.Client(port=38004)
data = []
# begin heating
temp.set_position(100)
# take data until temp greater than 90 deg
while temp.get_position() <= 90:
    reading = dict()
    reading["timestamp"] = time.time()
    reading["temperature"] = temp.get_position()
    # measure
    sensor.measure()
    while sensor.busy():
        time.sleep(0.001)
    reading.update(sensor.get_measured())
    data.append(reading)
    # wait for next second
    while time.time() - reading["timestamp"] < 1:
        time.sleep(0.001)
# stop heating
temp.set_position(25)
```

Imagine that the temperature starts out at room temperature, 25. We first set the temp client to 100, beginning to heat the sample. The outer while loop will continue until the temperature reaches our desired end-point of 90. Each second, the while loop collects the current temperature and the current sensor reading. This type of experiment is common in the chemical sciences.

Note that we pay no attention to the busy state of the temp client. When compared with the previous raster scanning example, we see that yaq can be flexible about which hardware operations are kept synchronous and which are allowed to run “freely” without blocking other operations. The ubiquity and simplicity of the busy message allows for this flexibility.

1.4 Landis

We discussed the Landis Group and their WiQK reactor in Section 3 of the manuscript. The Landis Group has defined several procedures for their flow reactor. These procedures are highly idiosyncratic and are based on the particular tubing lengths and the flow topology of the reactor. Each procedure involves articulating solenoid valves while injecting and withdrawing syringe pumps with a particular timing. The Landis Group has written Python functions that drive this hardware using `yaq`. The functions parameterize the procedures in terms of chemically interesting variables like flow rate and reaction time.

WiQK procedures and nascent graphical user interface can be found on GitHub at <https://github.com/uw-madison-chem-shops/yaqc-wiqk>.

1.5 Wright & Bluesky

The Wright Group uses a purpose-built graphical program to perform data acquisition. All of the hardware used is controlled via `yaq`. Data is collected using Bluesky [1], which runs in a background process.

The graphical program includes a queue of acquisition procedures (plans, in Bluesky terminology). In the main body of the queue window (Figure S1), a table represents the current state of the queue and history. More recently enqueued items appear towards the top of the table, older items towards the bottom. The items waiting in queue, thus, appear at the top, and the items in the history appear at the bottom. The currently running item (if it exists) appears at the boundary between the queue and the history. The enqueued and running items have descriptions shown in bright white, while the completed items appear with gray descriptions. The right most column indicates the position in the queue, given as a zero-based index. Items can be reordered by editing the value in that column. On left, there are two columns with buttons: remove and load. Items in the queue (but not the