# LINUX-BASED SOFTWARE CONTROL FOR XENON SYSTEMS

Reynold Suarez, Charles W. Hubbard, James C. Hayes, Tom R. Heimbigner, Jennifer M. Mendez, and Brian T. Schrom

Pacific Northwest National Laboratory

Sponsored by the National Nuclear Security Administration

Award No. DE-AC05-RLO1830/PL09-MPR-NDD05

## ABSTRACT

This paper will review software control solutions for systems requiring automated control and analysis. Written in the C++ language under the Linux operating system (OS), a client/server architecture was developed for automated systems enabling modularized processes to perform system control. In this scheme, narrowly focused clients and servers communicate with each other to perform tasks necessary for system execution. Each hardware interface, such as analog input/output modules, instrumentation, and detectors, have a server associated with it that can be accessed individually with various client interface programs. These servers also inherently support reporting all sensor values to a state-of-health server. One advantage to this software architecture is that all software servers can be accessed and altered independently without stopping the entire control system providing flexibility during development and operation.

Automation is achieved using a state-machine based control program that implements the process necessary to run the system. The state-machine provides a well-defined process flow with each state defining one step in the overall control process. The state-machine framework is highly flexible. During development, states can easily be modified, added, or removed as needed.

## OBJECTIVES

The main concept of this effort was to develop both a client/server based software control architecture and state-machine based software automation scheme that could operate under the Linux operating system. The client/server based architecture stems from a previous design that ran under a Real-Time Operating System (RTOS) called QNX (Hubbard and McKinnon, 1997). QNX has its own provisions for client/server message-passing built into the OS. An equivalent capability had to be coded from scratch for the Linux OS.

A simple, easily extensible state-machine architecture that permits execution of a series of distinct states in a well-defined order was also developed. This state-machine architecture is used to describe and implement the automated control processes that drive a number of analysis systems. These automation techniques are the software control foundation for complex Xenon systems.

## RESEARCH ACCOMPLISHED

### Background

QNX, a real-time operating system designed for system control applications, is based on a *microkernel* architecture. This means that, internally, the operating system is implemented as a series of narrowly focused processes that communicate with each other by passing messages back and forth through the operating system kernel. Each process performs a specific function and is independent from other processes running on the system. Under QNX, the message-passing ability is a native function of the operating system.

The message-passing model for implementing operating systems also proves to be a versatile, powerful, and robust, model for implementing automated system control applications, so it was adopted as the basis for our QNX-based (and later, Linux-based) control systems. Hardware control processes written under QNX were very specific to a single task. Like the QNX operating system itself, these applications ran concurrently, communicated with each other via message-passing, and, when taken together, make up a complete software control system.

Over the past several years, the open source Linux operating system has grown into a solid operating platform with an extensive suite of development tools as well as a large amount of support from the open source software community as well as industry. Linux is also free, whereas the price of a QNX development seat is high and increases every year. This makes Linux an attractive alternative for controlling complex systems. Unlike QNX, Linux is not a microkernel operating system, and it lacks the powerful built-in message-passing mechanisms QNX has for inter-process communication (IPC). However, it does provide the necessary building blocks to allow a developer to create a complete message-passing IPC solution, and these were used to do just that. Although different than the QNX message-passing architecture, the developed solution retains a number of the key features the QNX architecture provides. The Linux based solution has proven to be as versatile, extensible, and robust as the QNX solution it replaces.

### Client/Server Architecture

Under the client/server message-passing architecture, a server is any independent process (program) running on the system that can accept requests from other processes, execute those requests, and return the result (if any) to the requestor. A client is any independent process (program) that requests services of another process. Figure 1 illustrates the basic concept. Programs can be pure clients (they can send request messages, but they never accept requests from other processes), or they can be pure servers (they process client requests, but never generate client requests of their own). However, in the most common case, processes that make up a control system perform both roles. That is to say, they can accept and process request messages from other processes (clients), but they may also send request messages to other servers as well. On real-world systems, pure clients do exist. Examples of such include a graphical user interface (GUI) application or a text-based status reporting or logging application. Pure servers do also exist, but are somewhat less common.

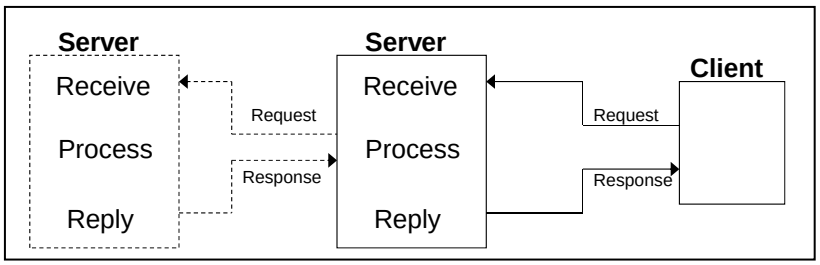


**Figure 1. The client/server architecture design.**

Traditional control automation software is implemented by a single (monolithic) program that implements all aspects of the control system. This method can be effective in some cases, but it also has many drawbacks including a lack of modularity, an application-wide shared memory space, no possibility for distributed execution, increased complexity, and the more difficult maintainability that goes with that.

