ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THMBCM007

REFLECTIVE SERVERS: SEAMLESS OFFLOADING OF RESOURCE INTENSIVE DATA DELIVERY *

S. Clark†, T. D'Ottavio, M. Harvey, J. P. Jamilkowski, J. Morris, S. Nemesure
Brookhaven National Laboratory, Upton, U.S.A.

Abstract

Brookhaven National Laboratory's Collider-Accelerator Department houses over 550 Front-End Computers (FECs) of varying specifications and resource requirements. These FECs provide operations-critical functions to the complex, and uptime is a concern among the most resource constrained units. Asynchronous data delivery is widely used by applications to provide live feedback of current conditions but contributes significantly towards resource exhaustion of FECs. To provide a balance of performance and efficiency, the Reflective system has been developed to support unrestricted use of asynchronous data delivery with even the most resource constrained FECs in the complex. The Reflective system provides components which work in unison to offload responsibilities typically handled by core controls infrastructure to hosts with the resources necessary to handle heavier workloads. The Reflective system aims to be a drop-in component of the controls system, requiring few modifications and remaining completely transparent to users and applications alike.

BACKGROUND

The Control System of the Collider-Accelerator Department (C-AD) at Brookhaven National Laboratory provides the operational interface to RHIC and all the other accelerators in the C-AD complex. Over 77,000 Accelerator Device Objects (ADOs) provide the software interface to more than a million settings and measurements for accelerator equipment [1]. ADO servers may run on different hardware platforms, but a majority of the ADOs in C-AD are hosted on Front End Computers (FECs) with limited memory and CPU resources. Efforts to upgrade those systems are often impeded by upgrade or redesign costs, labor efforts, and scheduling concerns. As such, effective and efficient use of existing resources is paramount.

The communications protocol used by ADOs has four primary data operations: synchronous gets (blocking while retrieving data), synchronous sets (updating a set point), metadata fetches (retrieving static properties about a readback or set point), and asynchronous gets (receiving live streams of data updates). Of these, the first three are stateless and have little impact on resource usages beyond the call context. The last, asynchronous gets (asyncs), is a stateful operation that require the maintenance of information such as client identifiers, data requests, and data queues, which can consume substantial memory when numerous clients are connected, or when high bandwidth asynchronous data is requested. Many C-AD applications establish asyncs by default, which can quickly consume resources on an FEC causing a crash of the ADO software. This interrupts connections to other users, and in the worst cases interrupts operations. Async load has been attributed to FEC downtime in the past, and has caused certain FECs to be identified as low-resource relative to demand. These units must be treated with caution when interacting with hosted ADOs to prevent resource exhaustion. This can require extraneous communication between users, developers, and the control room to coordinate use, limiting both operational and development efforts.

Previous efforts to develop a data reflection system took place over the last decade [2], but fell out of use due to reliability and maintainability issues within the system. The previous reflective tools suffered from connection issues, inherent to the underlying communication layer implementation, which had the potential to place multiple systems into bad states. Remedying this required process restarts and manual intervention to restore communication between clients and devices. Since then, general purpose calculation engines have been used as stand-ins for a proper reflective system, but are limited by overhead and configuration challenges. These issues of reliability, configurability, and maintainability were identified when outlining plans for an upgraded reflective system, and a new architecture was designed from the ground up to satisfy those needs.

REQUIREMENTS

The primary requirement for this project is straightforward: develop a system which removes asynchronous load from resource-constrained ADOs. However, the system must do so while also:

- Handling connection interruptions gracefully, allowing for quick and correct recovery from transient communication failures
- Being easy to configure and deploy, with a conspicuous way to examine connected clients and reflected devices
- Fitting seamlessly into the Controls ecosystem with minimal changes necessary to existing clients and without user intervention

SOFTWARE ARCHITECTURE

The software architecture of this system has been designed to be highly modular, with different logical units being broken into separate processes as outlined in Fig. 1. The first

* Work supported by Brookhaven Science Associates, LLC under Contract No. DE-SC0012704 with the U.S. Department of Energy
† sclark@bnl.gov

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THMBCM007

unit, the Reflective Server, provides the core functionality of the system. These are responsible for interfacing directly with client applications and providing facilities to communicate with ADOs efficiently. The second unit is the Reflective-Aware Central Name Service, which allows interested clients to quickly identify and retrieve details necessary to communicate with Reflective Servers.