history) can be removed individually with the red “REMOVE” buttons. All items can be loaded into the sidebar where the parameters can be modified and a similar (or identical) item enqueued. The other information provided include the plan name, the status (enqueued, RUNNING, completed, failed, aborted, stopped or unknown) and the description field, which lists the arguments passed to the plan. Hovering over the description shows the full JSON of the item, including an error message if applicable.

Along the left side of the window, there is a panel that allows enqueueing new items. Figure S2 shows the configuration panel for the most popular acquisition procedure, `grid_scan`. This plan allows an arbitrary number of dimensions to be scanned against one another, selecting the hardware to be scanned and the detectors to read.

The plot tab (Figure S3) presents a live representation of recent data collected for the current plan. The primary window is a PyQtGraph [2] plot showing the most recent five slices of collected data. The current slice is the brightest cyan slice, with the previous slices shown as duller colors progressively until the last one which is gray. At the top, the currently plotted channel name and most recent collected value are shown in large font size. The plotting does handle array detectors, for which the top number shown is the maximum value of the most recent array collected. At present, cameras and other higher dimensional sensors are not able to be plotted using the live plot tab. On the right hand side, there are controls to adjust the plot. At the top, a selector allows for choosing which channel to plot. The second selector determines which values appear along the X-axis of the graph. Additionally, there is a selector which allows for changing the units of the X-axis. Finally, there is a Scan Index indicator, which indicates which indexed pixel was the most recent collected.

00:01:51

bluesky-cmds 2022.7.0+fix_issues_39_40

00:05:55

Queue

Plot

Logs

Control Queue

INTERRUPT

CLEAR QUEUE

CLEAR HISTORY

Add to Queue

Type

plan

Plan

count

Metadata

Name

Info

Experimentor

Devices

daq

wa

Npts

Delay

APPEND TO QUEUE

Index	Type	Status	Description	
2	run_intensity	enqueued	[[daq], 'w1', 'delay_2']{width: 3.0, npts: 31, 'spectrometer': {device: 'wmi', 'method': 'static', 'center': 200.0}}	REMOVE
1	run_intensity	enqueued	[[daq], 'w1', 'delay_2']{width: 3.0, npts: 31, 'spectrometer': {device: 'wmi', 'method': 'track'}}	REMOVE
0	run_intensity	enqueued	[[daq], 'w1', 'delay_2']{width: 3.0, npts: 31, 'spectrometer': {device: 'wmi', 'method': 'scan', 'width': -350.0, 'npts': 33}}	REMOVE
	run_intensity	RUNNING	[[daq], 'w1', 'delay_2']{width: 4.0, npts: 41, 'spectrometer': {device: 'wmi', 'method': 'track'}}	
	run_intensity	completed	[[daq], 'w1', 'crystal_2']{width: 3.0, npts: 31, 'spectrometer': {device: 'wmi', 'method': 'zero'}}	
	run_intensity	completed	[[wa], 'w1', 'crystal_1']{width: 1, npts: 11, 'spectrometer': None}	
	run_intensity	completed	[[wa], 'w1', 'crystal_1']{width: 1, npts: 11, 'spectrometer': None}	
	grid_scan_wp	completed	[[daq], 'd0', 0, -1.0, 61, 'ps', 'd2', 0, -1.0, 51, 'ps', 'd1', 0, -1.0, 25, 'ps']{constants: []}	
