SMA-X: Versatile information sharing in and around telescopes, and beyond

Attila Kovács[a], Paul Grimes[a], Christopher Moriarty[a], and Robert Wilson[a]

[a]Center for Astrophysics | Harvard & Smithsonian, 60 Garden St, Cambridge, MA, U.S.A.

ABSTRACT

We developed the SMA eXchange (SMA-X) as a real-time data sharing solution, built atop a central Redis database. SMA-X provides efficient low-latency and high-throughput real-time sharing of hierarchically structured data among the various systems and subsystems of the telescope. It enables fast, atomic retrievals of specific leaf elements, branches, and sub-trees, including associated metadata (types, dimensions, timestamps, and origins, and more). At the Submillimer Array (SMA) we rely on it since 2021 to share a diverse set of ∼10,000 real-time variables, including arrays, across more than 100 computers, with information being published every 10 ms in some cases. SMA-X is open-source, and will be available to all through a set of public GitHub repositories in Summer 2024, including C/C++ and Python3 libraries, and a set of tools, to allow integration with observatory applications. A set of command-line tools provide access to the database from the POSIX shell and/or from any scripting language, and we also provide a configurable tool for archiving the observatory state at regular intervals into a time-series SQL database to create a detailed historical record.

Keywords: realtime database, information sharing, monitoring and control, observatory software, publish-subscribe, open-source

1. INTRODUCTION

When operating an observatory, its control system must gather information from hundreds of sources in quasi real time: telescope drive encoders, sensors from various equipment and devices, diagnostic data from instrumentation, weather data, various programs running on different computers etc. At the Submillimeter Array (SMA),[1] we monitor data from 8 antennas, 9 weather stations, 16 receivers, 48 correlator units, dozens of control computers, hundreds of programs and sensors. The distributed programs of the control system must have access to a selection of the shared information to manage observations, operate hardware optimally, and to create fully-described scientific data products. When components report values that are outside of their normal operating ranges, it is important to act without delay, both so that affected data can be flagged appropriately for problems, and to bring it to the attention of operators, who may be able to correct it. How well information is shared among systems in a telescope is a major factor in determining the efficiency of its operation. Even small latencies can compound in the chain of information flow and result in significant idle times, or else manifest in degraded telescope performance and/or poor data quality.

Different telescopes have found different answers to information sharing for their monitoring and control needs. Many observatories rely on peer-to-peer (P2P) communication, typically through ethernet (e.g. TCP/IP or UDP sockets), or other forms of digital links. Until recently, the SMA has relied almost exclusively on P2P information sharing both over commercial reflective memory hardware, and using Open Network Computing (ONC) Remote Procedure Calls (RPC).[2] Other telescopes, like APEX, developed their own custom P2P messaging systems,[3] e.g. with communication over UDP sockets. There are likely as many different P2P solutions used in telescopes today as there are telescope facilities themselves.

Further author information: Send correspondence to attila.kovacs@cfa.harvard.edu

2. BEYOND P2P

Such P2P systems have clearly been reasonably successful in providing the necessary means of communication among telescope systems. In some cases P2P is used primarily in relation to a centralized decision making 'control-system' software (such as APECS[3,4] at the APEX telescope). Alternatively, P2P links may route specific bits of information between various dependent programs of a distributed control system (see Fig. 1). However, P2P information sharing has at least two fundamental flaws: *inflexibility* and unnecessary *complexity*.

What happens when a new instrument, component is added to the system? Or, if a program is changed to have new switches, or it produces different/new values than before; or is migrated from one IP address to another (heavens forbid to a different network even)? The P2P system of information sharing does not usually handle such changes well. For every change, several programs may have to be explicitly modified to send or receive vital information regarding the new components, or to route data to/from a new address. Centralized control systems, such as APECS, may have to be re-initialized every time anything changes in how or what information is shared in the telescope ecosystem. Because control systems typically handle information from multiple sources in parallel, every such change has the potential to create new race conditions, bottlenecks, or even deadlocks. Such unexpected side-effects from changes are often non-trivial to trace and/or fix, especially because the developer too can only peek at the shared information through the programs that at on either end of individual P2P links.