Reflective Server

The Reflective Server (RS), core of the architecture, proxies ADO protocol requests between clients and devices. The client-facing interface (frontend) and device-facing interface (backend) are logically separate code, with an intermediate binding layer providing necessary logic to process requests. Any number of RS instances can be spawned, typically proportional to the number of FECs needing reflection. By convention, an RS instance handles requests for multiple ADOs hosted on a singular FEC. By assigning one RS to each FEC, the backend connections to the ADO can be aggregated, necessitating only one active TCP/IP connection to that FEC to handle any number of asynchronous get requests. However, this is not a restriction; one RS can reflect any number of ADOs hosted on any number of FECs.

Asynchronous Requests Asynchronous get requests (asyncs) from the frontend receive special handling when passing through an RS. When processing an async, the RS first checks if it has requested the given data from the reflected device through its backend interface. If the backend does not have an established async for the requested data, the backend requests that data be delivered asynchronously from the device. Once a backend async request is established, the RS adds a pointer to the frontend client to a queue of interested parties for the data, and returns a successful status to the client. If the backend fails to establish an async request due to some reason (data unavailable, internal device error, etc.), the associated error data is passed back to the client for processing just as if the client were communicating directly with the ADO.

Once a backend async is active, the device will deliver any data updates to the RS backend as they are available. These updates may be the result of a externally changed setpoint, or an internal data update - this functionality is dependent on the specific ADO implementation. When the backend receives an async update from the ADO, the RS binding logic begins fanning that data out to all interested frontend clients. Each frontend client has its own send-queue and thread, ensuring that misbehaving clients do not cause performance issues for the rest.

When a client is ready to exit, the ADO protocol states that a cancellation request shall be issued to the RS. The RS will remove the client from the interested parties and cleanly terminate the frontend connection upon receiving a cancellation from a frontend client. If cancellations are received for all parties interested in a specific piece of data, the RS will request through the backend that the ADO stop delivering the given data. This ensures that the RS is not processing, nor is the ADO delivering, data that is unnecessary. If all data requests are cancelled to an ADO, the RS will cleanly terminate the backend connection to that ADO to preserve both RS and FEC resources.

Synchronous Get Requests For synchronous get requests, processing is done to prevent unnecessary calls to the ADO. If a backend asynchronous connection already exists between the RS and the ADO which for the requested data, the RS simply returns the requested data from an internal cache. This improves efficiency as the RS is guaranteed to have the latest data from the ADO. If an active async does not exist, the frontend get request is simply proxied through the backend to the reflected ADO. Proxying adds a second round trip to the overall latency, so utilizing cache data has the potential to reduce overall request-response latency by half.

Other Requests As designed, all other requests are proxied from frontend client to the ADO nearly directly. Minimal request inspection is done to determine the destination ADO for the request. The response from the backend is passed back to the frontend unchanged. There is overhead associated with the round trip time from backend to ADO, and data marshalling within the RS binding layer, but overall these sum to only a 2x increase in latency for requests.

Reflective-Aware Central Name Service

The second aspect of the overall reflective architecture is the Reflective-Aware Central Name Service (rCNS), tasked with locating and directing clients to specific Reflective Servers.

Much as an RS is a proxy to an ADO, rCNS is a proxy to the Central Name Service (CNS). CNS is a core service of C-AD's controls system; it allows applications to lookup hostname and RPC program information for any given ADO using a common identifier such as a name. The choice to create a separate rCNS process prevents the master CNS database from being changed to support reflection. This allows clients to bypass reflection altogether simply by querying the master CNS server rather than rCNS, and provides a simple fallback in the case that rCNS or an RS fails.

rCNS provides the same information over the same protocol, but additionally tracks RS instances and which ADOs they reflect. When rCNS receives a request, it first checks to see if the name being requested is reflected by an RS. If so, rCNS sends back its internal data which the client will use to initiate a connection with the RS, and in turn leverage the benefits of the reflective system. If rCNS determines the requested name is not a reflected ADO, then it simply forwards the request onward to the master CNS server. This request will be filled by CNS as normal, and rCNS simply passes the response back up to the client. The client will connect directly to the ADO in this case.

