AN ERLANG-BASED FRONT END FRAMEWORK FOR ACCELERATOR CONTROLS

Dennis J. Nicklaus, Charlie Briegel, Jerry Firebaugh, Charlie King, Richard Neswold, Ron Rechenmacher, Jianming You. Fermilab, Batavia, IL 60510, U.S.A.

Abstract

We have developed a new front-end framework for Fermilab's Acnet control system in Erlang [1]. Erlang is a functional programming language developed for real-time telecommunications applications. The primary task of the front-end software is to connect the control system with drivers collecting data from individual field bus devices. Erlang's concurrency and message passing support have proven well-suited for managing large numbers of independent Acnet client requests for front-end data. Other Erlang features which make it particularly well-suited for a front-end framework include fault-tolerance with process monitoring and restarting, real-time response, and the ability to change code in running systems. Erlang's interactive shell and dynamic typing make writing and running unit tests an easy part of the development process. Erlang includes mechanisms for distributing applications which we will use for deploying our framework to multiple front-ends, along with a configured set of device drivers. We've developed Erlang code to use Fermilab's TCLK event distribution clock and Erlang's interface to C/C++ allows hardware-specific driver access.

INTRODUCTION

Our front-end computers are responsible for acquiring control system signals from the hardware or field bus and making this data available to the Fermilab control system. The front-end framework is a software architecture that is deployed to provide common central functions to serve the variety of devices or bus systems that we acquire data from. The primary requirements of this front-end framework are:

- Communicate readings, settings, etc. via the Acnet protocol
- Enable a mapping between the a device's name in the control system and the hardware channel.
- Manage the connections between control system requests and the data channel.
- Check for and report device alarms.
- Respond to the timing system to ensure prompt collection
- High degree of reliability.

We began discussing developing a new framework as an alternate path forward, replacing our vxWorks-hosted, C-language MOOC framework, with a stated goal of running under the Linux operating system. Initially, we were expecting to develop the new framework in C++, but the Erlang language caught our attention and we soon realized it would be a great candidate for our project.

ABOUT ERLANG

Erlang is a functional programming language originally developed for telephony applications. It was built to support soft real-time systems with a high degree of reliability and fault-tolerance. Erlang is designed to be scalable to very large distributed systems.

Functional Programming

Functional programming emphasizes the application of functions to get results, as opposed to the more familiar imperative paradigm which emphasizes changing state and side-effects of subroutines. The elimination of side-effects in functional programming makes predicting and testing function results more straightforward.

Properties of Erlang include dynamic typing, single assignment, and extensive support for pattern matching in assignment. The initial hurdle that new Erlang developers must jump is figuring out how to program when you can't change the value of a variable once it has been assigned. As you would expect from a functional language, there is considerable support for list creation and processing, as well as structured concepts built on lists.

Erlang has excellent support for concurrent processes and for message passing between processes. Spawning a new process is a relatively inexpensive operation in Erlang. There is a built-in process supervisor behaviour model, where the supervising process gets notified of the termination of the subordinate and can act to restart it.

WE ADOPT ERLANG

Once we started looking at Erlang, we realized it was very well-suited for our front-end framework. Reliability of front-ends is a key concern, and Erlang is designed from the ground-up with reliability in mind. It supports the soft real-time that we need and is available on a variety of platforms, including linux that we were specifically targeting. High level data structures, such as the Erlang dict offer a painless way to implement mappings between database devices and specific controls code. Easy availability of Erlang processes, and high level language support for message passing make Erlang a natural fit for handling all the various Acnet data requests at a large variety of frequencies. Standardized process behaviours, such as supervisor or generic server, have proven useful in supporting the modularity of our design. Utilizing multiple processes and the supervisor behaviour allows us to build in overall reliability even if the implementer of a particular device's driver code has made mistakes which might cause that driver code to crash. Erlang provides an interface to C/C++ that allows us to

Operated by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the United States Department of Energy.

use C++ when we are required to read/write hardware registers that aren't accessible to Erlang.

FRAMEWORK DETAILS

One can divide our Erlang framework into several relatively independent modules: Acnet Protocol Handler, Sync, Data Acquisition Core, and Device Drivers.

Acnet Protocol Handler

We have a library of routines for connecting with the acnetd daemon which transports Acnet traffic in and out of the Erlang framework. In order to hide the details of the Acnet binary network protocols, we've developed sets of marshalling and unmarshalling routines that convert between the Acnet binary packet and Erlang records (structured lists). With a separate set of these functions for each particular Acnet message (e.g. read, set, post alarm, get plot data,...) used in conjunction with the Acnet library functions for sending or receiving messages, the rest of the Erlang code is freed from any worries about the Acnet packet protocols.