The client/server model addresses many of these shortcomings. Clients and servers are narrowly focused, which makes them smaller, easier to develop, easier to debug, and easier to maintain. As a side effect, such narrowly focused applications are also very modular. They can frequently be reused on future control systems without any modifications. Experience has shown that the degree of server reuse between projects is often as high as 80 or 90 percent, meaning that complete control systems (including GUI) can be put together in a matter of hours or days whereas a custom monolithic solution may take many months of design, development and debugging. Extensive server re-use means much of system's control code has already been thoroughly tested and debugged on previous systems which further reduces the software development effort.

In addition, the client/server message-passing architecture developed for Linux operates over reliable transmission control protocol (TCP) network connections. As a result, control solutions may run all the server applications on a single computer, or they may distribute the servers over multiple computers. If a control application is distributed, the computers can be in different rooms, different buildings, or different parts of the globe. No matter what the case, the code to support each implementation is exactly the same. In general, clients and servers don't know (and don't care) if they're all running on the same computer or not.

Also, the client/server architecture's inherent modularity means that individual clients or servers can be stopped, replaced and restarted on a running system, often with little or no disruption. Along these same lines, new clients (data loggers, status reporters, alert monitors, etc.), unanticipated during the original system development, can easily be added to a system later on without the need to modify *any* of the existing system's software. These add-on clients can be started and stopped independently of the rest of the clients or servers on the system.

Finally, the client/server architecture has certain attributes that improve robustness of a system. One example of this has to do with memory sharing. In a monolithic implementation (one program does everything), all parts of the code share common memory. A memory corruption bug (indexing off the end of an array, or writing through an uninitialized pointer say) in one part of the system can potentially corrupt memory in *any* other part of the program. Under the client/server model, as implemented on modern multi-tasking operating systems like Linux, each client and each server runs in its own, protected memory space, and *cannot* corrupt the memory of other applications running on the same system. This memory protection is hardware-enforced by the central processing units' (CPUs') memory processing unit, and cannot be defeated by a poorly (or maliciously) written client or server. Limiting memory corruption problems to a single process substantially reduces the amount of time it takes to track them down and fix them.

There are advantages this architecture also provides in the world of research and development (R&D). Server modularity promotes code reuse which significantly speeds up development. Once a server is written for a specific instrument, for example, it is available to provide the communication mechanism for that same instrument on a different system. This also inherently provides a lot of code testing, so new projects and systems are able to use pre-developed well-tested software. Clients and servers can be started, stopped, and restarted while the overall

system continues to run. This is beneficial during new server design or update as single servers can be tested, stopped, fixed, and restarted without bringing down the entire control system.

**Linux-Specific Implementation**

The clients and servers were written for Linux using the C++ programming language. Since client/server message-passing is the centerpiece of the architecture, its implementation was given a lot of consideration. As mentioned previously, Linux provides many different options for inter-process communication (shared memory, named pipes, network streams, etc.), each with its own set of advantages and disadvantages. In the end, the Internet standard TCP was chosen as the message transport layer, because it offered certain attractive features. TCP has been around for many years and is a well-tested and reliable protocol. It inherently supports message transport between multiple computers. This was a useful characteristic of the QNX message-passing mechanism as well, and an important one to maintain. Also, servers need to be able to support connections from multiple clients simultaneously. This is an inherent feature of TCP.

Under Linux, all servers and all client interfaces are implemented as C++ classes. Servers are derived from a common server base class and client interfaces are derived from a common client interface base class. These underlying base classes implement the specifics of the message-passing framework. The client interface base class provides the code to establish TCP connections with servers, send requests to the servers, interpret response messages, and provide automatic lossless re-connection in cases where a TCP connection between a client and server has been broken (the server has been restarted or the network link between client and server has been temporarily disrupted for example). The server base class provides facilities to advertise the existence of the server to potential clients, handle incoming client connection requests, receive and serialize incoming client request messages and send responses. The two base classes also implement a small set of client/server messages for capabilities (such as “ping” and “shutdown”) that all clients and servers share.

**Server Heirarchy**

Client/server message-passing often results in *request message chains* where a client sends a request to a server, and, as part of handling that request, the server may send a client request of its own to another server (which may itself send a client request to yet another server, and so on). One design rule that must be rigorously enforced when designing a client/server based control system is that request message chains must *never* be allowed to form cycles. So a responding server *cannot* send a request of its own (either directly or indirectly) back to a server that is part of the request chain. This action causes a deadlock condition that will cause all clients and servers in the cycle to become unresponsive. One side effect of this design rule is that the clients and servers that comprise the complete control system naturally arrange themselves into a hierarchy. Servers lower in the hierarchy tend to be hardware controllers (user-mode device drivers responsible for one or more pieces of hardware of a given type – all digital I/O signals for example). More abstract clients and servers (the overall process control server or the system’s GUI for example) fall higher up in the hierarchy. Figure 2 shows a typical set of servers that may be deployed for a generic control system and its resulting hierarchy. Note that, in this diagram, client message requests *always* pass from higher levels to lower levels, and responses always come from lower levels to higher levels. This is a direct result of the design rule that forbids request message cycles.