Registration with rCNS is a dynamic process, with information pointing to RS instances being stored in memory. When an RS is brought online, it checks in with rCNS and

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THMBCM007

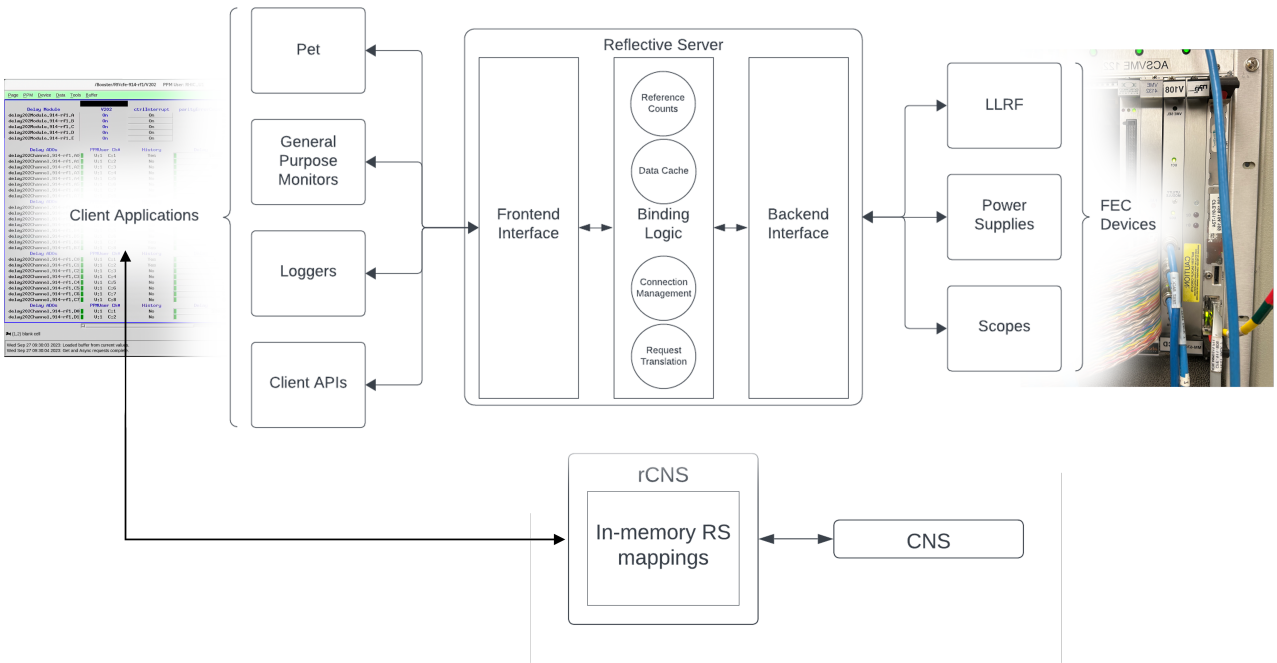

Figure 1: Data flow across the Reflective Server.

provides its hostname, RPC program information, and a list of ADOs it reflects. Likewise, when an RS exits, it deregisters from rCNS; this removes its information from rCNS. Upon deregistering, an RS has the option to trigger fallback behavior - meaning that incoming requests are forwarded directly to the ADO in the RS's absence. This is an opt-in behavior, as many ADOs leveraging RS technology reside on FECs unable to cope with the true number of requests, so a dead RS with fallback enabled would quickly overwhelm the FEC with incoming asynchronous connections.

Client Applications The reflective system has been designed to work seamlessly with existing client applications. Clients simply need to query the rCNS instance to leverage reflection for ADOs. In C-AD, this is performed by setting the `CNSHOST` environment variable to the host running the rCNS process; typically, this is set to the master CNS host. Once that variable is changed, any requests will query the rCNS instead and retrieve details for an RS (if one exists for the requested ADO). Providing the reflective system as an opt-in allows for a staged roll out. The roll out can be done to specific users or specific workstations simply by modifying the environment in each scope.

