Keck Adaptive Optics Note 678

Keck Next Generation Adaptive Optics External (CA/KTL) Interfacing

Jimmy Johnson, Doug Morrison, Erik Johansson

Draft: June 9, 2009

Author List

Document Change

1 Introduction

This document describes the client and server interfacing capabilities between KCSF and EPICS and/or any application using the Keck Task Library (KTL).

2 Related Documents

Need to dig up CAJ and JCA documents See www.cosylab.com

3 Document Conventions

When using code fragments as examples, we will use the non-proportionally spaced Courier New font:

```
class foobar extends foo {
      int foo1;
      double foo2;
}
```

4 Overview

KCSF has both the ability to act as a CA server (and KTL server via CAKE) and a KTL client. Server support allows for outside systems to act as a master pushing and pulling data to and from KCSF based on their timing needs. Client support allows KCSF applications to be the driver where all read and/or writes are instigated from the KCSF side.

5 CA Server

The CA Server will be implemented using the Channel Access Java (CAJ) and Java Channel Access Server (JCAS) libraries provided to the EPICS community by Cosylab. This is a 100% pure Java implementation requiring no JNI and so can work on any platform. The framework is simple. Java Channel Access Server (JCAS) is a pure Java implementation of a Channel Access Server. JCAS is an addition to the existing Channel Access in Java (CAJ) client library; both share common code and are packed in a same Java Archive (JAR) file.

Creating a server is very straightforward. ServerContext requires an implementation of the Server interface. This interface has two methods which must be implemented:

- `processVariableExistanceTest`
  - is called each time a CA client broadcasts a search for a particular process variable, identified by a name. If a positive answer is returned by the method, JCAS library will announce that it hosts that process variable.
- `processVariableAttach`
  - Once a CA client is knows a process variable exists it will most likely issue a request for channel creation. A channel is a connection between the server and client through which a single process variable is accessed. The

client never talks directly to a process variable, only through the channel. To create a new channel ServerContext will request a ProcessVariable instance by calling the processVariableAttach method. Then it will create a channel instance by calling the ProcessVariable createChannel() method.

Figure 1 Shows the sequence diagram for a client interaction with the CA server.

