ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOBL05

# PHOTON SCIENCE CONTROLS: A FLEXIBLE AND DISTRIBUTED LabVIEW™ FRAMEWORK FOR LASER SYSTEMS

B. A. Davis*, B. T. Fishler, R. J. McDonald
Lawrence Livermore National Laboratory, Livermore, California, USA

*Abstract*

LabVIEW™ software is often chosen for developing small scale control systems, especially for novice software developers. However, because of its ease of use, many functional LabVIEW™ applications suffer from limits to extensibility and scalability. Developing highly extensible and scalable applications requires significant skill and time investment. To close this gap between new and experienced developers we present an object-oriented application framework that offloads complex architecture tasks from the developer. The framework provides native functionality for data acquisition, logging, and publishing over HTTP and WebSocket with extensibility for adding further capabilities. The system is scalable and supports both single server applications and small to medium sized distributed systems. By leveraging the application framework, developers can produce robust applications that are easily integrated into a unified architecture for simple and distributed systems. This allows for decreased system development time, improved onboarding for new developers, and simple framework extension for new capabilities.

## INTRODUCTION

In contrast to large experimental physics programs, small to medium size experiments and test-beds generally have less resources in terms of manpower, funding, and schedule. Developers are often faced with the task of standing up a distributed control system from scratch under tight deadlines with limited personnel. In these situations, it is imperative to choose a programming language that allows for quick hardware integration and prototyping.

Under these circumstances, NI LabVIEW™ software is often chosen for several reasons. First, it has an extensive hardware ecosystem with options for benchtop, distributed, and embedded hardware systems [1]. Interfacing with these systems from the LabVIEW™ development environment is streamlined and there are offerings for systems across the spectrum of determinism. Simple DAQs provide baseline functionality for non-deterministic applications, and soft and hard real-time situations are handled by RTOS and FPGA applications, respectively.

LabVIEW™ software is also attractive due to its shallow learning curve and low barrier to entry for those without a classical programming background. It uses a graphical programming style combined with a "dataflow" paradigm for organizing functionals and variables and defining execution order [2]. Many workflows for data acquisition, analysis, and logging are built in, and examples and documentation abound. It is simple for novice developers or even end users to create a baseline DAQ system to collect experimental data.

The result is an attractive platform for developing small scale experimental systems. In many cases, control system developers need not get involved at all - scientists and operators can quickly develop the skills to work with LabVIEW™ programming. However, this story becomes less clear when moving from small scale systems to medium scale systems with multiple distributed Front-End Processors (FEPs). Leveraging NI hardware remains an attractive prospect, as it eliminates the need for custom RTOS machines or FPGA boards to handle deterministic applications. However, the very advantage of easy software development can quickly become a burden instead.

Simple LabVIEW™ applications are singular in purpose – they interface with a small number of devices, acquire data, perhaps execute a sequence, and log data to disk. This can be accomplished by a novice developer, or even an end user, as previously mentioned. However, more often than not, such simple systems suffer from a lack of scalability and extensibility. Of course, a piece of software designed to control a single experiment has no need for scaling or extension, provided that system requirements are well-defined before the development begins (a tall assumption, but one we take for granted here).

The disconnect arises when taking similar software development practices and applying them to a larger scale, distributed system. While developing LabVIEW code to control a single system can be accomplished by those with little to no previous software engineering background, developing a distributed, extensible, and scalable system for a larger system requires more experience, skill, and rigor.

Often for a simple system, there is a single developer who creates an application to run the experiment. But for systems of increased complexity, multiple developers of varying skill levels must work together to create a series of interconnected applications across a number of FEPs. In such a situation, a unified architecture must be developed to ensure scalability across the system. Similarly, extensibility becomes key to adding new capabilities over time as the system evolves, as a larger scale system will likely be in operation for longer than a small testbed.

Thus the ideal architecture for developing LabVIEW™ applications for mid-scale distributed control systems must be scalable for any number of devices and FEPs, extensible for adding capabilities across the lifetime of a project (and ideally to future projects as well), and – most importantly – accessible to developers at all skill levels.

