DESIGN CONSIDERATIONS FOR INTEGRATION OF WIDE RANGE OF CONTROL SYSTEM COMMUNICATION PROTOCOLS IN CROSS-PLATFORM FRAMEWORK IN THE EXAMPLE OF EPICS AND TINE

Aleš Pucelj*, G. Tkacik, R. Šabjan,

J. Stefan Institute and Cosylab, Ljubljana, Slovenia

Abstract

Accelerator community offers wide range of low-level communication protocols, from minimal solutions such as EPICS channel access to CORBA object model. This makes it natural to attempt their integration into single framework. Such generalizations are inherently susceptible to performance handicaps arising from either inappropriate abstraction or platform dependence of individual protocols. We present our experience in this area along with solutions and open questions to the problem.

This article presents examples of integration of EPICS channel access and TINE protocol into Abeans framework using pluggable model. Focus is placed on the user/developer aspects of such implementation outlining the advantages of such approach. First, we show how simple access to the control system can be made in a general way. Comparison of different approaches shows the advantages of using a single interface and benefits for the developer. Next, we look at examples of actual plug implementation and the common points between the systems. Lastly, performance impact is analysed. This is especially important, since our implementation is Java based, which can under circumstances be slow and memory consuming. Several methods are presented that eliminate this overhead. This is confirmed with actual examples.

1 INTRODUCTION

Development of physics application should allow the physicist to focus on solving the problem without looking too much into the details of the control system. When dealing with a single control system on all levels, this is not a great problem. An example of such approach could be MEDM and EPICS. In a case where there is need to access several underlying implementations a framework of some sort is required. Design of such framework turns out to be of critical importance, since it should provide balance between simplicity of use, extensible control system support and adequate performance. An example of such framework could be CDEV. It establishes naming hierarchy and property relations using C and C++ structures with optional Java GUI.

Abeans framework presents one such solution. This article focuses on performance issues and shows that development of such framework is feasible without sacrificing performance.

2 REQUIREMENTS

A useful framework should provide the following features.

- Simple to use
- Extensible
- Minimal overhead

Simplicity of use implies that development of application using such framework should differ as little as possible from the control system specific approach. Since this is not possible by definition, compromise must be made in such manner, that it minimizes the time required to learn the use of framework with existing knowledge.

Framework must also allow further extensions to support different control systems. This includes not only particular communication protocol, but also structure of the control system and the variables therein. All of these features add to the overhead caused by the abstraction and effort should be made to minimize it.

3 ABEANS

Abeans is a framework that attempts to address all of the mentioned requirements. Implementation is Java based and is therefore fully object oriented.

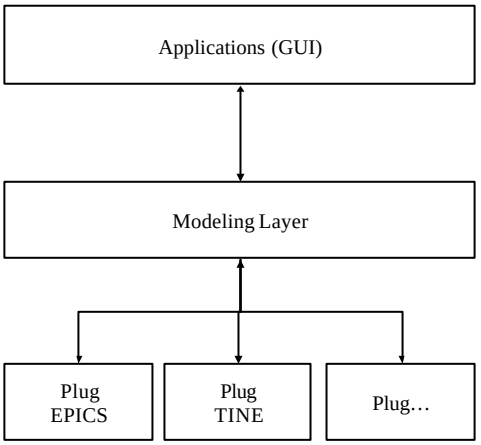

Figure 1: Abeans overview

Figure 1 shows simplified view of the Abeans design. GUI represents existing visual components (slider, number field, gauge, table), which issue requests to the modelling layer. Modelling layer is where actual requests for data from control system are made. Application requests are formulated and passed to the appropriate plug which performs the actual data transaction.

Application developer would either use the existing GUI components or connect to the modelling layer directly.

* ales.pucelj@cosylab.com

This design allows I/O abstraction on the modeling layer level. Current design uses request/response approach to handle communication. Modelling layer issues request objects and waits for one or more response objects. Each request contains information about the data source, type of operation (get/set/connect/…), requested data type, time stamp and other control data. Modelling layer is also responsible for queuing of these objects, error handling and synchronization to ensure thread safety.

Since multiple control systems can be accessed at the same time, it is important to establish unambiguous naming convention for individual I/O points. This is realized using URI (uniform resource identification) scheme, which is defined in RFC 2396 document. This approach proves sufficient to uniquely address any resource in a control system, where each I/O point is uniquely identified by a string or string representation. An example of the actual value of such URI would be: *abeans-EPICS:///PBEND_M_01_current?get*
or
abeans-TINE://ns.desy.de/DESY/BENDS/PBEND_M_01 /current?get.

One important aspect of this naming convention is that is used only internally to communicate between modeling layer and plugs. URIs are composed automatically, when requests are generated. This way, the actual information the application developer must provide does not differ from the information required by proprietary implementation in a particular control system. What must be considered is the overhead imposed by such name mapping. Each name must be first transformed into URI and later decoded by the plug, whenever requests are processed. As explained in plug implementation, this introduces much smaller overhead than expected.

4 PLUGS

Plugs are the most critical component of the entire framework, since they handle all of the communication. They must therefore support all requirements of the framework and perform them using existing implementations of communication libraries. Since they are expected to be a bottleneck in entire framework, especially under heavy load, care must be take in implementation. There are several issues that must be addressed. First is the resolution of name from the URI into appropriate name to be passed to communication protocol. Second is handling of connections. Abeans see the communication as state machine: data channel is first established, communication is performed and channel is closed. This may not necessary be the same as the underlying implementation.