A second, equally nagging issue of P2P systems, is that the number of P2P links is $\mathcal{O}(N^2)$ typically for N information sharing nodes present in a distributed control systems. Thus, when telescopes have hundreds of individual components that produce and/or consume shared information, the number of P2P links can easily be in the several thousand. Consequently, information flow can become tedious to trace and manage. When something does not work as expected, it may take a very long time to track down how related bits and pieces of information criss-crossed their way across the system, and where it all went wrong exactly.

Thus, while P2P systems can be pseudo-stable in an unchanging telescope environment, they all too often become visibly unstable in a constantly evolving observatory. As most observatories are in a perpetual state of development, perhaps P2P is not an ideal solution for them, in general.

2.1 Public Information & Decentralized Decision Making

One may observe that every bit of data is either produced or consumed by a particular component in the system, regardless of how complex that system is. There is no reason why the *producers* of information should prejudice who or what programs can access their shared data (or how). A weather station that measures humidity, does not need to have a definitive list of all programs in the telescope that need a humidity value. Conversely, a program that needs a humidity value, should also not care what specific component or program, at what IP address, provides the value. But that is exactly what P2P requires at its core. P2P dictates that either the producer of information send it to its consumer(s) explicitly, or else that the consumer(s) of information request it from the specific producer.

A better way is to share information without the explicit routing that P2P involves, such as via a public message board. Producers of information simply post their data to the public message board, and *any* consumer that needs *any* bit of shared information can obtain it from there. The general availability of information about all components is conducive to decentralized decision making also, as different programs may individually gather the information they need to make appropriate decisions on operating a particular sub-system without having to control all equipment from a central hypervisor software. A public message board can be implemented with a publish-subscribe (PUB/SUB) system (e.g. the one onboard SOFIA[5]), and/or via a real-time database (such as Redis or memcached).

2.2 PUB/SUB vs Persistent Real-time State

Publish-subscribe refers to messaging systems, where information that is *published* to a specific *channel* is immediately forwarded to the set of active *subscribers* of that channel. For example, SOFIA's Mission Controls and Communication System (MCCS)[6] published housekeeping data at regular intervals (e.g. 1 Hz, 10 Hz, 50 Hz)

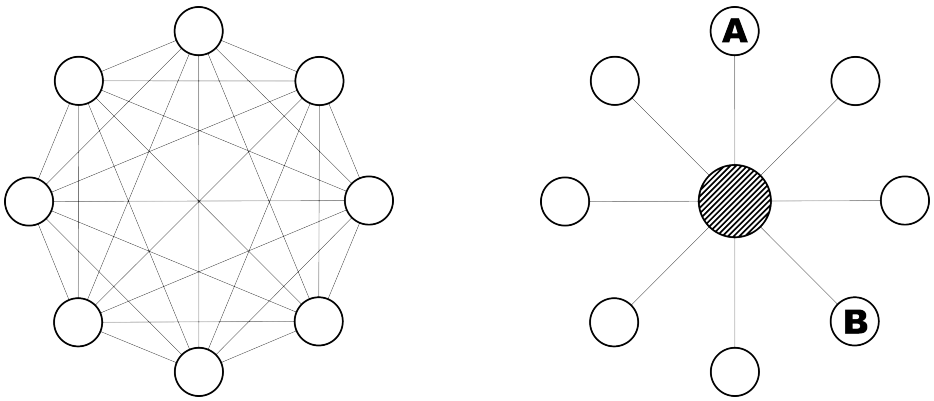

Figure 1. *Left:* Maximal network of decentralized P2P sharing among nodes, s.t. each node has access from information from all other nodes. *Right:* Centralized information sharing. The same graph may represent P2P sharing to a central (shaded circle) "control system" node, or else to a public message board. In the P2P case, nodes A and B are isolated from one another and do not have access to each other's shared data, unless the central control system explicitly forwards information from one node to another – decision making must be predominantly centralized. In the case of the central message board, however, nodes A and B have unfettered access each other's shared data, as well as data shared from any other node, so they can make their own control decisions as appropriate. A central message board, like SMA-X, enhances information sharing and enables decentralized decision making with a minimal network complexity.

in this way. Depending on what cadence was required by the consumer program, it would subscribe to the appropriate *channel(s)*, and would receive updated telescope data 'streamed' to it at the desired rate.