To accomplish these goals, we have developed an object-oriented distributed architecture for LabVIEW™ applica-

* davis287@llnl.gov

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOBL05

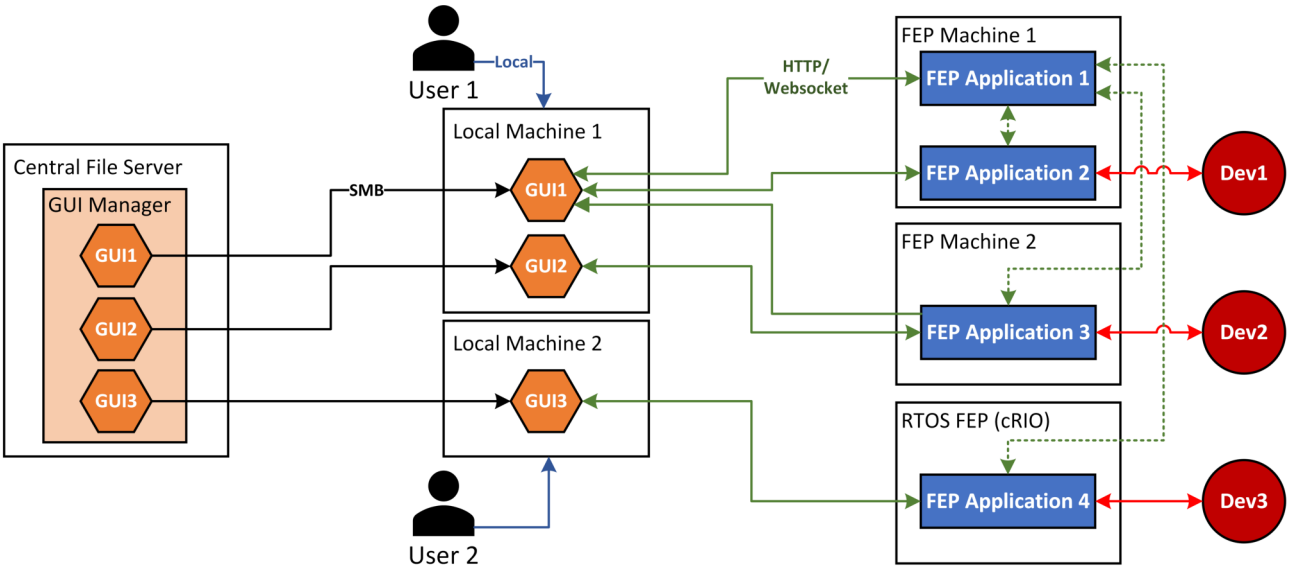


Figure 1: An example system architecture for a small project. Arrows represent communication over the LAN. Black arrows represent Server Message Block (SMB) protocol (mapped network drives), blue arrows represent local (mouse/keyboard) interaction, green arrows represent HTTP or Websocket connections, and red arrows represent device communication.

tions entitled Photon Science Controls (PSC). The architecture leverages inheritance and polymorphism to provide an extensible framework with discrete, encapsulated components. It natively provides flexible inter-application communication over HTTP and Websocket to support systems distributed over a Local Area Network (LAN), with capability to support other protocols as well.

Perhaps most importantly, it is designed to offload common software tasks from developers so that they can focus on discrete functional blocks. Tasks such as intra-application communication, logging, system health, and the aforementioned inter-application communication are pre-packaged and separately source-controlled. This ensures that the architecture remains accessible for novice and experienced users alike, while guaranteeing that functionality remains equivalent across all system applications.

## SYSTEM ARCHITECTURE OVERVIEW

Systems using the PSC architecture are designed as peer-to-peer systems composed of FEP applications and client applications. FEP applications are designed to handle device interfacing, sequencing, supervisory logic, and real-time calculation. These are the applications that use the PSC architecture. Client applications include GUIs, real-time status verifiers, and data displays. FEP application peers can communicate with any other FEP application on the network. Client applications can also communicate with any FEP application, but do not connect with each other. Figure 2 shows a example representation of this network architecture.

Combining FEP applications and client applications allows for the development of a distributed control system with user access available from any connected terminal. Figure 1 shows an example distributed system designed under this paradigm. The example system has four FEP applications distributed across three FEPs. Applications two through four handle device communication, and application one runs some supervisory application that requires data from the other three. The client GUI applications are served up from a central file server, which ensures that they are available from any local machine on the LAN.