**State-machine Based Control**

In general, the top-level *server* application in the server hierarchy is always the master control application (or *control server*). The control server is responsible for orchestrating all actions that occur on the system in a sequence that results in automated system processing. To facilitate this sequencing of actions, a simple extensible state-machine architecture was developed that is used internally by the control server.
A state-machine is a collection of well-defined *states* along with a well-defined set of *transitions* that connect them. Transitions define how control can move from one state to another and under which conditions it is permissible to take a particular path. Transitions conditions are carefully designed such that at most, only one transition out of a state will be valid at any one time.

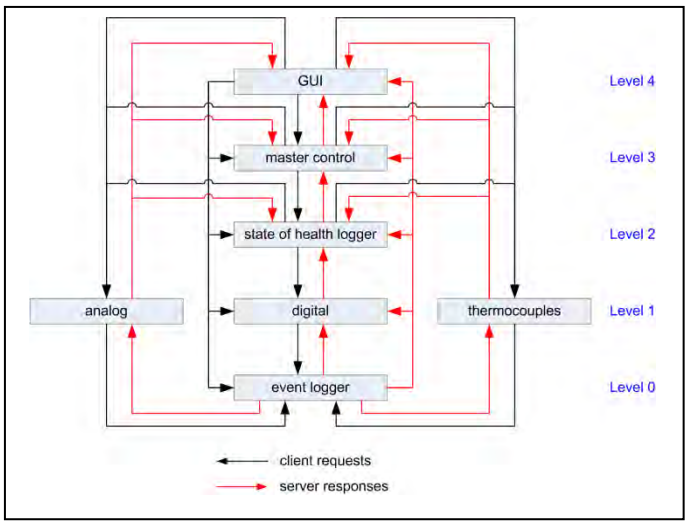


**Figure 2. Typical server set and resulting hierarchy.**

Each state in the state-machine is implemented in four parts (most of which are optional). They are: actions that must be performed when initially entering a state, actions that must be performed when exiting a state, transition logic (provides the conditions under which each transition may be followed), and internal processing logic (actions that are performed continuously while inside the state). The control server's state-machine provides the logical process flow for the system. Figure 3 above illustrates a process flow diagram for a simple state-machine that defines a marble sorting process. Table 1 below defines the actions assigned to each state. Clearly this is a trivial example designed simply to show what a process control state-machine might look like. On real-world systems, the control state-machine is typically somewhat more complicated, containing between 20 and 30 states and multiple entry points.

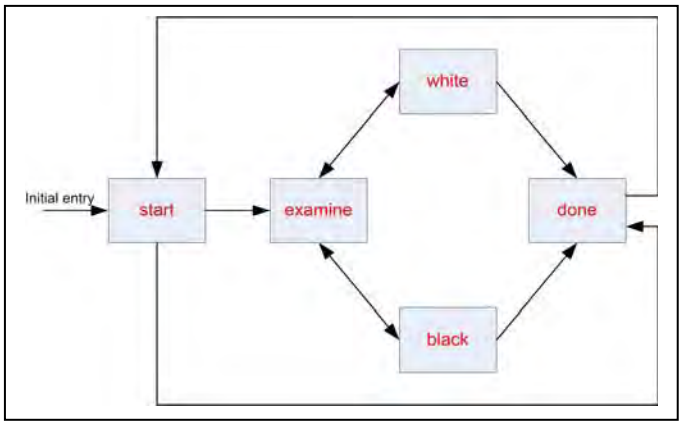


**Figure 3. Basic process flow diagram for a simple state-machine.**

**Table 1. The actions assigned to each state in a state-machine.**

| State | Entry Actions | Exit Actions | Internal Processing | Transition Logic |
|---|---|---|---|---|
| **start** | Turn on the "system is running" light. | None | None | If there are more marbles to be processed, jump to **examine**, otherwise jump to **done**. |
| **examine** | Get a new marble. | None | None | If the marble is white, jump to **white**, else jump to **black**. |
| **white** | Put the marble in the white bucket. | None | None | If there are more marbles to be processed, jump to **examine**, else jump to **done**. |
| **black** | Put the marble in the black bucket. | None | None | If there are more marbles to be processed, jump to **examine**, else jump to **done**. |
| **done** | Turn off the "system is running" light. | None | None | If the operator presses the start button, jump to **start**. |

## CONCLUSIONS AND RECOMMENDATIONS

The client/server architecture provides a modular approach to software control for systems. This has several overall advantages including ones specific to R&D and system operations. This approach provides a robust infrastructure for developing software for control systems.

The state-machine based control naturally provides a simple way to design, implement and describe the flow of a control system. The flexibility of this design allows states to be easily added, removed, and altered, make it a useful tool for research and development.

These system automation tools will be used for current and future Xenon processing systems.

## REFERENCES

Hubbard, C., and D. McKinnon (1997). Client-Server Architecture Applied to System Automation, *IEEE Trans. on Nuclear Science* 44: 3, 783–787.