NATIVE JAVA IMPLEMENTATION OF CHANNEL ACCESS FOR EPICS

M. Sekoranja
Cosylab, Ljubljana, Slovenia

ABSTRACT

Channel Access for Java (CAJ) [1] is a 100% pure Java CA client library. It was written as a result of detailed analysis of existing CA library to provide better stability opposed to the current JCA JNI implementation, while keeping the JCA API [2] for backward compatibility. This was achieved by minimizing code complexity by clean OO design and profound testing during the whole development cycle. Following the latest design patterns and technology, CAJ is also designed for performance. Some performance measurements will be shown.

WHAT IS CAJ?

EPICS (Experimental Physics and Industrial Control System) uses CA (Channel Access) protocol [3] as a standardized communication path to a field(s) within a record (representation of a single value, also called process variable) in any IOC (Input/Output Controller) database. EPICS base provides C++ implementation of CA library. Due to popularity of Java programming language support for it was implemented using Java Native Interface (JNI) technology which allows calls to a native C++ CA library. JNI solution implements/plugs into a set of interfaces called Java Channel Access v2 (JCA2). However, JNI solution was quite unstable under heavy load and it is not easily distributable comparing to platform-independed Java code – native C++ CA library has to be build for each (version of) operating system. This led to implementation of Channel Access in Java (CAJ) - a CA client library completely written in Java. CAJ was written as a result of detailed analysis of existing C++ CA library to provide better stability and performance opposed to the current JCA JNI implementation. Since it was written from scratch CAJ code is clean, follows OO design and uses lots of design patterns. It also implements/plugs into JCA2 interfaces which makes migration from JNI to CAJ transparent.

BASIC CA PROTOCOL DESCRIPTION

The goal of CA is to provide remote access to records and fields managed by IOC, including search and discovery of hosts and minimal flow control. Protocol itself is designed to provide minimal overhead and maximize network throughput for transfer of large number of small data packets. Additionally, implementation overhead of the protocol can be kept very small, to allow operation with limited resources.

All commands and events in CA are encapsulated in predefined messages, which can be sent in one of three forms:

- As a ***beacon***, which requires no confirmation. Used for host discovery and keep-alive notification.
- As a ***request/response pair***. Most commands use this method.
- As a ***subscriber notification***, where the client registers with the host and receives updates. Event notification uses this method to report value changes.

Communication between server and client is performed by sending command messages over UDP and TCP. Client will use UDP to search for hosts and PVs (process variables), server will use them to notify its startup and shutdown. Once client requests a specific PV (by specifying its name), UDP message will be broadcast to either a subnet or a list of predefined addresses, and the server which hosts requested PV will respond.

Data exchange between client and server is performed over TCP. After locating the PV, the client will establish a TCP connection to the server. If more that one PV is found to be on the same server, client will use existing TCP connection. Reusable TCP connection between client and server is called Virtual circuit. Channel Access protocol is designed to minimize resources used on both client and server. Virtual circuits minimize number of TCP connections used between clients and servers. Each client will have exactly one active and open TCP connection to an individual server, regardless of how many channels it accesses over it. This helps to ensure that servers do not get overwhelmed by too many connections.

Once a virtual circuit is established or already available, channels can be created to PVs.

DESIGN AND IMPLEMENTATION

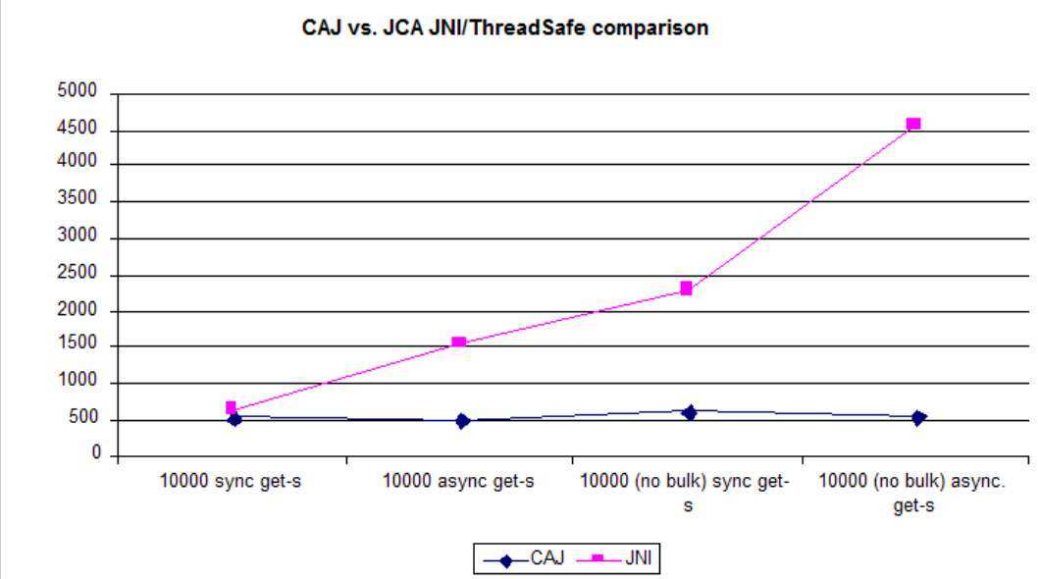
CAJ was implemented using the latest concurrent, network communication design patterns used to implement efficient event demultiplexing, minimize context switching and maximize CPU cache affinity (Leader/Followers design pattern [4]). Asynchronous I/O is used (Java NIO package) [5] to be able to handle large number of servers using as minimal system resources as possible. New Linux 2.6 kernel 'epoll-based selector' is also supported. Due to OO design light CAJ version is possible (one communication thread), convenient for light CA clients (handhelds).

Since CA uses UDP to search for channels, congestion control needs to be implemented to avoid congestion (UDP does not provide it, but TCP does). CAJ has implemented ‘TCP Reno’-like UDP congestion control.

Profound testing was enforced during the whole development cycle (~ 90% of code coverage!) - only a few bugs were found after first public release.

PERFORMANCE TESTS

Some tests where performed to compare CAJ with the JNI implementation. They showed much better performance of CAJ.



Tests where performed with the client on the same host as server using Pentium IV 1.6GHz, 1GB RAM, Red Hat 9. Note that these are only synthetic performance tests and do not reflect performance in practice!

ACKNOWLEDGEMENTS

I would like to thank DLS (M. Heron) and DESY (M. Clausen) for founding the development of CAJ.

REFERENCES

[1] http://caj.cosylab.com/
[2] http://jca.cosylab.com/
[3] http://epics.cosylab.com/cosyjava/JCA-Common/Documentation/CAproto.html
[4] http://deuce.doc.wustl.edu/doc/pspdfs/lf.pdf
[5] http://java.sun.com/j2se/1.4.2/docs/guide/nio/