A 2ND GENERATION NETWORK DISTRIBUTED APPLICATION PROGRAMMING INTERFACE FOR EPICS

J. Hill[1]
[1]*LANSCE, Los Alamos, USA*

ABSTRACT

The programming interface for developing EPICS[1] network distributed applications is called Channel Access. A high priority has been placed on stability and backwards compatibility with this interface, and accordingly it has undergone relatively few externally visible revisions during the lifespan of EPICS. There are of course fundamental tensions where the desire to maintain a stable interface competes with freedom to make technological improvements, and we expect to accomplish more when we have opportunities to make structural changes that will be seen by users. We have an accumulated list of potential improvements all motivated by weaknesses we can identify in the current interface. For example, the set of available meta-data, the mix of meta-data, and the extent that multiple parameters might be synchronized together into a command-response message must be application extensible. Likewise, the EPICS system, which is based on the publish-and-subscribe distributed system model, needs an application extensible set of subscription update triggering events along with additional flexibility related to client application specification of server event filtering, event matching, and property matching combinatorial expressions. Having recently completed a major release, the EPICS core developers are emerging from a code-polishing phase, and are in the midst of addressing the restrictions mentioned above. Our functional requirements and fundamental design decisions are discussed. Finally, some example code is provided in order to give some perspective for code authors who might use this interface.

INTRODUCTION

EPICS[1] has become the defacto standard project portable open source control system for physics experiments. The client server communication programming interfaces and associated support libraries for EPICS are called Channel Access[2][3]. These interfaces allow an application to establish a virtual channel (circuit) between the client side application and a process variable maintained by (or accessible to) a service. In order to not restrict performance the interfaces are fully asynchronous with callback based completion notification. In that respect the original design, choosing efficiency over easy of use for the most visible programming interface in the system, has been somewhat unusual compared to some of the more popular communication programming interfaces. The channel model as originally conceived is quite restrictive, with a fixed set of externally visible process control associated properties. That has enabled a wide range of site independent plug-compatible client side tools to be developed for the system, but has also made the system less useful outside of a process control context.

FUNCTIONAL REQUIREMENTS

We would like to see EPICS capabilities enhanced to meet the demands of data acquisition systems. In particular, we feel that the system should allow a device to extend the set of system events and channel properties seen through external interfaces. A service should be able to publish a device dependent event with a device dependent property capsule. The set of events and the set of properties should not be predetermined when the core components of the system are built – as is the status quo.

Furthermore, there need to be advanced subscription update criteria. Clients of the system need to be able to specify the required event set, excluded event set, property subset, maximum update rate, minimum update rate, % change based update rate, a property match criteria, and or the maximum length of the event queue maintained between high priority function block processing threads and the Channel Access Server.

We would like to see EPICS capabilities increased to better integrate intelligent instruments. These types of devices need support for message passing, complex command completion synchronization, and multi-property atomic reads and writes potentially crossing EPICS record (function block) boundaries. These types of advanced devices also require read, write, and or command response requests to be synchronized with events.

Finally, we see the need to add a snap-in interface for name resolution allowing sites to configure in alternatives to the default name resolution based on IP broadcast (in future versions also IP-multicast).

OUR DESIGN

In a previous paper we have identified weaknesses in EPICS related to how the system is integrated with high-level applications. To address that issue we defined foundation programming interfaces, and created associated support libraries, which we now call Data Access[4][5]. These interfaces allow data exporting programs to define containers of properties that can be uniformly indexed by a wide range of applications. Individual properties within containers are tagged with property identities making their standardized purposes known to the system. This is very unlike remote procedure call systems like CORBA where the purposes of fields within structures, or the purposes of parameters passed within messages, are unknown to the communication system. These additional property tagging capabilities allows Data Access to transport data between loosely coupled systems. For example, when entering a subscription a client can request a subset of the properties that are posted with an event. This decoupling of the data spaces of client applications from the data space of services is expected to be a fundamental enabler for data acquisition capabilities, device orientation, and a tool-based approach to high level application development. Properties may be stored as scalars and as arrays in all of the primitive data types. These interfaces are also carefully designed to allow property containers to be efficiently indexed when the structure of the data is known at compile time. Data Access is a minimalist interface with an emphasis on making it easy for the programmer to export data. Bridge classes layered over the fundamental Data Access interfaces are employed to provide the easy to use data manipulating interfaces that data consumers need – invariant of what type of data might be interfaced.

