SmartSockets/
Message Formats

05/20/99

Simple SmartSockets Java Example

```
    // This part sends a message
    ....

    public static void main(String[] argv) {

    TipcSrv srv;
    TipcMsg msg;

      try {
// Set location of server (can be done with property file)
        Tut.setOption("ss.server_names", "b0dau30.fnal.gov");
// Get handle to server connection
        srv = TipcSvc.getSrv();
// Create an empty message object
// Use the predefined format INFO (which is just a string)
        msg = TipcSvc.createMsg(TipcMt.INFO);
// Set the subject name of the message
        msg.setDest("/Subject1/SubSubject");
// Append a String to the end of the message
        msg.appendStr("Hello World");
// Queue the message for sending
        srv.send(msg);
// Send all pending messages to the server
        srv.flush();
      } catch (TipcException e) {
        System.out.println("Exception " + e.getMessage());
      }

    ...
```

Simple Java Example - receiver

```
public class SSClient implements TipcProcessCb {

    public static TipcSrv srv;
    private TipcMsg msg;
    private String name;
// callback method in interface TipcProcessCb
    public synchronized void process(TipcMsg msg, Object arg) {
// Set pointer within message object to start, then get string
      msg.setCurrent(0);
      String message = msg.nextStr();
      System.out.println(“Message received was: “ +message);
    }
// Class that receives messages of type INFO and subject name
    public SSClient(String name) {

      this.name = name;
      try {
// Add this object as a callback for this message type
        srv.addProcessCb(this, TipcSvc.lookupMt(TipcMt.INFO),
srv);
// Subscribe to the desired message name
        srv.setSubjectSubscribe(name, true);
      } catch (TipcException e) { }
    }

    public static void main(String[] argv) {

      try {
        Tut.setOption("ss.server_names", "b0dau30.fnal.gov");
        srv = TipcSvc.getSrv();
        new SSClient("/Subject1/SubSubject");
        while (srv.mainLoop(TipcDefs.TIMEOUT_FOREVER)){}
      } catch (TipcException e) {
        System.out.println("Exception " + e.getMessage());
      }
    }
```

Simple SmartSockets C Example

```
    /* This part sends a message */
    ....

    T_IPC_MSG msg;

    void main(int argc, char **argv) {

// Set location of server
        TutCommandParseStr("setopt server_names b0dau30");
// Get handle to server connection
        TipcSrvCreate(T_IPC_SRV_CONN_FULL);
// Create an empty message object
// Use the predefined format INFO (which is just a string)
        msg = TipcMsgCreate(TipcMtLookupByNum(T_MT_INFO));
// Set the subject name of the message
        TipcMsgSetDest(msg, "/Subject1/SubSubject");
// Append a String to the end of the message
        TipcMsgAppendStr(msg, "Hello World");
// Queue the message for sending
        TipcSrvMsgSend(msg, TRUE);
// Send all pending messages to the server
        TipcServFlush();
}

    ...
```

Simple C Example - receiver

```
// callback method
    void process(T_IPC_CONN conn, T_IPC_CONN_PROCESS_CB_DATA
buf, T_CB_ARG arg) {
        T_IPC_MSG msg = buf->msg;
        T_STR message;
// Set pointer within message object to start, then get string
      TipcMsgSetCurrent(msg, 0);
      message = TipcMsgNextStr(msg, &message);
      printf(“Message received was: %s“, message);
    }

    void main(int argc, char **argv) {
// Set location of server
        TutCommandParseStr("setopt server_names b0dau30");
// Get handle to server connection
        TipcSrvCreate(T_IPC_SRV_CONN_FULL);
// Add this object as a callback for this message type
        mt = TipcMtLookupByNum(T_MT_INFO);
        TipcSrvProcessCbCreate(mt, process, NULL);
// Subscribe to the desired message name
        TipcSrvSubjectSetSubscribe(“/Subject1/SubSubject”,
true);
        TipcSrvMainLoop(T_TIMEOUT_FOREVER);
    }
```

Message Types

- A format is associated with a type number for inter-machine translation where required
- There are many predefined formats
- Most are intended for internal use
- The most common general formats are
  - INFO - just a string
  - NUMERIC_DATA - alternating string/real*8
  - Note that neither includes the time
- Custom formats can be defined
- Format given as string - “int4 int8 string” etc.
- Last element assumed to be repeated indefinitely
- Elements can be grouped: int4 int8 {int4 string}
- An element may be another message type
- They must be registered with the server before use
- As types are numbered, not named => management issue
  - The distribution comes with a simple sample program that just takes a text file with a list of numbers/formats and registers them all
  - intended to be run when the server is started

Custom Message Type Example

```
// Register a custom type with three fields:
// 8 byte time, 4 byte severity, string message
        public static final int TYPE_GENERAL_STATUS=2034;
        try {
            TipcMt mt = TipcSvc.createMt(“GeneralStatus”,
                                         TYPE_GENERAL_STATUS,
                                         "int8 int4 string");
        } catch (TipcAlreadyExistsException exc) {
        } catch (TipcException exc) {
            System.out.println(
         "Error creating message type " + exc.getMessage());
        }
 ....
// Unpack a GeneralStatus message:
      msg.setCurrent(0);
      long sendTime = msg.nextInt8();
      int severity = msg.nextInt4();
      String message = msg.nextStr();


// Pack a GeneralStatus message
    try {
      TipcMsg msg = TipcSvc.createMsg(TYPE_GENERAL_STATUS);
      msg.setNumFields(0);
      msg.appendInt8(sendTime);
      msg.appendInt4(severity);
      msg.appendStr(message);
    } catch (TipcException exc) {
      System.out.println("Error creating General Status message
" + exc.getMessage());
    }
```

