CHANNEL ACCESS IN ERLANG

Dennis J. Nicklaus. Fermilab, Batavia, IL 60510, USA

Abstract

We have developed an Erlang language implementation of the Channel Access protocol. Included are low-level functions for encoding and decoding Channel Access protocol network packets as well as higher level functions for monitoring or setting EPICS process variables. This provides access to EPICS process variables for the Fermilab Acnet control system via our Erlang-based front-end architecture without having to interface to C/C++ programs and libraries. Erlang is a functional programming language originally developed for real-time telecommunications applications. Its network programming features and list management functions make it particularly well-suited for the task of managing multiple Channel Access circuits and PV monitors.

INTRODUCTION

The Fermilab main accelerator chain, and most other on-site beamlines such as NML/ASTA are controlled with the Fermilab Acnet control system. The transport protocol used by this system is also called Acnet. However, we have occasionally had the need to integrate sub-systems which use the EPICS control system, such as for individual instrumentation packages, and we envision the possibility of doing this more in the future to accommodate collaborators who have developed in EPICS. Furthermore, for the NOνA neutrino experiment, the detector control system is a mix of EPICS and Acnet.

EPICS uses the Channel Access (CA) protocol [1], primarily a TCP/IP socket based protocol, but including some UDP parts, such as for PV search beacons

We recently introduced ACSys/FE [2], a new Acnet front-end framework, written in the Erlang programming language. This front-end framework supports a variety of network-based device drivers, and we needed it to also support the EPICS CA protocol.

DEVELOPMENT

There are C-language libraries implementing the CA protocol, and these have been widely used. ACSys/FE has a standard mechanism to incorporate C/C++ code by communicating with a C++ -based executable using Erlang's standard interface to external C/C++ code. Our first implementation which enabled reading and setting EPICS PVs through Acnet used this standard C-language CA library. While this was functional, the extra step of converting from Erlang to C introduces inefficiencies and makes it more difficult to make everything from the C side available in the Erlang device drivers. In addition, using the C libraries also makes moving the front-end across other platforms slightly more troublesome. The CA C code is quite portable, but can require recompiling when moving to slightly different platforms, whereas compiled Erlang byte code is fully portable without recompiling on the target. For the above reasons, we implemented the CA protocol in pure Erlang, with no C code involved, and starting from the raw protocol documentation.

It should be noted that we only implement CA client functionality in Erlang since we have not yet needed to make our own CA server (IOC).

BACKGROUND

Channel Access

The CA protocol uses a combination of UDP and TCP network protocols. UDP is used for beacons, such as the search command which queries the network for the server hosting a particular PV, or for CA servers to announce that they are up. A TCP socket is established between the client and a particular server for commands directly to that server. CA uses the term "virtual circuit" to describe the one TCP connection that is made between a client and server for all the PVs from that server. All CA messages have a 16 byte header followed by a variable-sized payload. The first 16 bits of the packet header is a command indicator and the second 16 bits of the header indicates the payload size.

Repetitive CA readings are through *monitors*. Monitors are set up as a channel to the IOC server and the IOC sends updated readings over this channel as they become available.

Erlang Features

Erlang provides high level functions for network programming such as *open* and *send* for UDP and *connect*, *send*, and *close* for TCP. We've found that network programming with Erlang is much easier than in C because more of the details are hidden from the casual programmer.

A key feature of Erlang is its innate support of multiple processes and message passing between them. A common process implementation is the server loop, where the process is in an infinite recursive loop, always waiting on an incoming message, responding to that message, then calling itself to return to waiting on the next message. The server loop may have a state variable which is passed along with each recursive call of itself, possibly with modification as a result of the message response. This usage is so common in Erlang that the language has a standard *gen_server* behaviour which makes it easier for a programmer to implement it.

IMPLEMENTATION

Our Erlang implementation of CA is built around four levels of these server loops: the device driver, a manager of virtual circuits, the virtual circuit server, and a small loop for handling the TCP socket.