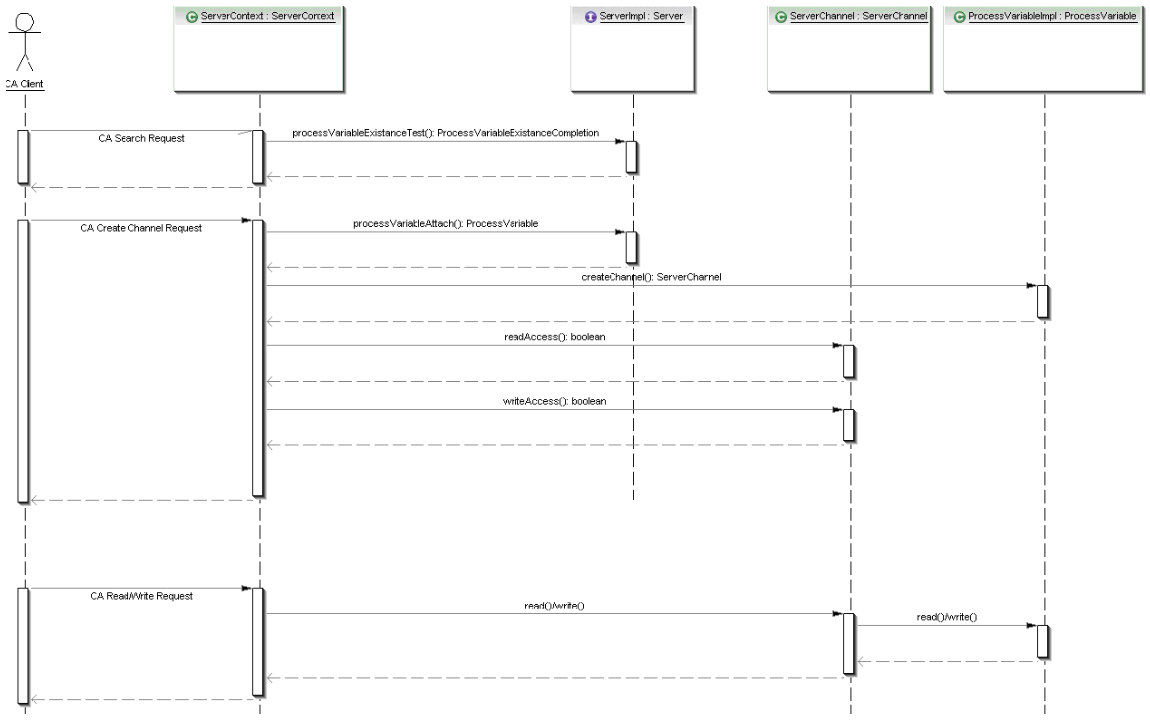

CA Client
ServerContext : ServerContext
ServerImpl : Server
ServerChannel : ServerChannel
ProcessVariableImpl : ProcessVariable
CA Search Request
processVariableExistanceTest(): ProcessVariableExistanceCompletion
CA Create Channel Request
processVariableAttach(): ProcessVariable
createChannel(): ServerChannel
readAccess(): boolean
writeAccess(): boolean
CA Read/Write Request
read()/write()
read()/write()

Figure 1 Sequence Diagram for CA Server

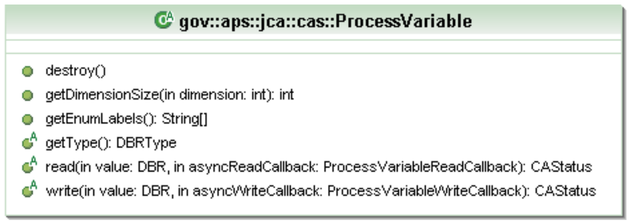
As can be seen from the following code there is very little required to get a basic CA server up and running. The code shows the use of a default implementation of the Server interface by the DefaultServerImpl class. All process variables registered by the DefaultServerImpl registerProcessVariable() method will be available to CA clients. The implementation of DefaultServerImpl processVariableExistanceTest() simply does a hash map existence test and DefaultServerImpl processVariableAttach() does a hash map value retrieval. It is conceiveable that the DefaultServerImpl could be used directly for KCSF.

```
/**
   * JCA server context.
   */
  private ServerContext context = null;

  /**
   * Initialize JCA context.
   * @throws CAException    thrown on any failure.
```

```
     */
    private void initialize() throws CAException {

            // Get the JCALibrary instance.
            JCALibrary jca = JCALibrary.getInstance();

            // Create server implmentation
            DefaultServerImpl server = new DefaultServerImpl();

            // Create a context with default configuration values.
            context = jca.createServerContext(

      JCALibrary.CHANNEL_ACCESS_SERVER_JAVA,
                                     Server);

            // Display basic information about the context.

      System.out.println(context.getVersion().getVersionString());
            context.printInfo(); System.out.println();

            // register process variables
            registerProcessVariables(server);
     }
```

5.1 *Process Variables*

`ProcessVariable` is an abstract class that should be extended that provides a default implementation that returns a `ServerChannel` instance. This implementation is a default implementation of a channel and grants all read and write rights to any user.



gov::aps::jca::cas::ProcessVariable

```
destroy()
getDimensionSize(in dimension: int): int
getEnumLabels(): String[]
getType(): DBRType
read(in value: DBR, in asyncReadCallback: ProcessVariableReadCallback): CAStatus
write(in value: DBR, in asyncWriteCallback: ProcessVariableWriteCallback): CAStatus
```

Figure 2 ProcessVariable class with methods that are important for a developer.