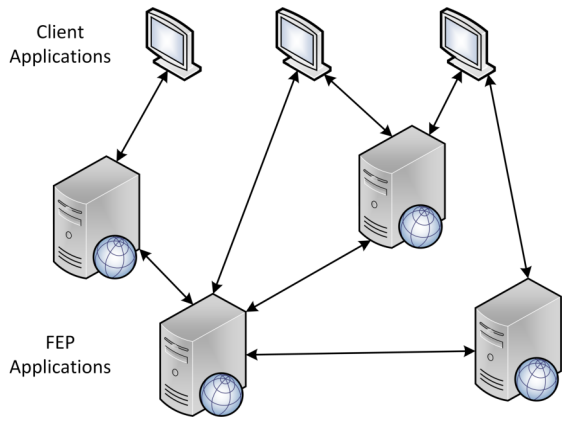


Figure 2: Peer to peer network architecture with FEP applications and client applications.

All communication between client applications and FEP applications, and between FEP applications is done over TCP - specifically using HTTP and Websocket. Users interact with the FEP applications through the remote client GUIs, which can send commands and receive data from the FEP applications. FEP applications can manage multiple connections from clients and peers, allowing for simultaneous monitoring from multiple local workstations. This is espe-

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOBL05



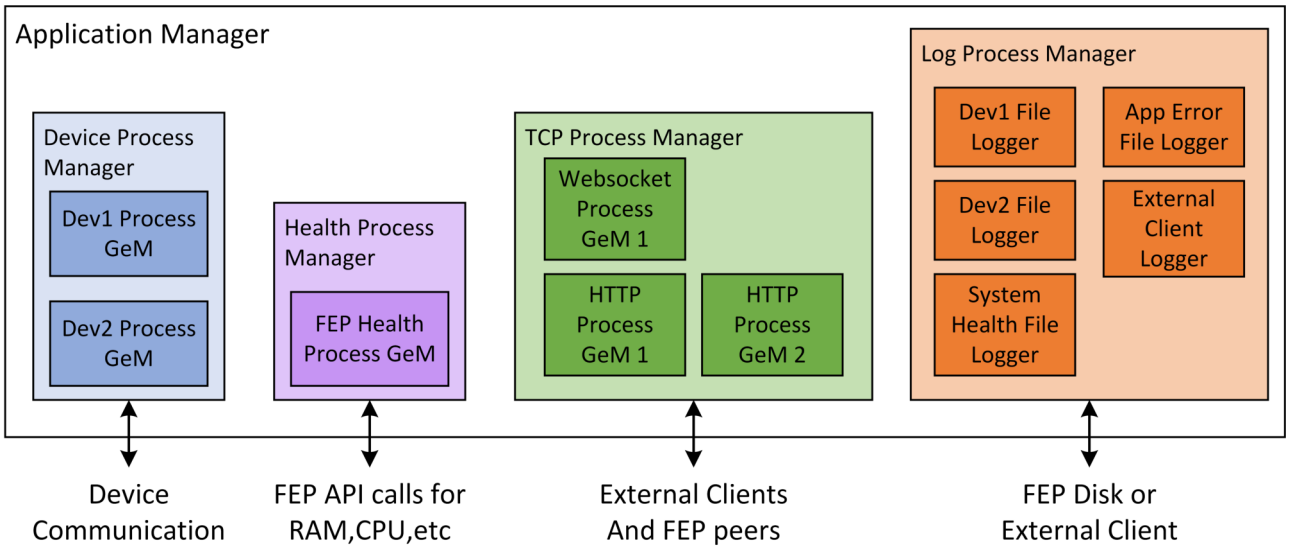


Figure 3: Sample FEP application with hierarchical architecture. Each Manager and Process is a self-encapsulated process running asynchronously.

cially important to support system designers and operators through the commissioning, qualification, and operational lifetime of the system as UI and workstation needs change.

The flexibility of the PSC architecture allows for it to be used on any target with a LabVIEW runtime. This includes Windows machines for non-deterministic applications, and NI hardware targets such as cDAQ and cRIO embedded processors for RT applications. These hardware devices run Linux RT as their RTOS, and cRIO targets also include an FPGA layer for hard real time applications. This allows the architecture to cover the full range of determinism needs for the system without requiring rework or adaptation.

## FEP APPLICATION ARCHITECTURE

### *Application Architecture Overview*

FEP applications form the core of the distributed control system and cover most of the functional requirements of the project. They communicate with devices; acquire, process and log data; broadcast and request data from other peers; and handle communication with client applications to send and receive data and commands from users.

