The yaq Project:
A Standard for Scientific Instrumentation

Kyle F. Sunden,[1] Daniel D. Kohler,[1] Kent A. Meyer,[1] Peter L. Cruz Parilla,[1] John C. Wright,[1] and Blaise J. Thompson[1]

University of Wisconsin–Madison

(*Electronic mail: blaise.thompson@wisc.edu)

(Dated: 31 October 2022)

Modern instrumentation development often involves incorporation of many dissimilar hardware peripherals into a single unified instrument. Increasing hardware availability has brought greater instrument complexity to small research groups. The flexibility promised to instrument makers by large hardware availability stretches the capability of traditional monolithic orchestration software. In many cases, a lack of software flexibility leads creative researchers to feel frustrated and unable to perform the experiments they envision. Highly modular hardware components enable extreme experimental flexibility. Herein we describe yaq, a software project defining a new standard way of communicating with diverse hardware peripherals. yaq encourages a highly modular approach to experimental software development, which is well suited to address the experimental flexibility needs of such instruments. yaq is architected to overcome hardware communication barriers that are insurmountable with typical experimental software. A large number of hardware peripherals are already supported with tooling available to create more. The yaq standard enables collaboration between multiple research groups, increasing code quality while lowering development effort.

I. INTRODUCTION

Instrumentation development is a key part of the scientific enterprise. Novel instruments are typically constructed of many individual components, purchased and home-built. Orchestration software must communicate with each hardware component in the course of a scientific experiment. This can involve utilization of many interfaces: NI DAQmx[1], SCPI[2], ModBus[3], PICam[4], Thorlabs APT[5], among many others. The process of integrating these is an important but time-consuming step of modern instrument and systems development. 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 over time. Scientists may struggle to rapidly innovate on 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 EPICS[6] and/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 state. 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 exisiting 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]. While this growth is encouraging, many of these are limited by their focus on particular types of hardware or particular experimental domains. Most small custom research instrumentation continues to rely on monolithic software which has hard-coded interface support for each particular connected device. These monolithic applications tend to be inflexible and difficult to develop.

In order to address these issues, we have created a novel network-based standard for scientific instrumentation, called yaq. This standard borrows the most important ideas from established projects used by large user facilities while retaining simplicity appropriate for small research labs. We have built this standard to be self-describing, portable, and reusable wherever possible. Our primary goal has been to make an interface which simplifies orchestration software development as much as possible.

Here, we discuss the design of the yaq standard in the context of four 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. Next, 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. The reader may also refer to the yaq website for a formal specification of the yaq standard.[?]
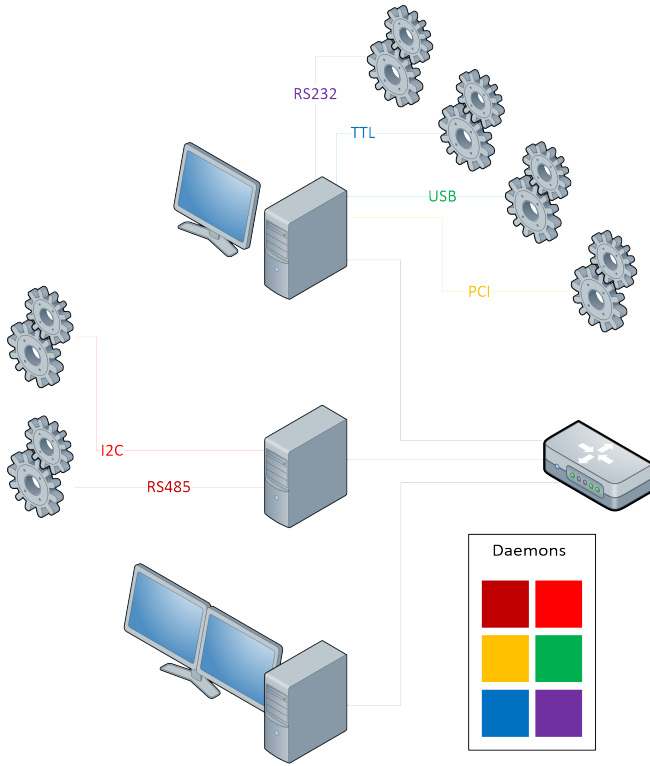
FIG. 1. Networking diagram