	grid_scan_wp	completed	[[daq], 'd0', 0, -1.0, 61, 'ps', 'd2', 0, -1.0, 51, 'ps', 'd1', 0, -1.0, 25, 'ps']{constants: []}	
	grid_scan_wp	completed	[[daq], 'wa', 'd0', 0, -1.0, 11, 'ps', 'd2', 0, -1.0, 11, 'ps']{constants: []}	
	grid_scan_wp	failed	[[daq], 'd0', 0, -1.0, 61, 'ps', 'd2', 0, -1.0, 51, 'ps', 'd1', 0, -1.0, 25, 'ps']{constants: []}	
	grid_scan_wp	completed	[[daq], 'wa', 'd0', 0, -1.0, 11, 'ps', 'd2', 0, -1.0, 11, 'ps']{constants: []}	
	grid_scan_wp	unknown	[[daq], 'd0', 0, -1.0, 61, 'ps', 'd2', 0, -1.0, 51, 'ps', 'd1', 0, -1.0, 25, 'ps']{constants: []}	
	grid_scan_wp	stopped	[[daq], 'wa', 'd0', 0, -1.0, 61, 'ps', 'd2', 0, -1.0, 51, 'ps', 'd1', 0, -1.0, 25, 'ps']{constants: []}	
	grid_scan_wp	completed	[[daq], 'wa', 'd0', 0, -1.0, 11, 'ps', 'd2', 0, -1.0, 11, 'ps']{constants: []}	
	grid_scan_wp	failed	[[daq], 'wa', 'd1', 0, 1, 11, 'ps', 'd0', 0, 1, 0, 'ps']{constants: []}	
	grid_scan_wp	failed	[[daq], 'wa', 'd0', 0, 1, 11, 'ps', 'd0', 0, 1, 11, 'ps']{constants: [[d0', 'ps', [[1.0, d1]]]]}	
	count	completed	[[daq], 'wa']{num: 8, 'delay': 0}	
	count	completed	[[daq], 'wa']{num: 3, 'delay': 0}	

Figure S1: The main queue window of bluesky-cmds.

Add to Queue

Type

plan

Plan

grid_scan_wp

Metadata

Name

Info

Experimentor

unspecified

Devices

daq

✓

wa

✓

Axes

Axis

Hardware

d1

Start

0.000

Stop

1.000

Npts

11

Units

ps

ADD

REMOVE

Constants

Constant

Hardware

d0

Units

ps

Expression

1*d1

ADD

REMOVE

APPEND TO QUEUE

Figure S2: The user interface for enqueueing a grid scan plan.

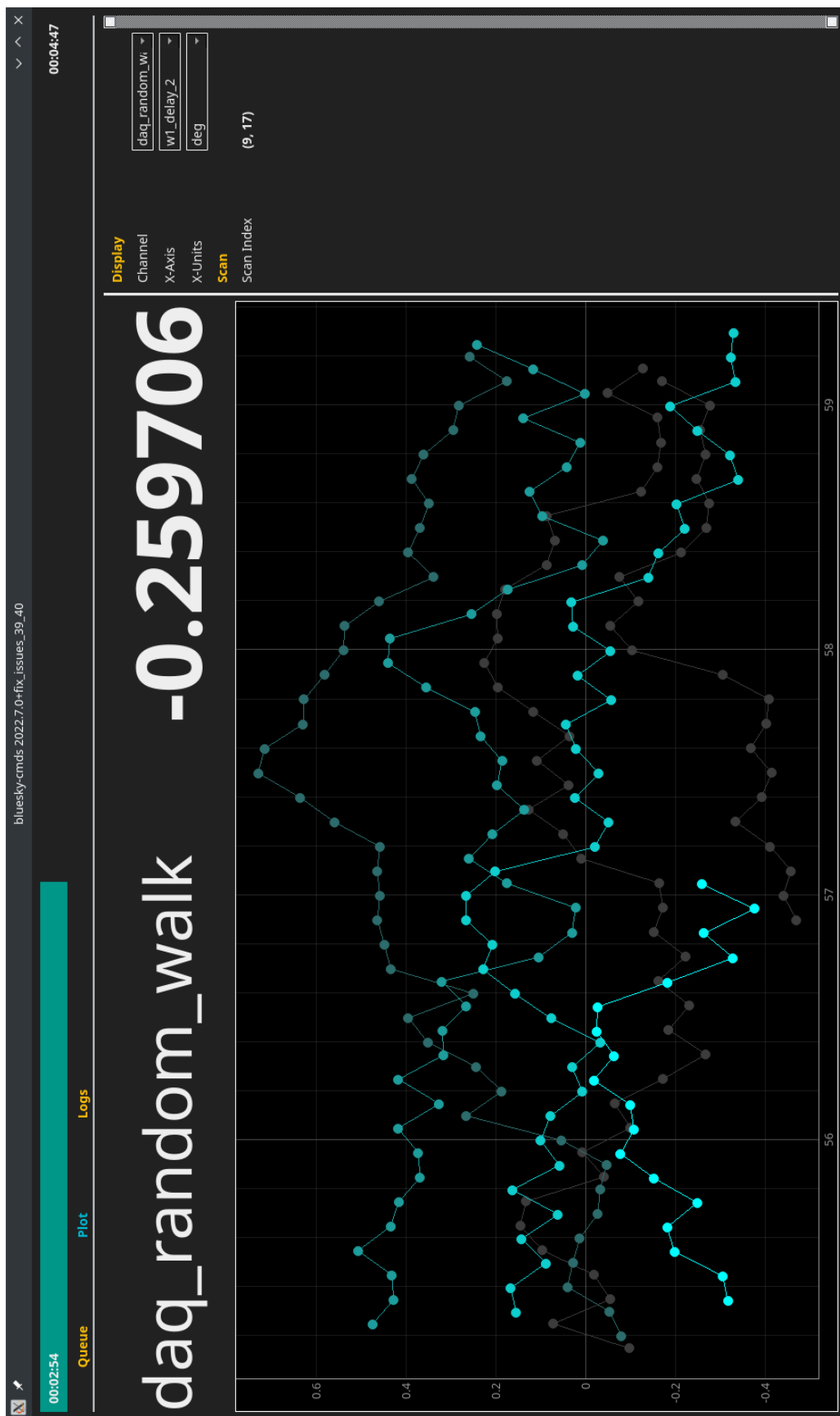


Figure S3: bluesky-cmds with the Plot tab open.

2 yaq transport layer

2.1 History of yaq protocol format

In the first days of yaq, the message format was entirely made up of JSON strings, forming a generic Remote Procedure Call (RPC) framework. There was no pre-formed protocol, though there were standardized messages to query for all allowed message names, signatures, and documentation.

Only after having a largely functional implementation, we discovered a community standard, JSON-RPC [3]. While the exact format was not identical, almost all of the ideas we had built into our home baked JSON-based RPC were easy to translate into this standard format. Instead of relying on our own esoteric format, we converted all yaq daemons and clients to use JSON-RPC.

As yaq gained hardware support and usage, it became evident that JSON was not an appropriate serialization format, especially when it comes to large binary data like images from cameras. As such, we transitioned to using an RPC based on JSON-RPC, but using Msgpack [4] for serialization. Msgpack is a format that can encode a strict superset of JSON messages, so swapping existing daemons required only changes to the core implementation and the client program (i.e. no changes to individual daemon implementations). Along with using Msgpack, we defined an extension type to represent NDArrays.