This works well for data that is broadcast frequently enough, such that it does not matter much if one misses an occasional update. If a consumer to the 1 Hz 'stream' subscribed to a channel just after a message was sent on it, it will have to wait at most a second before it receives the next update.

However, PUB/SUB is not so great when data is produced infrequently or at irregular intervals. Consider a program that publishes a message every time the telescope moves to a new celestial target. Another program that uses a subscription to that channel to determine what source is currently being tracked by the telescope, may be in bad luck if it subscribed to these updates just after the telescope was moved to a new source, and which it will be tracking for the following two hours. In this case, the unlucky consumer program would have to wait for up to two hours before it is finally informed what source is observed next. Until then it cannot know what it is being observed actually at present.

One solution to this problem, of course, is to publish irregularly and infrequently changing information at regular intervals (e.g. every second) also. However, when an observatory has tens of thousands of shared variables, it becomes impractical to publish all of them at a suitably high fixed cadence. Regardless of what cadence is chosen, it will either increase the volume of network traffic dramatically and/or result in unwelcome latencies due to the infrequency of updates.

The second public message board solution is to use a real-time database instead, which effectively stores a snapshot state of all shared variables at any given time. Producers of information update their state variables when these change, and consumers can poll them when they need it. However, this too can be abused easily. One may flood the network with high-frequency polling of slowly changing data, or else suffer unnecessary delays if polling data less frequently.

This is why we developed the SMA eXchange (SMA-X). It offers the best of both worlds: it can be used to provide the most current state of data regardless of when a client connects, and also notifies clients immediately when variables of interest are updated, without the need for frequent polling to detect changes in state – all with minimal network traffic to conduct transactions.

3. THE SMA EXCHANGE (SMA-X)

The SMA eXchange (SMA-X) is a data storage and messaging convention built on top a central Redis[*] database (version 4 or later). Redis is primarily a very efficient open-source key/value storage solution. Data are both labeled and stored verbetum as 'strings'. Beyond that, Redis (or one of its clones like Valkey[†] or Dragonfly[‡]) offers features that make it the ideal choice for a data sharing solution for distributed systems, in general:

- Stored data can be grouped into tables (hash tables in Redis terminology), which may be accessed efficiently and atomically both in their entirety or by individual fields.
- Supports pipelined (batch) mode access.
- Includes a PUB/SUB subsystem.
- Allows server-side scripting (via the LUA language) to implement higher-level atomic data access.

Table 1. SMA-X basic storage types

Data Type	Description
`int[8,16,32,64]`	signed integer value(s), with or without specific widths, e.g. `int` or `int32`
`float`	single-precision (32-bit) floating point value(s)
`double`	double-precision (64-bit) floating point value(s)
`boolean`	logical true/false value(s)
`string`	ASCII string value(s), stored in JSON-style[7,8] escaped form
`struct`	reference to a Redis hash table containing sub-structure data

Basic data types: Individual SMA-X variables are stored as 'fields' in Redis hash tables, either as scalar values or as flattened arrays. SMA-X supports all primitive and numerical data types used in typical programming languages. Individual variables are stored in serialized (ASCII) representation which allows platform independent access to these. See Table 1 for details.

Metadata: For every variable stored in SMA-X, we also store a set of essential metadata, such as the original variable type, its array dimensions, the precise time it has been shared, the host and/or program that provided the data, and statistics on how may times the data has been updated or accessed. For example, the original data type of field `bar` in table `foo`, is stored in the Redis hash table ⟨`types`⟩ under the field named `foo:bar`. See Table 2 for details. Beyond the essential metadata, additional optional metadata may be provided as needed (see Table 3). In the future we may extends the SMA-X standard to include further types of optional metadata also.