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: Coding interfaces adds to a maintenance overhead seldom manageable by researchers.* Updates to the hardware and operating systems often require maintenance updates to interfacing code that is not maintainable as researchers are not usually delegated to be strictly software maintainers. Third barrier: Certain specialty hardware have 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. Fourth 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 manufacturers configuration software can be used. A graduate student finds themself needing to master advanced concepts in concurrency in order to orchestrate many motors performantly.

Figure **??** diagrams the `yaq` architecture. Here, we show three different computers connected via an ethernet network.

The top and bottom computer are connected to monitors for interactive use while the middle computer is only accessible via the network. This 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 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"[16]. 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 Figure **??** 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 architecture can be used on a single computer or used across many networked computers, including fully remote operator interfaces. This client-server arrangement offers similar network capabilities to EPICS and TANGO. As we will show, usage of standards and the creation of tooling makes 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[17]. Avro provides an agreed upon 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 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". These self describing messages make up the lowest level functionality of the `yaq` interface. Each individual communication between client and daemon involves one message being requested by the client and the response returned from the daemon. Each daemon supports a collection of messages for its own unique functionality, called the "protocol".

`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 which implement a trait must support all of the messages from the trait. Importantly, specific protocols can also implement arbitrary additional messages that are not defined by any trait.

The first hardware interface barrier: Multiple incompatible interfaces are used to communicate with each component

3

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: Coding interfaces add to a maintenance overhead seldom manageable by researchers. Here, we implement an approach useful for systems integration in the sciences which we call a Minimal Programming Interface (MPI). The MPI is essentially a reduced set of functions found in the hardware API. The MPI is defined as the minimal set of API functions for a given hardware component concluded by the researchers as necessary for the fuller system development. The value of this MPI is that it can be formed by wrapping and/or converting the API from its native language into a language known to be transparent to researchers, such as the programming language Python. The `yaq` traits feature acts as a tool in exposing the MPI from the base APIs often supplied by the hardware manufacturer, as the traits automatically define the primary set of key functions, and any additionally required functions are added by the researcher on top of the primary set.

The third hardware interface barrier: Certain specialty hardware have 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. For example, the Windows XP computer which has the obsolete interface card and drivers can be placed into a private network with only `yaq` interface communication. 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 3.10 communicates seamlessly with a daemon running Python 3.7.

The fourth hardware interface barrier: Some hardware interfaces are blocking. 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. In `yaq`, each daemon is only responsible for a single hardware interface. It is expected that each message call over the `yaq` interface will return rapidly, ensuring that client applications are not blocked for extended periods of time. This principle applies even when the message starts an action that might take several seconds to complete, such as homing a motor or initiating a measurement for a sensor. In these instances, the initial message simply starts the action and returns, with a separate message provided to retrieve results when they exist. In order to know how long to wait, the "is-daemon" trait provides a message called "is_busy" which should return `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.

III. EXPERIMENTAL FLEXIBILITY

Existing experimental orchestration software is often highly inflexible. An experimentalist will spend many hours in lab 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 their 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.

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, to hardware interfacing, and to 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.

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 specialized scientific knowledge to develop this software.? `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.

A `yaq` client can scale in complexity from a small, lightweight script all the way to a complicated graphical program, as shown in Figure 2. At the bottom, we represent the most lightweight interface to `yaq`, Python scripting. `yaqc`[18] is a Python client which is excellent for using in scripts or any other Python program. The code shown creates three client objects which 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.