`The ProcessVariable` class has three abstract methods to be implemented:

- `getType()` – return PV native data type. Types used to read/write.
- `write()` – write value to PV. Given DBR type is basic native data type (e.g. `DBR_Double`).

- `read()` – read value from the PV. Given DBR type is always at least of DBR `TIME` type. If of `GR` or `CTRL` type, it depends on the PV to support it.

Methods `getDimensionSize()` and `getEnumLabels()` are most likely to be overridden. However, it is recommended that the default PV implementations located in `com.cosylab.epics.caj.cas.util` package be used. They provide instructions on how to implement your own process variables.

5.2 Monitors

Another nice aspect of the design is that monitors are automatically handled by the server framework. Once a process variable is registered and the necessary ProcessVariable read and write logic is implemented then the ability to have monitors applied to our data is automatic and addressed by the cosylab implementation.

6 CA Client Service

The CA client service allows KCSF components and applications to read, write and monitor channels using the channel access protocol. The implementation of the client service is based on Channel Access Java library.

The following methods are supported:

```
put (final String channelName, final byte value);
put (final String channelName, final int value);
put (final String channelName, final float value);
put (final String channelName, final double value);
put (final String channelName, final String value);
put (final String channelName, final short value);
put (final String channelName, final byte[] value);
put (final String channelName, final int[] value);
put (final String channelName, final float[] value);
put (final String channelName, final double[] value);
put (final String channelName, final String[] value);
put (final String channelName, final short[] value);

addMonitor (final String channelName, IMonitorCallback cb);
removeMonitor(final String channelName);

byte getByte(final String channelName);
int getInt(final String channelName);
float getFloat(final String channelName);
double getDouble(final String channelName);
String getString(final String channelName);
short getShort(final String channelName);
byte[] getByteArray(final String channelName);
int[] getIntArray(final String channelName);
```

```
float[] getFloatArray(final String channelName);
double[] getDoubleArray(final String channelName);
String[] getStringArray(final String channelName);
Short[] getShortArray(final String channelName);
```

The first time a new channel name is encountered a channel is created and kept in an internal cache. Calls to get, puts and monitors access the cached channels. Get (reads) and Put (writes) are all synchronous but asynchronous support can be easily added. All monitoring is asynchronous.

6.1 KCSF Specifics

The KCSF will provide a lightweight controller that is a CA Server. Like any component it can be load, unloaded, initialized, started and stopped. When initialized the component will instantiate an internal cache of all exported process variables. The cache will include their corresponding fully qualified KCSF attribute name and possible KCSF type.

Once the component is started the CA Server context will be set running `server.run(0)` and will continue to do so until the component is stopped at which time the ca server will be shutdown `serverContext.shutdown()`.

For KCSF the process variable names requested by the CA clients need to be translated to KCSF addresses. It is possible that there is a one to one mapping. To allow differences it is likely that through KCSF configuration there will be a mapping of CA process variable names to KCSF fully qualified attributes.

In the implementation the KCSF CA Server will create a process variable that is capable of mapping Attributes to CA types. Obviously PV writes will be issued through the component *set* command but it is unclear at this stage whether reads would be serviced through component *get* calls or through data subscriptions.

6.1.1 KTL Server Support

It is expected that by utilizing the current Channel Access Keyword (CAKE) layer which is a thin generic software layer providing KTL keyword access to EPICS systems (or more correctly systems that honor the CA protocol) the KCSF can be seamlessly exposed to existing KTL clients.

7 KTL Client support

It is unclear yet whether or not KCSF needs to provide a CA client access service or just a KTL client access service. If a CA client service is needed it can be easily provided through the CAJ libraries.

A KTL client library will be wrapped as a KCSF service. As such it can be created by the container and made available to all components that need it. Through the service a component can make a connection to a specific KTL service and issues reader and writes requests. There are a number of JAVA KTL implementation available currently at Keck and the detailed design phase will be used to chose the implementation best suited for KCSF.