Another important consideration for embedded systems is memory management. Continuous allocation and de-allocation of random sized blocks has a tendency to fragment the memory allocation pool, and fragmentation invariably leads to increased overhead. Advance pool management schemes help to reduce the impact of fragmentation, but this is difficult to accomplish without introducing additional overhead. Another approach to memory management is to allocate memory for fixed sized data structures off of free lists. When the data structure is no longer needed it can be returned to the free list for later reuse. This approach reduces fragmentation related overhead but is typically inappropriate for randomly sized, contiguous data structures such as strings and arrays. In contrast, Data Access interfaces do not require that arrays and strings be stored contiguously, and therefore they allow for high performance free list based data storage implementations in embedded systems where memory management has a critical impact on the performance and resilience of the controller. This inherent flexibility in data access also eliminates any need for a maximum block size configuration parameter.

When designing communication systems one must choose between synchronous interfaces and asynchronous interfaces. Asynchronous interfaces are harder for inexperienced programmers to use, but offer two orders of magnitude improvement in maximum throughput performance under optimum request batching circumstances. It is possible to layer synchronous layers above asynchronous interfaces, but not visa-versa, and so highest performance foundation layers are inevitably

asynchronous. This was the conclusion with the previous generation Channel Access interfaces and the next generation is based on similar conclusions. Furthermore, with the Data Access interfaces we see a somewhat parallel situation. We have an overloaded callback for each primitive type calling directly into code compiled to process that particular primitive type. That type of interface might be somewhat harder for simple programs to use but we believe that the increased learning curve cost is acceptable given that performance can be optimized. Furthermore, directly invoked interfaces can always be layered above callback based system programming interfaces, as might be preferred by the simplest types of programs. Of course the opposite (callback based layered upon directly invoked) is not a design option. This type of interface does require coding a unique callback for each of the primitive types. In the past code executing the same algorithm for a number of different primitive types was difficult to maintain, but C++ templates now greatly reduce that effort.

Another important consideration when designing this type of interface is mutual exclusion locking overhead. Multiple threads might be using the library at once. Should we take a lock and then release it inside of each call in the library? Our conclusion was that this requires too much overhead for fine-grained requests. Instead, all of the next generation interfaces require an additional parameter – a "guard" C++ object passed by reference. This "guard" class is very simple only taking a mutex lock in its constructor and releasing it in its destructor. By passing a reference to this "guard" object we can guarantee that the lock is held and thereby allow the same lock / unlock pair overhead to be amortized over several calls to the library. This design has a strong potential for improved performance compared to previous interfaces. The same design is also used with callbacks from the library. Here we have an even stronger benefit: it becomes clear to users what the locking strategy for the library is, and that greatly helps to avoid deadlocks between user code and the library when a multithreaded system uses callbacks.

Overall, with the new design we have followed a consistent approach compared to our past designs; we have sacrificed ease of use in the interest of efficiency with the foundation interfaces, but this is done fully anticipating that easier-to-use interfaces can be trivially overlaid upon the foundation interfaces.

DATA INTERFACING EXAMPLE – COMPILE TIME KNOWLEDGE

Discussions of programming interfaces without concrete examples have a tendency to be overly abstract. Therefore we have included some source code in the paper.

Here we have an example where a C++ data structure whose format is known at compile time is exported using the Data Access interfaces. Internally our class stores some properties. We would like to use the PropertyCatalog interface to index and traverse this container of properties. There are two member functions, propertyTraverse and propertyFind, which are used to implement the Data Access interface.

```
class StatsCPU {

public:
   void set ( const PropertyContainer & );
   void get ( PropertyContainer & ) const;

private:
   int num; float temp; double load;

   template < class VIEWER >
   void StatsCPU :: propertyTraverse (
      VIEWER & viewer );

   template < class VIEWER >
   void StatsCPU :: propertyFind (
```

```
    const PropertyId &,VIEWER & viewer );

};
```

Next we have an implementation of a "traverse" function that can be used to enumerate the properties within the container by calling a reveal function in the viewer object identifying each property using a PropertyId object. A PropertyId is a thin veneer class around an integer uniquely identifying each property.

```
extern PropertyId cpuNumber_p;
extern PropertyId cpuTemp_p;
extern PropertyId cpuLoad_p;

template < class VIEWER >
void StatsCPU :: propertyTraverse ( VIEWER & viewer )
{
  viewer.reveal ( cpuNumber_p, &StatsCPU::num );
  viewer.reveal ( cpuTemp_p, &StatsCPU::temp );
  viewer.reveal ( cpuLoad_p, &StatsCPU::load );
}
```