Sync

Our Erlang Sync library is our general purpose triggering system for data acquisition or other periodic tasks in Erlang. A unified interface handles read requests for periodic updates, updates on our hardware clock events, or other requests, such as on Acnet state changes. For instance, if the Erlang front-end receives a request to read data at 1 Hz, the framework registers its generic ''read data callback'' function with the Sync library and thus gets notification to run at the desired period.

Data Acquisition Core

The data acquisition tasks are the main function of the front-end. At the core here is a server process waiting on new read or set requests to arrive via Acnet. This server keeps a local mapping of the local device drivers and the database identifiers which will access those drivers. At system startup, the drivers in use are registered with the data acquisition server. Currently this is accomplished with a configuration file, one for each front-end to indicate the device types it owns, but we envision this information one day being stored in the global device database and sent to the front-end from the database at startup time.

When the data acquisition server receives a new reading request, it will generally spawn a separate process to handle that request at its desired update rate, thus keeping the central server free to receive new messages and insulating it from any buggy behaviour in the device drivers.

Alarm limit checking is also done with processing similar to a simple read, but with the obvious added limit checking added on.

Device Drivers

We refer to the code which contains the details of reading or setting any particular device as its ''device driver''. These aren't necessarily traditional linux device drivers, although that may be a part of our Erlang device driver. There is a standard Erlang programming interface that each driver conforms to. Each device driver in an Erlang front-end is registered with its local data acquisition server in order to connect to Acnet requests. This interface sends the device driver routine the details of the access request, a container of information shared throughout that device driver, and the triggering event specification.

Some controls tasks require accessing hardware which isn't reachable from Erlang code. For these tasks, we've also developed a standard method of calling from Erlang to C++, using standard Erlang-C interfaces. Our C++ programming interface lets the C++ compiler do much simple type and error checking to ensure reasonable parameters are passed to the C++ code.

PROGRAMMING IN ERLANG

Everyone on our development team is used to programming in imperative languages such as C/C++ and Java. In addition to mastering the new syntax, functional programming requires a fundamental mind-shift by the programmer. This shift is somewhat similar to changes one must make when going from a serial, procedural C program to an event-driven Java graphical user interface. The functional tools provided by Erlang also allow one to consider different solutions to a programming problem.

Two paradigms used extensively in Erlang include message passing between processes and tail recursion. Erlang's message passing is built into the language, and passing indicator atoms or message data can be done without much thought by the programmer. Since Erlang passes copies of the data, the programmer doesn't need to worry about issues like semaphore locks around shared data. Because Erlang variables are single assignment, many tasks which might be done by iteration in C are done by recursion in Erlang. The run-time system is highly optimized for tail recursion, where a function calls itself recursively and the last thing the function does is make the recursive call.

Many features of the Erlang language and run-time system only become useful as one gains some expertise with the language, for instance, the generic server (gen_server) behaviour. A novice Erlang programmer will probably spend some time writing some simple message receive loop functions. As the functionality of that loop expands, the utility of the predefined gen_server package becomes more apparent.

Functional languages by their nature encourage unit testing. Since pure functions don't have side effects, effective test routines can prove that your code works. The Erlang development system provides tools that explicitly tell you which lines of you code are tested by your unit testing functions. These coverage maps obviously help demonstrate well-tested code. The interactive Erlang shell also speeds testing because you can construct test vectors and interactively call your functions.

With reasonably modern 1-2 GHz processors running linux, Erlang's performance has been sufficient for us The Fermilab linac runs at 15Hz, so 15Hz response has been our baseline for testing Erlang system response. Using Erlang and its command-line shell, it is easy to create stressful load tests of the system, for example, setting up requests for hundreds of device readings at once. Erlang provides run-time profiling tools that help point out performance bottlenecks and show how a programming or data structure choice can influence the run-time performance.

We have not yet made use of the "hot-swap" feature that lets one upgrade a software module of a running Erlang system, but we look forward to taking advantage of this in the future.

SUMMARY

We have developed a new front-end data acquisition and controls framework in Erlang. Adapting to the functional programming paradigm has been well-worth the effort and some early deployments of Erlang front ends include control commercial motor controllers over UDP and monitoring the status of the near detector of Fermilab's Nova experiment. A few other example device drivers have also been written. We've found Erlang very suitable for this type of application and are looking at implementing other control system infrastructure in Erlang.

REFERENCES

[1] The Erlang Programming Language website at www.erlang.org.