The functions of a given FEP application are broken into discrete chunks called Generic Messengers (GeMs). Each GeM is a self-contained asynchronous process that handles a defined subset of the FEP application functionality. GeMs in an FEP application are launched in a manager-worker hierarchy. There is a top-level Application Manager GeM, one to many mid-level Process Manager GeMs, and one to many Process GeMs. Process GeMs control the core behavior and provide the main functionality of the application. Process Managers and the Application Manager allow for data roll-up from the Process GeMs as well as high-level application management functionality.

Figure 3 shows an example of this application hierarchy with some typical GeMs. This particular application would communicate with two devices, monitor the health of the FEP (CPU usage, RAM, etc), manage communications with external applications, and log device and application data to disk and an external client like an industrial Historian. Process managers and the Application manager would handle high level application behavior - monitoring the health of different processes, managing processes closing or being instantiated, etc.

There is no defined limit to the number or type of GeMs present in a single application. Different classes, devices, protocols, and loggers can be supported to accomplish the purposes of the application as long as they conform to the structure of the GeM ancestor class.

### *Flexible Functionality Distribution*

Intra-application communication between GeMs is a microcosm of the peer to peer inter-application communication scheme of the distributed system. Each GeM is individually addressable so that data can be passed around the application as necessary. For example, the Dev1 CSV File Logger in Fig. 3 would subscribe to data from the Dev1 Process stream to log it to disk. Like the peer-to-peer relationship between FEP applications at the system level, any GeM can get data from or issue commands to any other GeM in the FEP application.

This nested approach to communication allows for maximum flexibility in defining application and system organization. Depending on system requirements, FEP applications can be organized by functionality, device type, or logical subsystem grouping. GeM processes that fulfill the system requirements are then distributed across the FEP applications to match these functional groups. System hardware archi-

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOBL05

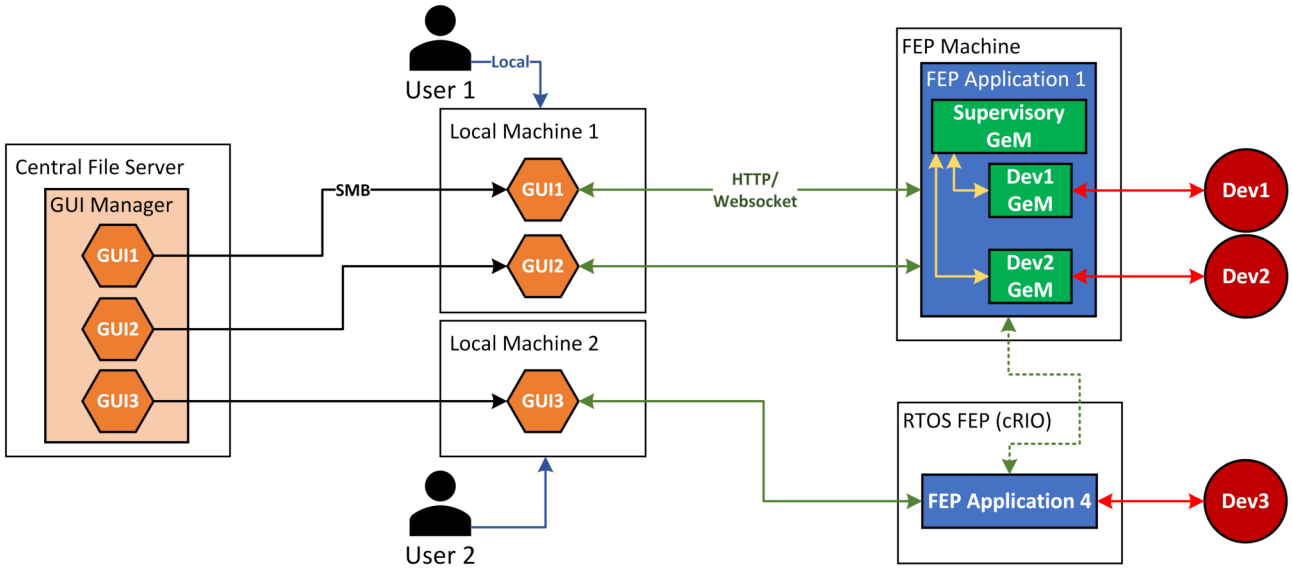


Figure 4: Redistributed architecture from Fig. 1. Note that the functionality of applications two and three has been moved into GeMs in application one. Yellow arrows denote internal communication (queues). This architecture is functionally identical but may provide different performance characteristics that would prefer its usage. Not pictured are the GeMs handling the external communication.

tecture, performance requirements, and logical delineations can all drive these decisions.