Hierarchical data: SMA-X is designed to store data organized in hierarchically, much a like a file-system would be. It allows data to be grouped in a logical way by systems, subsystems, components etc, so they may be easily located and related bits of information can be accessed together easily. The SMA-X storage convention allows for efficient atomic access of structured data both as a whole, or any of its branches or leaf-nodes individually. (This we see as a crucial advantage over the native JSON[7,8] storage format for hierarchical data introduced in Redis 7, which can only be accessed as a whole, and never by its components).

Atomicity Bundled data, such as the included metadata and/or structures, are both written to and retrieved from the SMA-X database in a single atomic operation. Thus, the user need not worry about concurrencies, such as one client getting incompletely updated data while another is in the middle of setting new values for these. This will never happen with SMA-X as long as the user reads and writes bundled data with singular calls.

[*] https://redis.io
[†] https://valkey.io
[‡] https://dragonfly.io

Update notifications: Every time a variable or entire structure in SMA-X is updated, a set of notifications on specific PUB/SUB channels bearing the variable's name (as well as those of its parent hierarchies) are sent, so that subscribed clients are notified immediately of state changes on any/all variables of interest, and can act on these as appropriate.

Remote settings: Extending on the set of features described above, we also specify a convention for 'commanding' program settings remotely through SMA-X. Remote settings are analogous to Remote Procedure Calls (RPC), except that they do not provide a 'computed' return value for the specific caller. (You may think of them as RPC with `void` return type.) The SMA-X database simply stores identical sets of `commanded` and `actual` values that represent a program's settings. The program may monitor the `commanded` set for changes, and then report applied values back in the `actual` set, which in turn a client program may monitor for confirmation that changes were applied (and how exactly).

Remote messaging: Beyond the PUB/SUB channels that carry notifications for all SMA-X variables, we also standardize the naming for channels that may carry messages from individual programs, and which can be monitored by clients, e.g. to selectively monitor progress or errors / warnings remotely.

Table 2. Essential metadata, stored for every variable in SMA-X.

Meta Table	Description
⟨`types`⟩	storage data type of each variable, see Table 1.
⟨`dims`⟩	array dimensions for each variable.
⟨`timestamps`⟩	precision UNIX timestamps when the variable was last updated in SMA-X.
⟨`origins`⟩	host and/or program that provided the variable
⟨`reads`⟩	number of times the variable as read by clients
⟨`writes`⟩	serial number, i.e. number of times the variable was updated in SMA-X

Table 3. Optional metadata, stored and accessed as needed.

Meta Table	Description
⟨`descriptions`⟩	a concise description of what data the variable stores
⟨`units`⟩	physical unit of the data (e.g. if not SI).
⟨`coords`⟩	coordinate system description for array data in one or more dimensions

3.1 SMA-X Client Tools & Libraries

We provide C/C++ and Python 3 libraries, as well as a simple set of command-line tools to help access and maintain your custom SMA-X database. The command-line tool `smaxValue` can be used to query SMA-X variables, including the essential metadata, or to list the contents of specific tables, while `smaxWrite` can be used to add/update SMA-X data from the command-line. Both command-line tools can be used with scripting languages, such as `bash`, `perl` or similar.

The C/C++ and Python libraries come with generally similar set of features. Python, of course, allows for a cleaner, type-agnostic interface, and for exception handling. In contrast, the C/C++ library offers families of related functions, with separate calls providing the same functionality for each supported data type, and it relies on return values and `errno` to indicate error states. Both libraries offer detailed API documentation to guide users.

Below we summarize some of the main features of the C/C++ library, noting here only that some of these may be implemented somewhat differently for the Python client library. Refer to the documentation included with the libraries themselves on their particular implementations.

The SMA-X C/C++ client keeps up to three separate TCP/IP channels open to the SMA-X Redis server: (1) for interactive transactions; (2) for pipelined (batch) requests; and (3) for PUB/SUB notifications and management thereof.