The last major change to yaq's protocol format was the adoption of Apache Avro-RPC [5]. Avro is similarly terse like Msgpack for serializing messages, while having a well defined specification for defining an RPC schema: a static definition of available messages, including their signatures and documentation. This statically defined schema pairs well with yaq's trait system. Daemon authors provide a short, minimal TOML file which describes the implemented traits and unique messages for a particular daemon. This then gets transformed by an automated tool to a full Avro-RPC JSON file describing the protocol. Avro is then used to encode individual messages with minimal overhead.

2.2 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 yaq 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 S4 shows the relationship between number of pixels and time for a yaq message to read the array. Below about 2^{18} ($\approx 250,000$) pixels, there is virtually no dependence on size, with the standard overhead of a yaq message dominating the time for transfer. yaq remains usable for up to approximately 2^{20} ($\approx 1,000,000$) 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 yaq even with relatively small cameras. This test was conducted using 32-bit integer arrays.

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 yaq 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.

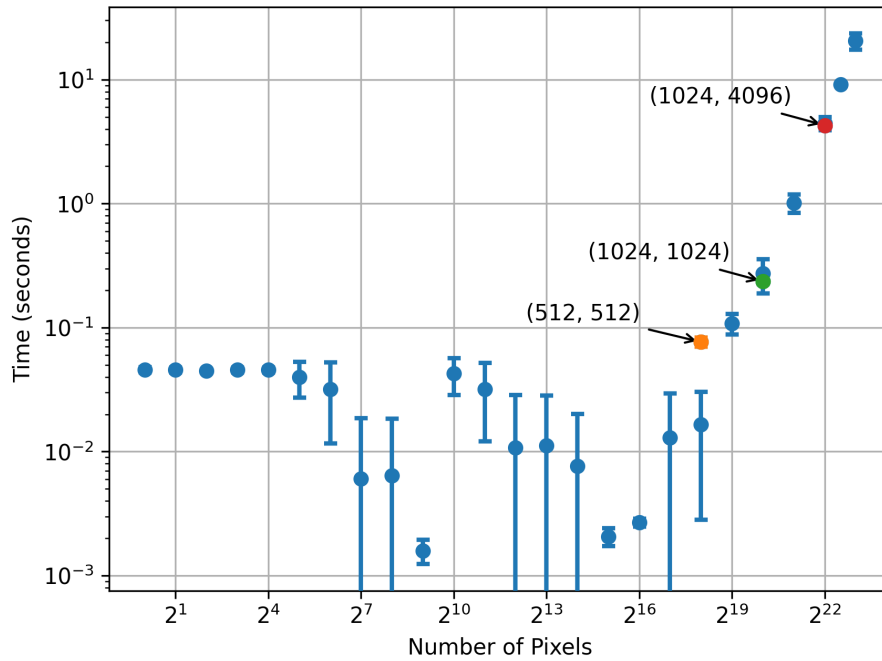


Figure S4: Scaling of transport time as a function of number of pixels for 32-bit integer arrays. Both the x- and y-axes are logarithmically scaled. Common camera sizes (512, 512), (1024, 1024), and (1024, 4096) are highlighted to provide context for camera users.

The Avro Specification [5] 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. EPICS provides a similar NDAarray data type, which has individual image level compression configurable [6]. Compression 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 yaq, 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 yaq ecosystem [7, 8], however we are still in the experimentation phase and have not standardized ROI configuration into a yaq trait. When only a subset of the total camera area is interesting, this is one way to limit array size over the interface.

Finally yaq daemons could be implemented to write arrays locally to disk, rather than transferring over the yaq interface directly. Anecdotally, many users of Bluesky, EPICS, and related technologies use a similar strategy for large image data. This has not yet been implemented for yaq, 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 yaq 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 yaq daemon, are located in “figures/scaling”. To generate the data, run the yaq daemon using `python scaling.py --config config.toml` from within 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.

3 yaqd-control: tooling for managing yaq daemons

yaqd-control is a command line utility provided by the yaq project to manage yaq daemons. The primary purpose of the tool is to maintain a list of active daemons, provide access to their configuration, and their running state. This includes querying the active state of known daemons, as well as starting daemons running as background services.

3.1 installation

yaqd-control can be installed via PyPI [9] or conda-forge [10].