For example, Fig. 4 shows how the example system in Fig. 1 could be rearranged. We have consumed the functionality of FEP applications two and three into FEP application one, and instead used the internal data lines for the supervisor rather than external (HTTP, Websocket) to get their data. Note that FEP application four remains on a different FEP, so FEP application one still uses external communication for its data.

This reorganized system would perform the same tasks as the architecture presented in Fig. 1, but may provide performance benefits that would make it preferable. In short, a PSC2 system architecture provides the capability to tune the level of distribution to match physical and performance needs at the macro "distributed system" level and the micro "application GeM" level.

### *Class Hierarchy and GeM Functionality*

Figure 5 shows a class diagram for the Generic Messenger ancestor class. Note that most GeM classes are descendants of the Process class, as that is where the core functionality of an application is executed. However, Process Managers and the Application Manager also inherit from the GeM ancestor class, which means they inherit similar functionality to Process classes.

The functionality provided by the GeM ancestor includes five key items. The ancestor provides a common process main that ensures these five items are all implemented identically across all descendant GeMs.

**Data polling / creation** Every GeM includes a method to poll and create data at a predefined rate. For device process classes, this would include the device status data. For other GeMs, this provides an internal status. As an example, a logger class would provide information on its logging speed, how many items are left in the logging buffer, the current file, etc. Data is generically typed in a key-value structure for easy handling across GeMs and peer applications.

**Command Handling** Every GeM has a command handler that allows it to receive messages from other GeMs in the system. This is the method by which data is passed throughout the application. A command message is usually bundled with a response queue, which allows for a response to be sent back upon command action completion.

**Lossy and Non-Lossy Data Streams** After the GeM data is produced in the polling loop, it is piped into two data streams. The first, a lossy stream, is a single element queue. The element is overwritten upon each polling cycle, hence the designation as lossy. This is what the command handler can use to report the latest data sample upon request. It is often used for discrete requests, like by a TCP process for sending data to a GUI.

The second data stream is a non-lossy stream. Data is pushed into a buffered queue so that each sample is captured. This is the preffered method for passing data to loggers or other processes that need a complete data history. GeMs can subscribe to this data stream by sending a subscription request over the command handler.