In the middle, the same three hardware peripherals are represented in an interactive, graphical form. `yaqc-qtpy`[19] is a graphical application which builds interactive controls based on traits for any conceivable `yaq` protocol. The self describing `yaq` interface is used to provide graphical elements for the
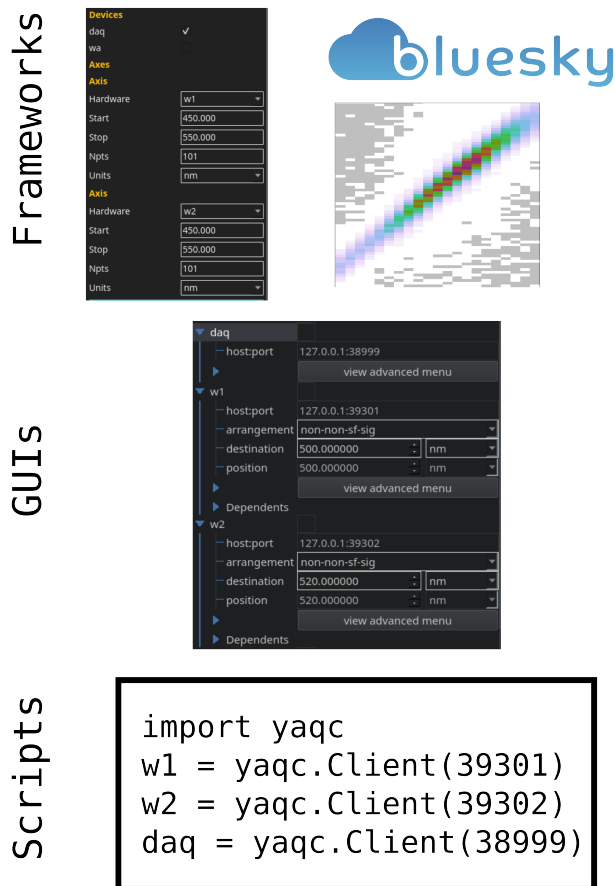
Frameworks

GUIs

Scripts

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

,

FIG. 2. `yaq` scales from simple scripts to integrations with featureful frameworks.

most commonly used messages. This is an invaluable tool as it provides a "free" graphical user interface (GUI) to any daemon.

At the top, we show several features associated with the integration between `yaq` and the Bluesky project[20]. Bluesky provides a powerful orchestration layer for conducting and recording data for a wide variety of experimental procedures. Shown in Figure 2, we see one such GUI for parameterizing a Bluesky procedure and a representation of the resultant data. `yaqc-bluesky`[21] provides a bridge to the Bluesky ecosystem. Similar translation layers could be built for a variety of orchestration layers such as PyMoDAQ[8], Instrumental[9], or TRSpectrometer[10].

All three types of clients represented in Figure 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 for different requirements of a custom instrument For example, `yaqc-qtpy` can provide a quick interface for setting and viewing hardware positions while Bluesky can focus on the actual data acquisition itself and the necessary planning on how the acquisition

should be performed.

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[22,23]. This need for exquisite hardware control has resulted in several prior attempts at "home-built" orchestration software[24–27]. 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 the active laboratory computers. Remote client development on viritual hardware has allowed client design to proceed without delaying ongling experiments. Clients developed by the Wright Group have proved flexible enough to be used on four laser systems, each with different complements of hardware. We expect that 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 separated from the particulars of any individual client. Often, this means that initial hardware enablement work can be done on a researchers personal machine before the new hardware peripheral is installed in the instrument. 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-
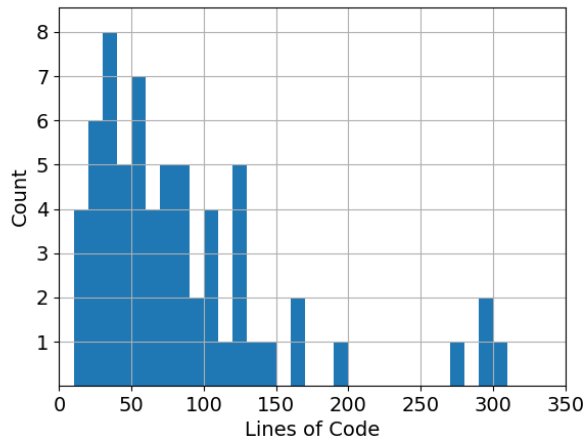
FIG. 3. Histogram of the number of lines for each implemented daemon. Some daemons are implemented in ways that share code, resulting in apparent line counts less than 10. For example, the Thorlabs APT[5] motor implementation supports at least eight different daemons with each specifying only a handful of constants. These are extreme examples which are not representative of most hardware interfaces, so we omit them here.

