Next Generation CA Client API

Jeff Hill

Summary

- Functional requirement highlights
- Our design
- Example client applications

API Functional Requirements – Backwards Compatibility

- Backwards compatible client API

API Functional Requirements – Interface Design

- Eliminate maximum data size configuration parameters
- Unified client and server programming interfaces

API Functional Requirements – Data Packaging

- User extensible meta-data
  - Channel properties
  - Event properties
- Multi-dimensional arrays
- Unlimited length strings

API Functional Requirements – Data Acquisition

- Application extensible event set
  - Server posts event “arcDown”
    - – Application specific multi-property capsule supplied with each post
      - Within an IOC hopefully this originates both from the database and device support
  - Client subscribes for event “arcDown”
    - – Specifies subset of properties to be copied from the capsule posted with the event

API Functional Requirements – Data acquisition

- Advanced subscription update payload composition
  - Subset of available properties
    - – Decoupled Client and Server data spaces
  - Property selected from event payload
    - – Mutex synchronized
  - Property selected from an unrelated channel
    - – Scheduling priority synchronized

API Functional Requirements – Data acquisition

- Advanced subscription update trigger criteria
  - Each event has one or more triggers
    - Trigger set is client and server extendable
      - Client and server need only agree on the name, purpose, and minimum property set
    - Triggering events from that channel
      - Must be present set, must not be present set

API Functional Requirements – Data acquisition

- Advanced subscription update trigger criteria
  - Periodic
    - Maximum, minimum, fixed period
  - Arbitrary channel property expressions
    - % change, absolute value, relative value
    - Property match criteria
      - Multiple properties
      - Event properties match criteria
      - Channel properties match criteria
        - Possibly properties of some other channel
  - Event queue length

API Functional Requirements – Database mirroring

- Channel mirror event
  - Event payload has only the properties that have changed in it
  - Subscription callback passes in only the properties that have changed
  - Implementation issues need to be better understood

API Functional Requirements – Intelligent instruments

- Message passing
  - Device defines multi-property request/response interfaces
    - – Command completion synchronization
    - – Multi-property atomic reads and writes
      - Hypothetically crossing record boundaries
- Event synchronized requests
  - Gets, puts, or message interaction synchronized to events in the event queue

API Functional Requirements – Name Resolution

- Name resolution snap-in

Our Design – Data Packaging

- Eliminate maximum data size configuration parameters
  - Must have efficient non-fragmenting memory management
  - Therefore, do not preclude non-contiguous storage of all large data items
    - Arrays
    - Strings – can be very large
    - Containers

Our Design – Data Interfacing

- Based on Data Access
  - What it is
    - A minimalist interface and support library to be used when interfacing data to infrastructure
      - Communicating proprietary data containers
      - Transferring between proprietary data containers
      - Comparing proprietary data containers
  - What it isn't
    - A container to store data in
      - its only an interface to data

Our Design – Guard Classes

- Too much overhead to take and release mutex lock in every function in the library
- This is avoided with guard class
  - Guard class takes mutex in its constructor
  - Guard class releases mutex in its destructor
- Library interface requires reference to guard class
  - One lock / unlock pair amortized over several calls

Interfacing to User Defined Property Sets

- Two situations
  - Property set defined at compile time
    - Typical of devices, IOCs, client side tools, site specific applications
      - Example: server event queue
    - Efficient implementation possible, necessary
  - Property set unknown at compile time
    - Typical of parameter page like client side tools like probe and the CA gateway

Data Interfacing Example – Compile Time Knowledge

```
class StatsCPU {

public:
    void set ( const PropertyCatalog & );
    void get ( PropertyCatalog & ) const;
private:
    int num; float temp; double load;

    template < class VIEWER >
        void statsCPU :: propertyFind (
            const PropertyId & id, VIEWER & viewer );

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

Data Interfacing Example – Compile Time Knowledge

```
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 );
}
```

Data Interfacing Example – Compile Time Knowledge

```
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 );
}
```

Data Interfacing Example – Compile Time Knowledge

```
void statsCPU :: set ( const PropertyCatalog & in )
{
    ObjectCatalog < statsCPU, PropertyManipulator >
        catalog ( *this );
    catalog = in;
}

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

Data Interfacing Example – No Compile Time Knowledge

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

void dumpProperty (
    ostream & cout, const PropertyId & id, PropertyCatalog & X )
{
    PropertyViewerTempl < StreamViewer > viewer ( cout );
    X.find ( id, viewer );
}

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

Data Interfacing Example – EZ CA

```
extern PropertyCatalog & containerX;
PropertyContainer bagOfProperties ( containerX );


PropertyContainer::iterator it = bagOfProperties.begin();
while ( it != bagOfProperties.end() ) {
    it->displaySelf ();
}
```

CA Client Examples

- What follows are the lowest level asynchronous interfaces
  - High performance clients need
    - Callback based interfaces
      - Asynchronous completion
      - Primitive type overloaded
    - Guard classes allow mutex context to be reused
      - Over multiple requests
      - Over multiple callbacks
  - EZCA
    - This is layered above the callback and guard based interfaces

CA Client Example – Create Channel

```
using namespace ca;

static epicsMutex myMutex;

epicsGuard guard ( myMutex );

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

CA Client Example – Property Catalog Registration

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


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

CA Client Example – Asynchronous Read Request

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

rr.read ( guard, myReadCompletionNotifyInstance );
```

CA Client Example – Asynchronous Read Response

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