**Thresholding and Alarming** Particularly for device process GeMs, it is important to monitor data values for

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOBL05



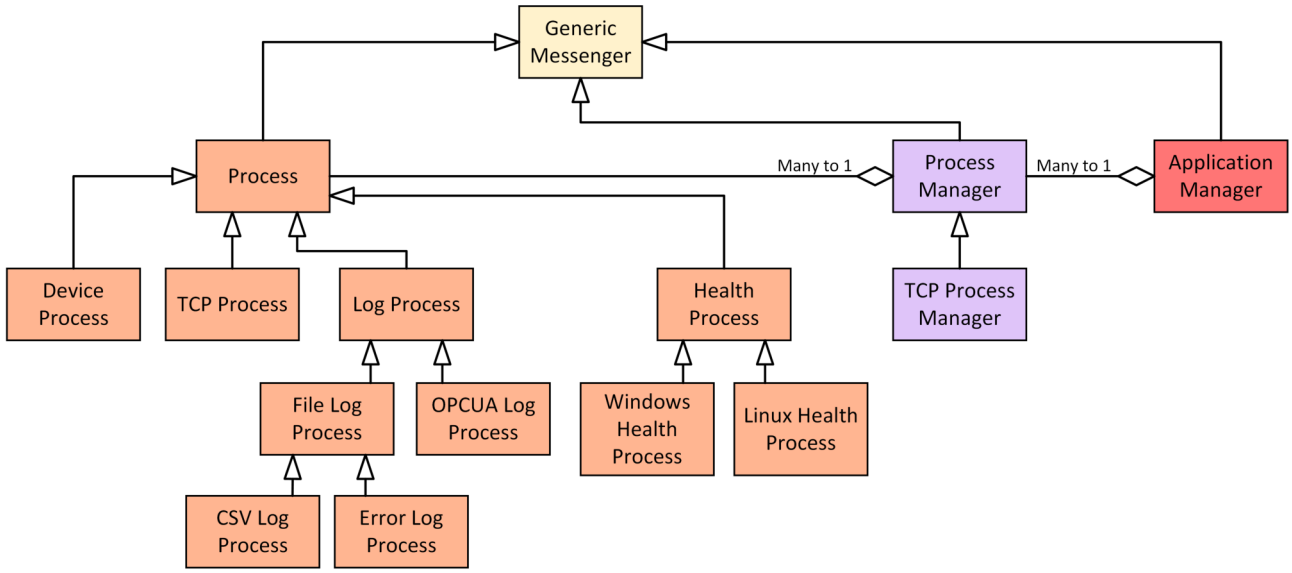


Figure 5: Generic Messenger Class Hierarchy. Most available architecture GeMs are of the "Process" descendancy, as this is where functionality is most heavily defined.

anomalies and take actions if they exceed pre-defined levels. Every GeM has a built in threshold handler that allows for definition of levels and actions to take based on named data values. These thresholds can be numeric or boolean. Based on the level of the data under review, UI warnings and alerts or Machine Safety System (MSS) actions can be taken.

**Internal Error Handling** Occasionally GeMs will encounter an internal error and self-terminate. The error handling built into all GeMs ensures that these errors are logged to disk for review. This is essential for addressing issues that occur during commissioning and operations.

### *GeM Lifeline and Application Startup*

Each GeM in the system goes through a similar lifeline. They are instantiated and initialized, then spun off to run an asynchronous process main. They are then terminated upon task completion or application end and run a cleanup routine.

PSC uses a Factory style object creation scheme [3] upon application startup. When a PSC application is loaded, an application factory is created. This factory object (not a GeM) loads a configuration file that defines the objects needed for the application to run. It then loads the necessary classes into memory and instantiates them.

The launch process at application startup is recursive in nature, based on the application hierarchy. The factory creates an application manager GeM, within which are a number of process manager GeMs who each contain a number of process GeMs. GeMs are then launched from the bottom of the hierarchy to the top – processes are each spun off, then the corresponding process managers, and then finally the application manager. This ensures a consistent boot order that is defined in the application configuration. A defined boot order is important to ensure that GeMs that depend on other GeMs are initiated in dependency order.

Not all GeMs are initiated at application startup. It is entirely possible for GeMs to be initiated by events that occur during application operation. As an example, a TCP Process Manager (see Fig. 3) listens for external connections and instantiates TCP processes to handle them during runtime. These processes persist until the connection is closed or timed out, and then self-terminate.

### *Data Recursion*

All GeM data is wrapped up recursively through the application hierarchy (Process -> Process Manager -> Application Manager) (see Fig. 6) In other words, each Process Manager's data packet contains all the data packets of its worker Processes, and the Application Manager's data packet contains all the data packets of its worker Process Managers. This allows for an external client or peer to get all the data from an application by making a single query to the top-level Application Manager.

### *Addressing and External Communication*

As previously mentioned, GeMs can communicate with other GeMs in a specific FEP application, and FEP applications can communicate with other peers and client applications. Every FEP application in a distributed PSC2 system has an IP address and associated TCP ports. Furthermore, each GeM within an application is assigned a URL based on the application hierarchy. This URL serves as an address for both external and internal communication. URLs are generated by the following pattern: **IPaddress:port/ApplicationName/ProcessManager/Process**.

As an illustrative example, **185.20.20.1:5437/SampleApplication/DeviceManager/Device1** would refer to the Device1 process of the Device Manager Process Manager on