ISBN 978-3-95450-139-7

Erlang CA Device Driver

The device driver is the ACSys framework's boilerplate technique for connecting a particular device to the Acnet control system. The driver plugs into the framework, and provides reading and setting functions for accessing the assorted attributes of a device. By implementing a framework device driver, the programmer doesn't need to know any details of the Acnet binary protocols. The CA device driver retrieves and manages information such as the mapping from Acnet device to Epics PV name.

Virtual Circuit Manager

This server is a middleman between the Acnet device driver and the CA communications of the virtual circuits. It cache's IOC servers found for PVs and keeps track of mappings between IOC servers (IP addresses) and the Erlang virtual circuit process managing that server and the mapping between individual PV read requests and the circuit process. It also manages new PV search requests and either makes the connection to a responding server or declares the PV unfound after a timeout.

The Virtual Circuit

The core of the CA communications is through the virtual circuit process. This process manages the simple process which reads from the TCP socket. It implements the requests from the device driver to start CA monitors or make settings.

The virtual circuit process has to maintain an extensive state in order to remember which monitors are underway and which monitor is connected to which Acnet reading request process. (In the ACSys framework, an individual Acnet reading request can contain an arbitrary number of devices to be read all at the same frequency. A separate Erlang process is spawned off to handle each request.) Additional state machine information is required when a new monitor is started because establishing a monitor channel requires more than just one command-reply message pair. Settings similarly require opening a new channel to the server, but then closing it once the setting is complete.

The virtual circuit server has to manage all the subscriptions (monitors) over one particular CA virtual circuit. It also has to retain sufficient information so that if the TCP socket closes (e.g. if the IOC is rebooted), it can notify all of the current reading requests of this fact, as well as notifying the virtual circuit manager.

TCP Socket Handler

This is a very simple looping function which constantly waits on new messages from the TCP socket connected to the IOC. It simply sends a message to its controlling virtual circuit process every time it reads a message from the socket. If it detects the socket closing, this loop simply exits. That exit is noticed by the controlling virtual circuit and a new TCP connection can be made. The details of Erlang message passing and the TCP communications made it simpler to have this separate process perform the socket reading rather than the controlling virtual circuit.

Message Binary Encoding

At the very bottom level of our CA implementation are a set of functions which encode specific CA messages into Erlang binaries. We have implemented parameterized functions like *encode_clear_channel, encode_event_add, encode_echo*, … for the messages at the lowest level of the CA protocol. These functions produce the binary packets which are sent over the CA communication sockets.

FUTURE PLANS AND SHARING

Much of this Erlang implementation of CA is tied to our ACSys framework. However, many useful pieces are not and can be used or tested without any Acnet connection or knowledge. In particular, lower level functions, such as those which encode specific CA messages into Erlang binaries ready to transmit to the IOC would be useful for other facilities. Currently, we only have implemented certain aspects of CA, namely monitor values and settings. We have not yet had a need to monitor EPICS alarms via CA, but it would be easily added. Our API and documentation would need improvement to be more globally useful, and we are open to suggestions as to what is needed. We only implement CA as a client library, and have no plan to create an Erlang CA server.

SUMMARY

We have implemented significant parts of the Channel Access protocol in pure Erlang. Our software provides client access to CA servers and makes EPICS PVs available to the Acnet control system. We have been using this implementation operationally at Fermilab and at its NOνA Far Detector in northern Minnesota and are pleased with its performance.

REFERENCES

[1] Žagar, Klemen et al, *Channel Access Protocol Specification*. Ljubljana, Slovenia (2003-2008). http://epics.cosylab.com/cosyjava/JCA-Common/Documentation/CAproto.html

[2] Nicklaus, D. et al, "An Erlang-Based Front End Framework for Accelerator Controls," The 13th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALPECS), Grenoble, France (2011).

ISBN 978-3-95450-139-7