```
$ pip install yaqd-control
```

```
$ conda config --add channels conda-forge
$ conda install yaqd-control
```

3.2 Usage

yaqd-control is a command line application.

Help: learn more, right from your terminal.

```
$ yaqd --help
Usage: yaqd [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  clear-cache
  disable
  edit-config
  enable
  list
  reload
  restart
  scan
  start
  status
  stop
```

Try `yaqd --help` to learn more about a particular command.

the cache yaqd-control keeps track of known daemons, referred to as the cache

Status: yaqd-control can quickly show you the status of all daemons in yaqd-control's cache. This is usually the most used subcommand, as it gives a quick overview of the system, which daemons are offline, and which are currently busy.

```
$ yaqd status
+-----+-----+-----+-----+-----+-----+
| host   | port  | kind               | name | status | busy |
+-----+-----+-----+-----+-----+-----+
| 127.0.0.1 | 38202 | system-monitor     | foo  | online | False |
| 127.0.0.1 | 39054 | fake-continuous-hardware | bar  | online | True  |
| 127.0.0.1 | 39055 | fake-continuous-hardware | baz  | online | False |
| 127.0.0.1 | 39056 | fake-continuous-hardware | spam | offline | ?    |
| 127.0.0.1 | 37067 | fake-discrete-hardware  | ham  | online | False |
| 127.0.0.1 | 37066 | fake-discrete-hardware  | eggs | online | False |
+-----+-----+-----+-----+-----+-----+
```

List: this is essentially the same as status except that it does not attempt to contact the daemons, so it does not give you additional context. List supports a flag `--format` which accepts "json", "toml", "prettytable", or "happi".

```
$ yaqd list
+-----+-----+-----+-----+
| host   | port  | kind               | name |
+-----+-----+-----+-----+
| 127.0.0.1 | 38202 | system-monitor     | foo  |
| 127.0.0.1 | 39054 | fake-continuous-hardware | bar  |
| 127.0.0.1 | 39055 | fake-continuous-hardware | baz  |
| 127.0.0.1 | 39056 | fake-continuous-hardware | spam |
| 127.0.0.1 | 37067 | fake-discrete-hardware  | ham  |
| 127.0.0.1 | 37066 | fake-discrete-hardware  | eggs |
+-----+-----+-----+-----+
```

Scan: Scanning allows you to add currently running daemons to the cache.