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOBL05

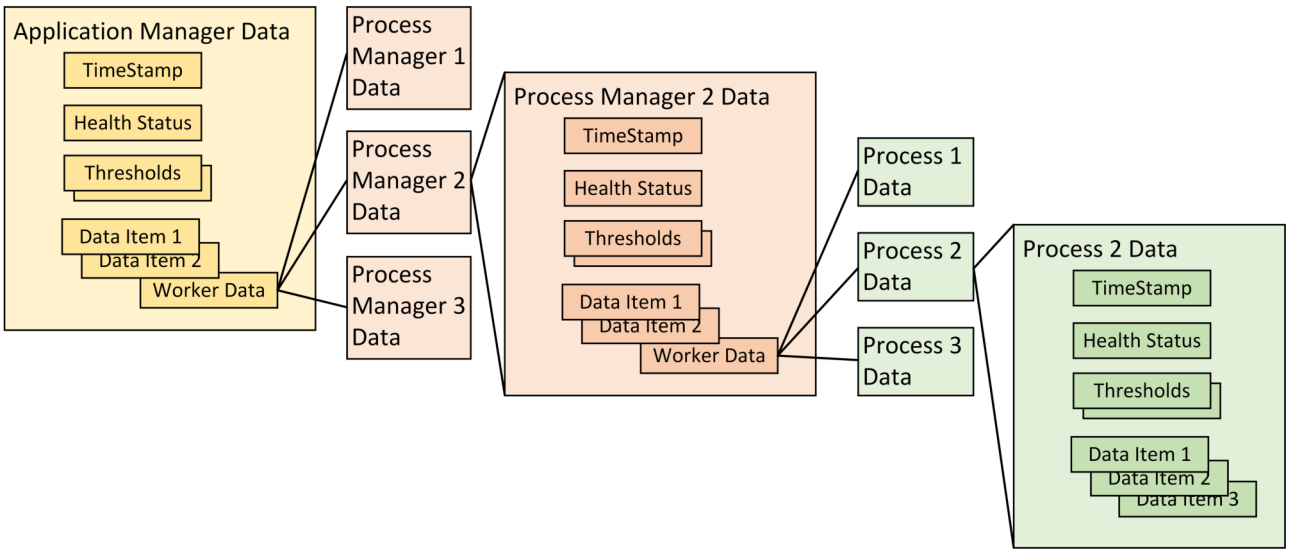


Figure 6: Data Recursion in FEP applications. The top-level application manager contains all the data from the GeMs in the application.

the Sample Application hosted on 185.20.20.1. The port 5437 may be the port setup for HTTP, Websocket, or some other protocol.

The same URLs are used as identifiers for the command queues of each GeM in the application. GeM peers send commands to others' command queues by name. Thus, the unique identifier URL for a GeM serves as a routing path for both external and internal communication.

External communication is routed through a TCP Process Manager GeM, which instantiates worker TCP Processes, one for each connection. Figure 7 shows the dataflow for connecting to an external application. The TCP Process Manager maintains a listener on the specified TCP port. Upon receiving a connection request, it instantiates and launches a TCP Process GeM. This GeM establishes the TCP socket connection with the external application. Messages are sent from the external application to the TCP Process, which then forwards them to other GeMs by URL. The TCP Process GeM persists until the connection closes or times out, at which point it self-terminates.

The TCP Process Manager can instantiate any number of TCP Process GeMs to support multiple connections. These TCP Process GeMs can be HTTP connections, Websocket connections, or other TCP protocols. There is a single connection for each external application, which ensures that data is not corrupted by multiple writers/readers on a single socket.

### *Application Configuration and User Sets*

Configuration items in a PSC application exist across three levels of accessibility - "hard-coded" (lowest accessibility), "static," and "configurable" (highest accessibility). Hard-coded items are configuration items that cannot be modified without rebuilding the application. These are protected by source control and are collected in an Application Configuration file. Static items can be modified without rebuilding the application, but are only applied on startup. Configurable items can be loaded and saved during runtime. Both static and configurable items are collected in User Sets.

**Application Configuration File** Every GeM maintains a list of configuration items that define its behavior. These form the basis of the Application Configuration that defines the GeMs included in a specific application build. The Application Configuration is loaded upon application startup, and is used by the Application Factory to instantiate all the GeM objects of the application. It lists the participating GeMs, and their configuration parameters that define their behavior. These can include polling rates, timeouts, links to other GeMs, etc. These values are considered hard-coded because they define the behavior of the final application. Because of this, application configuration files are protected as source code, and require rebuild upon changing.

**User Sets** Device Process GeMs maintain a series of separate configuration files known as User Sets. User Set files allow operators to save and recall device states in order to facilitate different modes of system operation. As with application configuration items, each Device Process GeM maintains a list of user set items. Each item in a user set list is tagged as "startup only" or not. This tag is what separates static and configurable items. Static items are only applied when the application is restarted, so changes to such items in a user set will not be applied immediately. Configurable items, on the other hand, can be recalled at any time during operations.