tion II, `yaq` gives you options to design using remote hardware or unusual interfaces when necessary.

We have created several tools to aide in daemon development. First, a Python library, `yaqd-core`[28], which implements shared functionality. Second, `yaq-traits`[29] is a command line application which 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. While lines of code is an imperfect metric, we use it here to represent the amount of work required to create a daemon for a new hardware peripheral. Most interfaces, such as Brooks MFC[30], have been implemented in as less 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 tooling libraries that Python now provides. In our experience, the process of creating a new daemon involves about a day of work after communication with the hardware using Python is well understood. There are currently 70 daemons in the yaq project supporting at least 50 kinds 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. The daemons are constructed under MPI methodologies and are more freely maintainable and adaptable than the original APIs. A living list of all daemons and supported

hardware can be found on the `yaq` website.

The Stahl Group at UW-Madison created a custom reactor which monitors gasses being produced or consumed in the reaction head-space.[31] 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, and so downtime will be minimized.

V. TECHNICAL DEBT

Years after the original researchers leave, large monolithic acquisition programs become unknowable, undocumented, and unmaintained. A graduate student discovers a long standing hard-coded conversion factor that is incorrect years after implementation. Scientists resort to sourcing a replacement for an old, broken oscilloscope due to their reliance of their software on that particular interface while newer, cheaper oscilloscopes are readily available. A graduate student is forced to meticulously reverse engineer the LabVIEW codebase that they inherited in order 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". Technical debt grows especially fast in academic environments where graduate students are involved in projects for a limited time.

The `yaq` approach attempts to minimize techinical debt by favoring many smaller, more easily adjustable single-purpose applications over larger, monolithic ones. For daemons, the purpose of each application is obvious and unambiguous. There is a strict, well defined interface which explicitly limits the kinds of interactions that are provided to the hardware, thus limiting 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`[32] 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 which 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 make 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.

Software documentation is famously difficult and thankless work. yaq attempts do 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.

yaq is open source software. Anyone can view, install, edit, and suggest changes to our growing collection of daemons and clients. Thirteen individuals have contributed code to the development of yaq.

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.

ACKNOWLEDGMENTS

DATA AVAILABILITY STATEMENT

Data sharing is not applicable to this article as no new data were created or analyzed in this study.

REFERENCES

[1]"Getting started with ni-daqmx," (2022), accessed: 2022-10-02.

[2]*Standard Commands for Programmable Instruments (SCPI)*, SCPI Consortium (1999).

[3]*MODBUS APPLICATION PROTOCOL SPECIFICATION*, Modbus (2012).

[4]*PICam^{TM} 5.x Programmer's Manual*, Teledyne Princeton Instruments (2021).

[5]*Thorlabs Motion Controllers Host-Controller Communications Protocol*, Thorlabs, GmbH (2022).

[6]L. R. Dalesio, M. R. Kraimer, and A. J. Kozubal, "Epics architecture," ICALEPCS **91** (1991).

[7]J.-M. Chaize, A. Gotz, 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," Review of Scientific Instruments **92**, 045104 (2021).

[9]N. Bogdanowicz, C. Rogers, S. Weber, Zakv, S. Pelissier, J. Wheeler, Cxz, F. Marazzi, Eedm, and I. Galinskiy, "mabuchilab/instrumental: 0.7," (2022).

[10]P. Tapping, "Trspectrometer," (2021), accessed: 2022-10-02.