```
$ yaqd scan
scanning host 127.0.0.1 from 36000 to 39999...
...saw unchanged daemon fake-discrete-hardware:eggs on port 37066
...saw unchanged daemon fake-discrete-hardware:ham on port 37067
...found new daemon system-monitor:foo on port 38202
...found new daemon fake-continuous-hardware:bar on port 39054
...saw unchanged daemon fake-continuous-hardware:baz on port 39055
...known daemon fake-continuous-hardware:spam on port 39056 not responding
...done!
```

Scan has some additional options, passed as flags on the command line, which allow you to change the default scan range and host (for remotely accessed daemons):

```
$ yaqd scan --help
Usage: yaqd scan [OPTIONS]

Options:
  --host TEXT      Host to scan.
  --start INTEGER  Scan starting point.
  --stop INTEGER   Scan stopping point.
  --help           Show this message and exit.
```

Edit Config: `yaqd-control` provides an easy way to edit the default config file location for a daemon kind. This uses your default editor (`EDITOR` environment variable), and defaults to `notepad.exe` on Windows, and `vi` on other platforms. Using `yaqd-control` to edit config files means that you do not need to know the default location. Additionally, it does some basic validity checks (that the toml parses and that each daemon section has the `port` keyword). If an error is found, you are prompted to re-edit the file. Daemons from the config file are added to the cache. You may pass multiple daemon kinds, which will be opened in succession.

```
$ yaqd edit-config fake-continuous-hardware system-monitor
```

Clear Cache: Note that this is a destructive action. `clear-cache` deletes all daemons from the cache (thus `list` and `status` will give empty tables) There is no user feedback.

```
$ yaqd clear-cache
$ yaqd status
+-----+-----+-----+-----+-----+
| host | port | kind | name | status | busy |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
```

Running in the background Each of the commands in this section can take multiple daemon kinds. **Enable:** by enabling a daemon, you allow the operating system to manage that daemon in the background. An enabled daemon will always start again when you restart your computer. Enabling is required for the rest of the commands in this section to work as expected. After enabling, it's typical to start the daemon as well, this does not happen automatically. Enablement works in slightly different ways on different platforms, but the commands are the same (don't worry if the password prompts are different). Currently supported platforms are Linux (`systemd`), MacOS (`launchd`) and Windows (via `NSSM`, bundled with the distribution).

```
$ yaqd enable system-monitor
[sudo] password for scipy2020:
==== AUTHENTICATING FOR org.freedesktop.systemd1.manage-unit-files ====
Authentication is required to manage system service or unit files.
Password:
==== AUTHENTICATION COMPLETE ====
```

Disable: this is the inverse operation to enable, which makes it so that the daemon does not start on reboot. This does not affect the running daemon.

```
$ yaqd disable system-monitor
==== AUTHENTICATING FOR org.freedesktop.systemd1.manage-unit-files ====
Authentication is required to manage system service or unit files.
Password:
==== AUTHENTICATION COMPLETE ====
Removed /etc/systemd/system/multi-user.target.wants/yaqd-system-monitor.service.
```

Start: This starts the daemon running in the background immediately. It must have been enabled to run in the background using this command.

```
$ yaqd start system-monitor
==== AUTHENTICATING FOR org.freedesktop.systemd1.manage-units ====
Authentication is required to start 'yaqd-system-monitor.service'.
Password:
==== AUTHENTICATION COMPLETE ====
```

Stop: This stops the daemon running in the background immediately. It must have been running in the background using yaqd-control (either on startup via enable or via the start command above).

```
$ yaqd stop system-monitor
==== AUTHENTICATING FOR org.freedesktop.systemd1.manage-units ====
Authentication is required to stop 'yaqd-system-monitor.service'.
Password:
==== AUTHENTICATION COMPLETE ====
```

Restart/Reload: This stops (if running) and restarts the daemon running in the background immediately. Reload is slightly different in that it signals to the daemon to reload its configuration rather than completely restart, but effectively it is the same as restart (and is a pure alias where such a signal is not supported). It must have been enabled to run in the background using this command.