Some issues

- Send and receive code must know exactly the message format
- Must be kept in synch between C and Java code
- This may or may not be a big deal

What DAQMSG does

- As an illustration, not necessarily a proposal
- Starts with message definition file:

```
const SUCCESS_SEVERITY = 1;
const WARNING_SEVERITY = 2;
const INFO_SEVERITY = 3;
const ERROR_SEVERITY = 4;
const MAX_STRING_LEN = 256;
struct GeneralStatusStruct {
  int severity;
  string messageText<MAX_STRING_LEN>;
};
```

- Then run code generators on this:
  - C code generator from NDDS package
  - Custom written Java code generator

Generated C code:

- .h file with C struct definition
- Unpack an input message into the structure
- Register the type with NDDS (registration based on name)
- Print the structure
- Serialize/deserialize to XDR stream (mostly invisible)

Generated Java code

- Java class
  - Member variables corresponding to structure elements
  - Various status information (last time/status etc.)
  - Associated get/set methods
  - Callouts to above C code to register message type, publish, subscribe to data
- JNI C code
  - Handle above callouts
  - C callback to NDDS to receive data, unpack it and store it directly in the object member variables
- Also provided “home-brew” serialization into Strings (field name&&type&&data) for use by Applet server
  - C callout restricted in applets
  - Home-brew serialization turned out to be 3xfaster than Java Object Serialization
- Usage analogous to smartsockets - example callback:

```
public void onMessage(BaseMessage Message) {
    GeneralStatus msg = (GeneralStatus) Message;
    System.out.println("GeneralStatus message:  " +
msg.getMessageText() + msg.getSeverity());
```

- Difference is the casual user doesn’t need to know the internals of the class, just call various get/set methods

- Possibly useful to automatically generate a class with
  - member variables corresponding to message fields
  - (static?) methods that
    - » Given an instance of some class, return a TipcMsg
    - » Vice Versa
- Possibly also C code
  - Structure corresponding to fields
  - Message <-> structure as above

Possibilities?

- Just use SmartSockets as - is
- Do some automatic code generation:
  - Custom message definition file like DAQMSG
  - Basic Java class
    - » Add packing/unpacking code - analogs in other products
    - » Generate C code using reflection - (as in calibration database API)
  - XML

XML

- Extensible Markup Language
- Standard under auspices of WWW consortium
- The “Next Big Thing”
- Markup language analogous to HTML to describe data
  - Documents do not describe how data is displayed
  - However a “style sheet” may be defined that does that and/or other things as well
- Intended as platform independent way of exchanging structured data
- Various tools available to parse, create, validate documents

Ingredients

- Document Type Definition (describes format)

```
<!ELEMENT GeneralStatus (EMPTY)>
<!ATTLIST Severity (Info|Warning|Error|Severe)#REQUIRED
          Message CDATA #REQUIRED
>
```

- Document:

```
<?xml version="1.0"?>
<!DOCTYPE GeneralStatus "GeneralStatus.dtd">
<GeneralStatus
   Severity="Info"
   Message = "All is Well">
</GeneralStatus>
```

Style Sheets

- Describe a display format to render to html:

```
<?xml version=“1.0”?>
<xsl:template match=“/GeneralStatus”>
  <xsl:if test=‘Severity[not(.=“Severe”)>
  <tr BGCOLOR=“Green”>
  </xsl:if>
  <xsl:if test=‘Severity[.=“Severe”]
  <tr BGCOLOR=“Red”>
  <td>
  </xsl:if>
  <td>
  <xsl:value-of select=“Severity”>
  </td>
  <td>
  <xsl:value-of select=“Message”>
  </td>
  </tr>
</xsl:template>
```

Style Sheets cont.

- Can also use for code generation. Consider a “message definition” document:

```
<?xml version=“1.0”?>
<!DOCTYPE MessageDefinition “MessageDefinition.dtd”>
<MessageDefinition
   Name=“GeneralStatus”/>
<Attributes>
   <Attribute name=“Severity” format=“int”/>
   <Attribute name=“Message” format=“String”/>
</Attributes>
</MessageDefinition>
```

- And a style sheet:

```
<?xml version=“1.0”?>
<xsl:template match=“/MessageDefinition”>
  import com.smartsockets.*;
  public class <xsl:value-of select=“Name”> {
  <xsl:for-each select=“Attribute”>
      private <xsl:value-of select=“format”> <xsl:value-of
select=“name”>;
  </xsl:for-each>
  }
      . . .

</xsl:template>
```

Programatic Interface

- Various parsers exist in various languages
- For Java, a “Document Object Model” is defined
  - Invoking a parser on a document returns a DOM object:

```
doc = parser.readStream(stream);
String root = doc.getRootName();
for (Node next=doc.getDocumentElement().getFirstChild();
         next != null; next = next.getNextSibling();) {
   System.out.println(next.getNodeName());
   next.getAttribute(“Severity”);
   next.getAttribute(“Message”);
}
```

- Parsers for C exist as well - plain C, nothing to bother VxWorks

Possibilities?

- Just use SmartSockets as - is
- Do some automatic code generation:
  - Custom message definition file like DAQMSG
  - Basic Java class
    - » Add packing/unpacking code - analogs in other products
    - » Generate C code using reflection - (as in calibration database API)
  - XML
    - » Could be used just for code generation, or both for that and transmission of the data
      - message formats would all then be just a string (+ time probably)
    - » Leverage industry development of validating parsers and other tools.
    - » Allows significant flexibility in evolving formats
    - » Very new, no current expertise
    - » Very verbose
  - Anything other ideas?