[11]L. Koerner, "Instrbuilder: A python package for electrical instrument control," Journal of Open Source Software **4**, 1172 (2019).

[12]L. Campagnola, M. B. Kratz, and P. B. Manis, "Acq4: an open-source software platform for data acquisition and analysis in neurophysiology research," Frontiers in Neuroinformatics **8** (2014), 10.3389/fninf.2014.00003.

[13]"Pymeasure scientific package," (2022), accessed: 2022-10-02.

[14]G. Giesbrecht, A. Amsellem, T. Bauer, B. Mak, B. Wynne, Z. Qin, and A. Persaud, "Hardware-control: Instrument control and automation package," Journal of Open Source Software **7**, 2688 (2022).

[15]A. Shkarin, "pylablib," (2022), accessed: 2022-10-02.

[16]E. S. Raymond, *The new hacker's dictionary*, 3rd ed. (MIT Press, Cambridge, Mass, 1996).

[17]"Apache avro specification," (2022), accessed: 2022-10-02.

[18]"yaqc," (), accessed: 2022-10-02.

[19]"yaqc-qtpy," (), accessed: 2022-10-02.

[20]D. Allan, T. Caswell, S. Campbell, and M. Rakitin, "Blueskys ahead: A multi-facility collaboration for an a la carte software project for data acquisition and management," Synchrotron Radiation News **32**, 19–22 (2019).

[21]"yaqc-bluesky," (), accessed: 2022-10-02.

[22]S. Mukamel, "Multidimensional femtosecond correlation spectroscopies of electronic and vibrational excitations," Annual Review of Physical Chemistry **51**, 691–729 (2000).

[23]J. C. Wright, "Multiresonant coherent multidimensional spectroscopy," Annual Review of Physical Chemistry **62**, 209–230 (2011).

[24]R. 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).

[25]K. A. Meyer, *Freqncy-Scanned Ultrafast Spectroscopic Techniques Applied to Infrared Four-Wave Mixing Spectroscopy*, Ph.D. thesis, University of Wisconsin–Madison (2004).

[26]S. 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).

[27]B. J. Thompson, *Development of Frequency Domain Multidimensional Spectroscopy with Applications in Semiconductor Photophysics*, Ph.D. thesis, University of Wisconsin–Madison (2018).

[28]"yaqd-core," (), accessed: 2022-10-02.

[29]"yaq-traits," (), accessed: 2022-10-02.

[30]"yaqd-brooks-mfc-gf source code," (), accessed: 2022-10-02.

[31]C. 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," Review of Scientific Instruments **92**, 044103 (2021).

[32]"yaqd-fakes," (), accessed: 2022-10-02.

[33]A. Arkilic, D. Allan, D. Chabot, L. Dalesio, and W. Lewis, "Databroker: An interface for nsls-ii data management system," Proceedings of the 15th Int. Conf. on Accelerator and Large Experimental Physics Control Systems **ICALEPCS2015**, Australia (2015).

[34]A. Arkilic, D. B. Allan, T. A. Caswell, L. Li, K. Lauer, and S. Abeykoon, "Towards integrated facility-wide data acquisition and analysis at nsls-ii," Synchrotron Radiation News **30**, 44–45 (2017).

[35]L. J. Koerner, T. A. Caswell, D. B. Allan, and S. I. Campbell, "A python instrument control and data acquisition suite for reproducible research," IEEE Transactions on Instrumentation and Measurement **69**, 1698–1707 (2020).

[36]K. Lauer, "ophyd devices: Imposing hierarchy on the flat epics v3 namespace," Proceedings of the 17th International Conference on Accelerator and Large Experimental Physics Control Systems **ICALEPCS2019**, USA (2020).

[37]K. A. Meyer, J. C. Wright, and D. E. Thompson, "Frequency and time-resolved triply vibrationally enhanced four-wave mixing spectroscopy frequency and time-resolved triply vibrationally enhanced four-wave mixing spectroscopy," The Journal of Physical Chemistry A **108**, 11485–11493 (2004).