Next we have an implementation of a "find" function that can be used to index a specified property within the container and identify it by calling the reveal function in the viewer object.

```
template < class VIEWER >
void statsCPU :: propertyFind (
        const PropertyId & id, VIEWER & viewer )
{
  if ( id == cpuNumber_p )
    viewer.reveal ( cpuNumber_p, &statsCPU::num );
  else if ( id == cpuTemp_p )
    viewer.reveal ( cpuTemp_p, &statsCPU::temp );
  else if ( id == cpuLoad_p )
    viewer.reveal ( cpuLoad_p, &statsCPU::load );
}
```

Finally, we show, as an example, a get function which copies the properties in the container into a supplied container – should we desire to allow that type of access in the public interface.

```
void statsCPU :: get ( PropertyContainer & out ) const
{
  ObjectCatalog < statsCPU, PropertyViewer >  catalog ( *this );
  out  = catalog;
}
```

DATA INTERFACING EXAMPLE – NO COMPILE TIME KNOWLEDGE

Here we have an example where C++ programmer must interface with data whose structure isn't known at compile time using the Data Access interfaces. The dumpCatalog function prints the entire contents of the specified container of properties.

```
void dumpCatalog ( ostream & cout, PropertyCatalog & X )
{
  PropertyViewerTempl < StreamViewer > viewer ( cout );
  X.traverse ( viewer );
```

```
}
```

The “PropertyViewerTempl” in the function above automates the creation of a viewer class by instantiating an overloaded reveal function for each of the primitive types. That template calls another template below providing a personality for the viewer. In this example its personality is dumping the property contents to standard out.

```
template < class T >
inline void StreamViewer :: reveal (
   const PropertyId & id, const T & property,
   const PropertyCatalog & subordinates = voidCatalog )
{
   outStream << id << “ = “ << property;
   dumpCatalog ( outStream, subordinates );
}
```

CHANNEL ACCESS CLIENT EXAMPLE

Finally we show some code interfacing at the client level. We can see that a guard class guarantees mutual exclusion, that a property catalog is used to specify what data will be received, and that this property catalog is also used to receive the requested data.

```
using namespace ca;

static epicsMutex myMutex;

epicsGuard guard ( myMutex );

Channel & chan = myClientContext.createChannel ( guard, "fred" );

PropertyCatalogRegistration & pcr =
   MyContainer::createPropertyCatalogRegistration ( guard, myClientContext );


propertyCatalogRegistration & statsCPU::createPropertyCatalogRegistration (
      epicsGuard & guard, clientContext & ctx )
{
   ClassCatalog < StatsCPU, PropertySurveyor > surveyor;
   return ctx.createRegistration ( guard, surveyor );
}

readRequest rr = myChan.createReadRequest ( guard, pcr ) ;

rr.read ( guard, myReadCompletionNotifyInstance );

class myReadCompletionNotify public readCompletionNotify {
public:
   void success ( epicsGuard &, const propertyCatalog & incoming )
   {
      statsCPU.set ( incoming );
   }
   void exception ( epicsGuard &, const diagnostic & diag )
   {
      throw diag;
   }
```

```
private:
   StatsCPU statsCPU;
} myReadCompletionNotifyInstance;
```

CONCLUSION

A new Channel Access application-programming interface is being designed for EPICS. We followed a conventional software design process starting with functional requirements, proceeding to design constraints, and cumulating with an interface design. Efficiency, and its relation to callback based interfaces, embedded system memory management, and mutual exclusion locking granularity was an important design consideration. Finally, an abbreviated example showing how the interface might be used was included as a practical perspective for programmers.

REFERENCES

[1] http://www.aps.anl.gov/epics/
[2] J. Hill, "Channel Access: A Software Bus for the LAACS," ICALEPCS, Vancouver, British Columbia, Canada, 1989.
[3] J. Hill, “EPICS Communication Loss Management", ICALEPCS’93, Berlin, 1993.
[4] J. Hill, R. Lange, “Next Generation EPICS Interface TO Abstract Data”, ICALEPCS’2001, San Jose California, USA, October 2001.
[5] R. Lange, J. Hill, “Data Access — Experiences Implementing An Object Oriented Library On Various Platforms”, ICALEPCS’2001, San Jose California, USA, October 2001.