Additionally, sample applications have been provided which are reflective-aware without necessitating switching to reflection for processes for a given user or workstation. This was accomplished by releasing a binary which sets the `CNSHOST` variable appropriately before all application logic, allowing requests originating from that application to use reflection. This was invaluable in the commissioning process, giving an easy way to test while also providing an escape hatch in case of failure. Long-term, special applications are not expected to be used as the reflective system is adopted more widely.

HARDWARE ARCHITECTURE

No special hardware is involved with deploying a Reflective Server. Commodity, off-the-shelf servers and workstations running a common Linux distribution can host an RS instance. C-AD utilizes a virtualized environment for server deployments, and a dedicated virtual machine (VM) was commissioned for hosting an RS. The reflective VM was created with 4 CPU cores running at 2.2GHz, and 11GB of memory. These specifications are more than capable of running multiple RS instances, and far exceed the resource limitations of the FECs hosting reflected ADOs. Hosting RS instances on a local server allows flexibility in deployment strategy, resource allocation, and network placement.

IMPLEMENTATION

The entire reflective stack was designed and implemented in Python. Since the reflective system does not have hard latency constraints, Python was a natural choice with which to develop this project. Python as a development environment has become a major player within C-AD over the past 5 years [3]. The tools and libraries developed for Python provide communication support with feature and performance parity with respective C++ and Java counterparts.

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THMBCM007

The Reflective Server implementation in Python is a modular structure, with the communication frontend, backend, and binding layers all being separate classes. The binding layer interfaces with frontend and backend classes to provide the necessary logic and translation for data flow between them. The binding layer is designed to be thread-safe, using common concurrent data structures provided by Python, to ensure safety regardless of the design of frontend or backend implementations.

Frontend The frontend interface acts as a server - that is, it provides an API compatible with the ADO protocol with which clients can communicate. This is done by wrapping custom logic around SUN-RPC communication code, which is the basis for the ADO protocol. Initial attempts at the project began with using application-level communication tools, but those libraries were foregone as too much functionality is abstracted away for use cases in the RS design. Contributing to the success of the project is the fact that ADO communication libraries for Python are designed in a modular way as well, exposing much of the needed low-level functionality such as request inspection and modification, and connection management.

Backend The backend interface acts as a client, connecting to FECs and requesting data from them. This use-case is straight forward and did not require any unusual handling to make it work in the RS. This allowed existing, high-level communications tools to be used when implementing the backend interface. These tools are well tested and in common use across the complex, and include features such as robust reconnection management, connection multiplexing to ADOs, and efficient processing algorithms - the benefits which are realized by the libraries' use in RS.

PERFORMANCE

Performance comparison of the Reflective Server to direct communication with ADOs was completed in Python, utilizing C-AD standard communication tools along with the python `timeit` module for benchmark analysis. These tools may incur additional latency inherent with using Python, but both sides of the comparison utilize the same tools so that base latency would be moot.

Synchronous Gets (Non-Cached) Non-cached synchronous gets represent a simple proxy between frontend client and backend ADO; no additional processing to the request is performed, so the expected overhead from passing through the RS should amount to no more than $1 \times RTT$.

Concrete testing in Python supports this claim. Table 1 displays results of the timeit tests. The results are reproducible and represent a 1.7x overhead when using reflection, reinforcing the efficiency of the RS implementation in Python. Additionally, since all other operations (sets, metadata, etc.) are passed through as well, these metrics also apply to those as well.

Table 1: Synchronous Get Performance, No Data Caching

Type	# of Calls	Time / call (µsec)
Direct	1,000	334
Reflected	500	594

Synchronous Gets (Cached) When the RS is actively servicing asynchronous gets for a property, it stores the most recently received value of that property internally. Serving this data to synchronous get requests allows the RS to avoid unnecessary round trips to the ADO. As such, there is no expected latency associated with this operation when communicating with an RS. Again, data acquired through testing supports this as shown in Table 2.