Automatic type conversions: The C/C++ library implements automatic type conversions for leaf-node access, such that the client software can access the shared data as a different type from how it was produced. Because data is always stored as string(s), the parse type of data may be anything really, regardless of the original storage type, as long as the stored string value(s) can be parsed as the desired type. This includes both widening and narrowing conversions, such as from `int16` to `int8`, `int64`, or `float`, and vice-versa – as well as conversions between numerical, logical, and string types also. The Python library is intrinsically not strongly typed and allows for most of the same conversions with little or no effort.

Pipelining (batch mode): Data that is pushed to SMA-X is sent out via the pipelined connection (provided it's available). As such, data sharing from your application is always non-blocking and incurs no unexpected overheads, since it does not wait for confirmation back from the Redis server. This fire-and-forget approach ensures that sharing of information is safe even when used from within timing critical processes and threads. Data can also be retrieved asynchronously in batches through the pipeline connection. Queries can be submitted to a queue, and the responses that received for these are processed asynchronously in a separate background thread. The user or application can perform other time critical tasks while the responses are gathered, and then either wait until the requested batch of data is available (via appropriate synchronization points inserted into the queue), or else request an asynchronous callback. Because pipelined transfers do not require a chain of round-trips to the SMA-X server over the network, they can provide orders of magnitude higher throughput for bulk transfers than sequentialized individual interactive transactions.

Lazy access: The SMA-X library provides lazy access for data retrieval. Lazy retrieval is most useful when one needs information more frequently than it is produced, without flooding the network with frequent polling requests. It essentially amounts to maintaining locally cached copies of the requested SMA-X variables. The caches are either invalidated if SMA-X notifies of an update, or else seamlessly updated in the background. (The caching mode can be selected for each lazy variable individually if desired.) When your application needs the current value of a variable, it is either retrieved instantly from the local cache (with zero network traffic), provided that the cache is valid, or else its current value is fetched from the SMA-X database.

Semaphores & callbacks: We provide means to wait (block the current thread) until some variable or some set of variables change. Monitored variables to wait upon can be designated individually or via glob patterns, and a timeout value can also be specified to return with an error (or exception) in case no changes were detected in the allotted amount of time. Alternatively, one may specify a callback function to be invoked asynchronously when a variable with matching name or pattern is updated.

Resiliency: We designed our C/C++ and Python libraries to be generally resilient to intermittent outages of the SMA-X server or the network infrastructure. If locally produced data cannot be pushed to the SMA-X server, the libraries will maintain a local store of the pending updates, and will try reconnecting to the SMA-X server in the background at regular intervals. If and when SMA-X is successfully reconnected, the pending local updates are pushed before any new data is sent to or received from the SMA-X server.

3.2 Creating a Historical Record

By nature SMA-X stores only a snapshot of the current state of all shared variables. It has no sense of history or prior state information, by design. (Redis offers such features, but SMA-X does not use these.) It is however often useful to keep a historical record of the observatory's runtime state. It can help diagnose problems after they occur; or validate data at a later time; and engineers may use it to troubleshoot hardware and/or software components by having access to a long-term behavioral record. At the SMA we have been keeping such a historical record of shared data (even before SMA-X) at 1-minute resolution, going back some 20 years. We find it an invaluable resource, and the possibility of being able to check specific conditions and sensor values at the telescope on a given day and time some years back is a priceless feature.

For this reason, we have also developed a connector daemon which can regularly sample and/or snapshot all, or a selection of, SMA-X variables at regular intervals and store them in a PostreSQL database,[9] with or without a TimescaleDB extension for more efficient retrieval of the time-ordered data. At the Submillimeter Array, we store a snapshot of nearly all SMA-X variables once every hour, and store updates to changing variables once per minute as necessary. The connector application stores metadata history also, when changes to metadata are detected. The connector application is highly configurable. You can set:

- the database location, name, and user credentials.
- whether or not TimescaleDB extension should be used.
- the frequency of regular updates for changing variables (e.g. 1 minute).
- the frequency of full snapshots of the SMA-X database (e.g. 1 hour).
- variables and glob patterns specifying which variables to include in or exclude from the archival.
- a maximum size for variables that will be archived automatically.
- a maximum age of variables to be included automatically (so that orphaned data is not polluting the historical record forever).
- a sampling for array variables where that only every n^{th} array element is stored.
- to force archival of select variables or patterns even if they would otherwise be excluded by one of the other settings above.