```
$ yaqd restart system-monitor
==== AUTHENTICATING FOR org.freedesktop.systemd1.manage-units ====
Authentication is required to restart 'yaqd-system-monitor.service'.
Password:
==== AUTHENTICATION COMPLETE ====
```


4 Package size analysis

The raw data for package size analysis was collected using Tokei [11], 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 yaq python implementations. This is a measurable proxy for the amount of work that implementing an individual yaq daemon entails, though of course cannot capture the work involved in learning the lower level interface.

5 The name yaq

The name yaq started out as an “inside joke” with several meanings. For posterity, we describe those meanings here.

1. yaq is similar to the popular acronym “daq” (Data AcQuisition)
2. yaq could stand for Yet Another acQuisition, a reference to the popular tongue-in-cheek naming convention “yet another X” in software development [12]. This is a particularly poignant reference for the authors, who are well aware that yaq is “yet another project” in a long line of increasingly sophisticated hardware interface software packages in the Wright Group.
3. yaq can reference the animal “yak”, *Bos grunniens*, who serves as a mascot for the project.
4. yaq is an obscure reference to the term “yak shaving”, coined by Carlin Vieri at MIT in the 1990s [13]. Quoting from Wiktionary [14]: yak shaving is *any apparently useless activity which, by allowing you to overcome intermediate difficulties, allows you to solve a larger problem*. The entire yaq project is one giant “yak shaving” task that ultimately enables progress towards our primary mission—creative, reproducible, insightful experimental science.

References

- [1] Daniel Allan, Thomas Caswell, Stuart Campbell, and Maksim Rakitin. “Bluesky’s Ahead: A Multi-Facility Collaboration for an a la Carte Software Project for Data Acquisition and Management”. In: *Synchrotron Radiation News* 32.3 (2019), pp. 19–22. DOI: [10.1080/08940886.2019.1608121](https://doi.org/10.1080/08940886.2019.1608121).
- [2] *PyQtGraph: Scientific Graphics and GUI Library for Python*. <http://pyqtgraph.org/>. Accessed: 2022-10-02.
- [3] *JSON-RPC 2.0 Specification*. <https://www.jsonrpc.org/specification>. Accessed: 2023-02-08. 2013.
- [4] *msgpack*. <https://github.com/msgpack/msgpack/blob/master/spec.md>. Accessed: 2023-02-08.
- [5] *Apache Avro Specification*. <https://avro.apache.org/docs/1.11.1/specification/>. Accessed: 2022-10-02. 2022.
- [6] *EPICS Normative Types Specification*. <https://docs.epics-controls.org/en/latest/specs/Normative-Types-Specification.html/>. Accessed: 2023-02-08. 2019.
- [7] *yaqd-andor*. <https://pypi.org/project/yaqd-andor>. Accessed: 2023-02-05.
- [8] *yaqd-pi*. <https://github.com/yaq-project/yaqd-pi>. Accessed: 2023-02-05.
- [9] *yaqd-control*. <https://pypi.org/project/yaqd-control>. Accessed: 2022-10-02.
- [10] *yaqd-control*. <https://anaconda.org/conda-forge/yaqd-control>. Accessed: 2022-10-02.
- [11] *Token*. <https://github.com/XAMPPRocky/token>. Accessed: 2023-02-05.
- [12] Eric S. Raymond. *The new hacker’s dictionary*. 3rd ed. Cambridge, Mass: MIT Press, 1996. ISBN: 9780262181785.
- [13] *Yak shaving*. <https://web.archive.org/web/20210112174206/http://projects.csail.mit.edu/gsb/old-archive/gsb-archive/gsb2000-02-11.html>. Accessed: 2023-03-16.
- [14] *yak shaving*. https://en.wiktionary.org/wiki/yak_shaving. Accessed: 2023-03-16.