Table 2: Synchronous Get Performance, With Data Caching

Type	# of Calls	Time / call (µsec)
Direct	1,000	334
Reflected	1,000	317

In the test cases above, the average time when communicating with the RS is actually less than communicating directly to the ADO. This may be due to factors such as network congestion, routing latency within the network, and other external factors.

Asynchronous Gets Asynchronous get benchmarks are more complex than the prior benchmark, as they are encumbered by more than just simple latency. These asynchronous calls must also factor in the number of clients handled and the ability to deliver the requested data to those clients effectively.

Testing leveraged a property of asynchronous ADO data delivery: timestamps. All data is timestamped by the device prior to sending on the wire, which can be used by a client to determine the true origination time of an update without network or processing latency.

Table 3 displays a comparison of the latency encountered during asynchronous delivery (arrival time − timestamp, in milliseconds) between direct connections and reflected connections. Each test establishes the listed number of independent connections, and awaits 3 data deliveries. This is run 3 times for each connection count, and the latencies averaged.

Table 3: Asynchronous Get Delivery Delay

# Clients	Direct (ms)	Reflected (ms)
10	0.987 ± 0.259	1.997 ± 0.602
50	1.62 ± 0.679	4.10 ± 1.93
100	2.07 ± 0.612	5.14 ± 1.83

ISBN: 978-3-95450-238-7 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2023-THMBCM007

Expectations Notably missing from the requirements defined at the start of the paper are performance guidelines. The RS architecture is being developed as an application-level tool, foregoing support for real-time processing capabilities or timing restrictions, so strict performance requirements were not paramount in the development process. This is appropriate for tools such as GUIs and loggers which work on human-level timescales, or which work asynchronously to process data delivery. However, feedback mechanisms which may need to react to real-time conditions are less suited to utilizing reflection. The choice to use reflection must be made on a case-by-base basis, strongly dependent on the application and the context in which it is used, and the architecture was designed such that applications may opt-out if necessary.

FUTURE WORK

Broader rollout of the reflective system across the C-AD complex will put the system's scalability to the test. A robust, centralized configuration management system has been proposed to simplify spawning and configuring RS instances. This will likely be implemented as a Python web interface using the FastAPI or similar. This could also serve as an interface to inspection of RS clients, providing developers a view into what is utilizing reflection and any potential issues.

Analysis of existing ADOs will be necessary in order to leverage all potential benefits of the RS architecture. Now, the get-caching functionality is opt-in, toggled by a runtime flag. Certain ADOs have get-code, or specific functionality such as polling hardware that is run when a get request is processed. This conflicts with the RS get-caching feature as gets to an RS fail to trigger this get-code on the underlying ADO. Get-code is largely phased out in favor of asynchronous data updates from hardware, but a more critical analysis must be done to ensure RS deployment does not interfere with any legacy ADOs which still leverage get-code.

The modularity of the reflective system may prove invaluable in providing a merging ADO and EPICS components in the future EIC controls systems. The frontend and backend interfaces to the RS clear path to developing extensions which allow for EPICS clients to communicate with RS instances, and with RS instances to reflect IOCs. Additionally, future frontend and backend interface may interoperate to allow for a bridge between ADO and EPICS tools and devices.

REFERENCES

[1] D. Barton *et al.*, "Rhic control system", *Nucl. Instrum. Methods Phys. Res. A*, vol. 499, no. 2, pp. 356–371, 2003. doi:10.1016/S0168-9002(02)01943-5

[2] B. Frak, T. D, M. Harvey, J. P. Jamilkowski, and J. Morris, "Application of Transparent Proxy Servers in Control Systems", in *Proc. ICALEPCS'13*, San Francisco, CA, USA, Oct. 2013, pp. 475–478. https://jacow.org/ICALEPCS2013/papers/MOPPC157.pdf

[3] S. Clark, P. Dyer, and S. Nemesure, "Standardizing a Python Development Environment for Large Controls Systems", in *Proc. ICALEPCS'21*, Shanghai, China, 2022, paper MOPV049, pp. 277–280. doi:10.18429/JACoW-ICALEPCS2021-MOPV049