The connector daemon can be integrated with and managed via `systemd`, e.g. to start automatically after boot. It will monitor the SMA-X database continuously, and insert new tables into the PostgreSQL database as necessary for any new variable(s) appearing in SMA-X, provided these aren't excluded from archiving by the configuration.

A particularly welcome aspect of having a historical record stored in a PostgreSQL database, is that it is very easy to create time-series visualizations of these, e.g. with Grafana. As such, one can easily produce web pages for visual monitoring of shared variables of particular importance.

3.3 Current Status and Future Plans

SMA-X has been in use at the Submillimeter Array since 2021, where it has proven to be highly stable and reliable. We use both the C and Python libraries routinely for our everyday operation. SMA-X is also used with the new control system of the MIT Haystack 37-m telescope.[10] We are in the process of open-sourcing our configuration and source code, which involves some modularization, and improving the documentation. The SMA-X source code, libraries, and tools will be accessible by the end of Summer 2024 via the set of GitHub repositories[§] listed in Table 4.

Table 4. SMA-X GitHub repositories.

GitHub repository	**Description**
`Smithsonian/smax-server`	server configuration and `systemd` integration.
`Smithsonian/smax-clib`	C/C++ client library and command-line tools.
`Smithsonian/smax-python`	Python3 client library.
`Smithsonian/smax-postgres`	PostgreSQL connector application

If you want early access to any of the SMA-X repositories before they are made public in Summer 2024, please send a request for access to the first author (see contact details on the title page).

There is still room for SMA-X to grow in capabilities. We readily foresee a number of avenues for adding new features or improving existing ones. At present we are considering the following areas, in which SMA-X can extend and expand functionality:

[§]You may find the specific repositories by appending the listed repository names in Table 4 to `https://github.com/`.

- Support for Redis Sentinel for redundant, high-availability, Redis server configurations.
- Initializing optional metadata and static configuration data via a separate persistent configuration database such as a MongoDB or an SQL database. This way SMA-X may provide the same access protocol for static data, which is not directly produced by the real-time system.
- More optional metadata, such as normal operating ranges, and critical levels for measured values, which may be used e.g. by and automated warning system.
- Direct support for complex-valued data types (e.g. `complex[32,64]`).
- Standardized support for non-persistent, streaming-only data via PUB/SUB. It may be useful to provide bundled, self-contained, data packets (e.g. real-time telescope pointing information) published at a fast cadence (>10 Hz) to support low-latency real-time client applications for these.
- Use Redis list storage types to implement data queues (FIFOs) on SMA-X, such as for managing an observing queue, or a set of tasks that have to be executed sequentially.
- Asynchronous Remote Procedure Call (RPC) through SMA-X. SMA-X could allow for more proper zeroconf remote calls using the Redis PUB/SUB infrastructure, with unique program IDs (e.g. `host:program[:task]`) in lieu of physical addresses. We have a draft protocol for RPC over SMA-X, but it has yet to be finalized, and implemented for the client libraries.
- Client libraries for other languages, e.g. Java, Rust, Go. We welcome external contributions for implementing additional client support for the programming language(s) of your choice.
- We plan to package the SMA-X server, libraries, and tools for Linux distributions (e.g. Debian and Fedora RPM packages), as well as for Homebrew (MacOS X) to make it more easily available to users and applications.
- Enhanced security. SMA-X presently relies on restricting access to internal networks to prevent unauthorized access. Redis provides additional security features, such as database user authentication, SSH-tunnel-only access, and TLS support – which can enhance and further protect the database from outside attacks.
- Let us know what other added feature would make SMA-X work better for you.