As an example, imagine a flow controller that is handled by a Device Process GeM. Perhaps there are some warning/alarm thresholds associated with the device that we would like to have the flexibility to change without rebuilding the application (i.e., they cannot be hard-coded in the

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-MOBL05



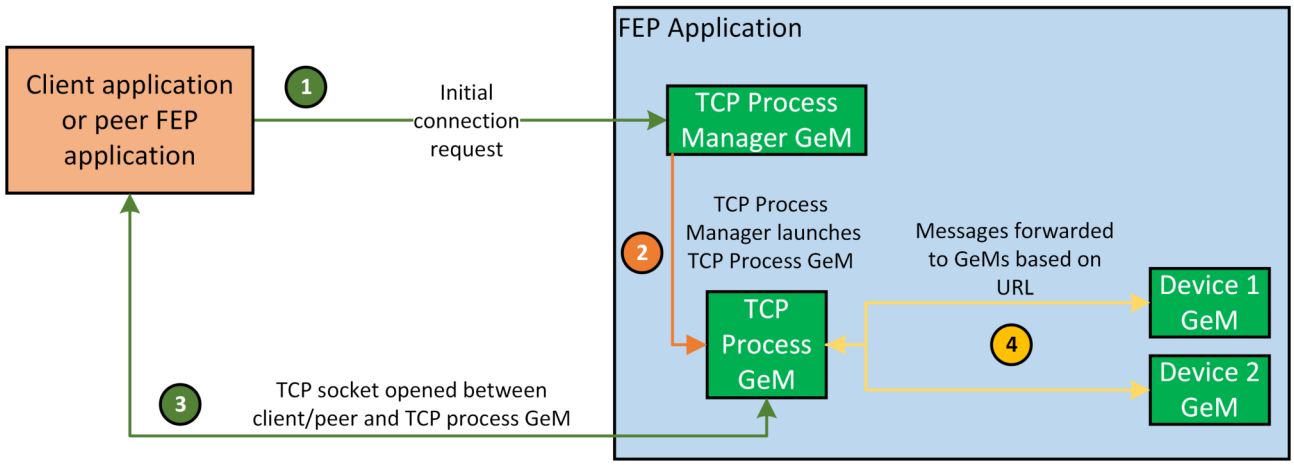


Figure 7: External Communication Process. The TCP Process Manager listens on the specific port for an initial message. Once the message is sent, the Process Manager instantiates a TCP Process to handle the socket connection. The TCP Process persists until the connection is closed or times out. Commands are sent by the client or peer to the TCP Process GeM, which forwards them to their destination based on URL.

application configuration). However, we do not want them to change without review and management approval. These would be an appropriate candidate for static (startup-only) behavior. We may also have a setpoint for the flow controller that needs to change based on what phase of an experiment is currently running. This may be a good configurable item - we can save one user set for each phase, and recall the appropriate one at runtime.

User sets are organized by name, and can be saved and loaded from client UI applications. This gives operators flexibility in defining, modifying, and managing the user sets that need to be applied for continuous operation. Table 1 summarizes the three types of configuration items.

Table 1: Configuration Item Summary

| Type | Associated File | Accessibility |
|---|---|---|
| Hard-Coded | Application Configuration | Rebuild Application |
| Static | User Set (startup-only tag) | Restart Application |
| Configurable | User Set | On Demand |

## CONCLUSION

The use of the PSC architecture helps to bridge the gap between small and large scale experimental systems by providing a scalable and extensible framework for rapid development timelines. Encapsulating functionality into interconnected Generic Messenger objects allows for unlimited flexibility in defining the distribution of system requirements over the system LAN. By leveraging inheritance and polymorphism, base functionality is preserved across all descendant GeM classes, providing a simple workflow for developing new features and capabilities.

By abstracting functionality into the classes that form the architecture, developers can focus solely on developing code that fulfills system requirements without having to deal with high level processes like communication and logging. Taken all together, the architecture provides an excellent solution for medium scale distributed systems and projects with funding, time, and personnel limitations.

## ACKNOWLEDGMENTS

The authors would like to acknowledge the work of Daniel Smith, Chen Lim, and Sridhar Kuppuswamy for their work on the previous iteration of the architecture.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Released as LLNL-CONF-827711.

## REFERENCES

[1] Designing Control Applications with Data Acquisition Hardware and NI-DAQmx, https://www.ni.com/en-us/innovations/white-papers/09/designing-control-applications-with-data-acquisition-hardware-an.html

[2] Benefits of Programming Graphically in NI LabVIEW, https://www.ni.com/en-us/innovations/white-papers/13/benefits-of-programming-graphically-in-ni-labview.html

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Creational Patterns: Factory Method," in *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Boston, MA, USA: Addison-Wesley Professional, 1994, ch. 3, sec. 3, pp. 107-117.