4.1 EPICS Plug

EPICS support has been realized through JCA, java wrapper for native channel access libraries written in C. JCA has turned out to be reliable, the only problem arose from underlying implementation, which is single threaded. To allow access to the EPICS database using JCA the following functionality had to be added to the plug. When a request to access a channel is made, new process variable (PV) object must be made, which implements the actual communication. To avoid unnecessary object creation, PVs are stored in a hash table, where they are associated with channel names. If several requests are made to the same channel, same object will be reused. Also, since these objects are not created directly by application developer, this table is used to free the objects when they are no longer needed. Searching of hash table proves to be an efficient operation, capable of handling even large numbers of PVs without causing delays.

EPICS defines many primitive types, some of which cannot be explicitly represented in Abeans. When a certain variable is first accessed, it's type is obtained from the EPICS database and matched against supported types. If the type is unsupported, it is converted according to the following table:

Table : EPICS – Abeans type conversion

EPICS type	**Java type**
Double Float	Double
Int Byte Short Enum	Long
String	String

This conversion is made when passing values from the plug to the Abeans; when sending data through channel access, the original EPICS type is retained.

All errors that might occur during communication are handled by the plug and are reported to Abeans as Java exceptions.

Processing required by the plug is minimal and quite efficient. The only overhead is introduced by type conversion and wrapping of data types passed to JCA.

To test the performance, get and set requests were issued on a single machine hosting both Abeans client and EPICS server. Single machine was used to avoid network lag affecting the measurements. Average time required to complete a single operation was 1ms, where approximately half of that was spent performing the JCA call. This indicates, that despite measurable impact of the Abeans framework, it represents no significant slow down, since real-time response is not expected anyway. Also, in a normal network environment, overhead of Abeans would be insignificant. Interestingly, the time required to complete several requests with a single call (EPICS Channel access allows queuing of requests) did not take any more time (for a small number of requests). This leads to a conclusion, that 1ms per call can be considered worst-case scenario.

4.2 TINE Plug

Implementation of TINE plug has turned out to be much simpler. It uses java implementation of TINE protocol, which is already object oriented. There is also no need to perform any data conversion, since that is handled by java and its wrapping of types in objects. Slight disadvantage compared to EPICS and JCA is the parameter passing to TINE calls. These require creation of several new objects, which increases memory footprint of each call, but since this is the design of the library itself, plug implementation is no different than any use of the library would be. There is also no need for specific connection handling, since persistent connections are not directly implemented in the library, at least not at the API level. Performance of the plug is the same as with EPICS, where the actual overhead of each call is minimal. Memory footprint per each call is larger, but since this is not an issue of the plug it is not considered as overhead.

5 CONCLUSIONS

Abeans framework offers immense functionality, some already implemented and some supported through extension mechanisms, all of which is beyond the scope of this article. Issues presented here deal with some specifics of actual data transfer and communication and despite apparently complex implementation the overhead introduced by Abeans compared to native libraries is minimal, as long as slow-time control is sufficient. Since performance is not an issue, extensibility becomes the greatest advantage. Generalized access to underlying resources simplifies the handling of additional services and reduces code duplication that would be otherwise required for each implementation. Another advantage comes from the fact that a single solution works for all supported systems which reduces amount of testing required.

I must also thank DESY Hamburg and Oak Ridge National Laboratory for opportunity to work with these technologies and their support given during that time.

6 REFERENCES

[1] Jeffrey O. Hill, “EPICS Channel Access Reference Manual”, March 1995
[2] http://desyntwww.desy.de/tine
[3] J.Dovc et al., “New Abeans for TINE Java Control Application”, Proceedings ICALEPS’01, 2001.
[4] I.Verstovsek et.al., “The New Abeans and CosyBeans: Cutting Edge Application and User Interface Framework”, Proceedings PCaPAC’02, 2002
[4] Abeans and CosyBeans documentation, http://www.cosylab.com
[5] http://kgb.ijs.si/KGB/articles.html

7 APENDIX A

Code Examples:
These examples show the differences and similarities between different approaches.

EPICS	Abeans	TINE	Comment

status = ca_search(cname, &chan_id);
status = ca_pend_io(2.0);
	ApplicationContext ac;
	try {
	DoubleChannel dc =
	new DoubleChannel();
	dc.setRemoteInfo(ac.createRemoteInfo(name));
	dc.connect();
			Creation of channel

i = ca_field_type(chan_id)
status = ca_get(i,1, chan_id, value);
status = ca_pend_io(5.0);
	value = dc.getValue();	Object valueHolder =
		TINEType.getReadData(type, size);
		TDataType dout =
		TINEType.getReadTDataType(type, valueHolder);
		TDataType din = **new** TDataType();
		
		TLink link = **new** TLink(devName, propName, dout, din, TAccess.CA_READ);
		int status = link.execute(1000);
			Reading value

status = ca_put(i,1, chan_id, value); status = ca_pend_io(5.0);	dc.setValue(value); catch (RemoteException e) {…}	TDataType dout = **null**; TDataType din = TINEType.getWriteTDataType(type, value); TLink link = **new** TLink(devName, propName, dout,din, access); **int** status = link.execute(1000);	Setting value