4. CONCLUSION

SMA-X offers an appealing, fast, and versatile solution for distributed monitoring and control in and around telescopes, or for other distributed systems. It relies on a central Redis server, as a public message board within the telescope ecosystem, for sharing and accessing structured data to/from various programs and nodes. Data access is atomic both for arbitrary data branches, sub-branches, or leaf elements, and includes metadata that describe the stored values. SMA-X unites persistent storage of the current state of shared information with a PUB/SUB notification system for updates – a powerful combination which allows clients to efficiently access the information both on demand and/or when values are updated. The dual mode information sharing minimizes network traffic, and eliminates latencies beyond those of the underlying networking layer. The centralized Redis server also minimizes the number of network connections necessary to $\mathcal{O}(N)$ instead of the typical $\mathcal{O}(N^2)$ links required in distributed P2P systems. And, because all information is available to all programs/nodes within the ecosystem, SMA-X supports distributed decision making in a way P2P typically does not. As such SMA-X provides a superior solution to existing P2P models for data sharing in distributed systems in general.

We provide a set of C/C++ and Python libraries that client applications can use to bring the most out of SMA-X. These libraries offer additional features, such as asynchronous pipelining (fast bulk data access), lazy access (with or without background caching), semaphores and callbacks that can be used to act immediately on specific state changes, and resiliency for intermittent network or server outages. Simple command-line tools provide basic access to SMA-X that can be used with any scripting language (e.g. `bash` or `perl`) also. Furthermore, we provide

a configurable application that can be used to create a long-term historical record of all shared state variables in a PostgresSQL database. These may be visualized, e.g. with Grafana, to show time evolution of select monitoring states, or can be used for diagnostics in general. (At the SMA, we have such a record going back to 20 years with a 1 minute cadence, predating even SMA-X – and we find it enormously useful for diagnostics, and for data quality related checks even years after the fact).

It is our plan to maintain and develop SMA-X for the foreseeable future. As such, we expect to expand and enhance the capabilities it provides, and evolve the client libraries we maintain to match. All of our existing SMA-X related code will be open-sourced during Summer 2024, and we hope to offer packaged versions in the near future also. We will welcome your questions, comments, suggestions, or any other feedback that you may provide to make SMA-X better.

ACKNOWLEDGMENTS

The Submillimeter Array is a joint project between the Smithsonian Astrophysical Observatory and the Academia Sinica Institute of Astronomy and Astrophysics and is funded by the Smithsonian Institution and the Academia Sinica.

REFERENCES

[1] Ho, P. T. P., Moran, J. M., and Lo, K. Y., "The Submillimeter Array," ApJ **616**, L1–L6 (Nov. 2004).

[2] Birell, A. D. and Nelson, B. J., "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems* **2**, 39–59 (1984).

[3] Muders, D., Hafok, H., Wyrowski, F., et al., "APECS - the Atacama pathfinder experiment control system," A&A **454**, L25–L28 (Aug. 2006).

[4] Muders, D., König, C., Schaaf, R., et al., "APEX Control System (APECS): Recent Improvements and Plans," in [*Astronomical Society of the Pacific Conference Series*], Ruiz, J. E., Pierfedereci, F., and Teuben, P., eds., *Astronomical Society of the Pacific Conference Series* **532**, 563 (July 2022).

[5] Becklin, E. E., Tielens, A. G. G. M., Gehrz, R. D., and Callis, H. H. S., "Stratospheric Observatory for Infrared Astronomy (SOFIA)," in [*Infrared Spaceborne Remote Sensing and Instrumentation XV*], Strojnik-Scholl, M., ed., *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series* **6678**, 66780A (Sept. 2007).

[6] Papke, B., Brock, D., and Graybeal, J. B., "Extensible and flexible software architecture for the SOFIA mission controls and communications system," in [*Airborne Telescope Systems*], Melugin, R. K. and Röser, H.-P., eds., *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series* **4014**, 399–410 (June 2000).

[7] "ECMA-404: The JSON Data Interchange Format," standard, ECMA International (2013).

[8] "Information technology — The JSON data interchange syntax," standard, International Organization for Standardization (11 2017).

[9] Stonebraker, M. and Row, L. A., "The design of POSTGRES," *ACM SIGMOD Record* **15**, 340–355 (1986).

[10] Kauffmann, J., Rajagopalan, G., Akiyama, K., et al., "The Haystack Telescope as an Astronomical Instrument," *Galaxies* **11**, 9 (Jan. 2023).