MDCF Tutorial
-- Communication Infrastructure

http://mdcf.santos.cis.ksu.edu/

Kansas State University
University of Pennsylvania

Acknowledgements:
Funding provided by US National Science Foundation awards 0734204, 0930647, 0932289
Tutorial Topics

Communication Infrastructure

App Development Environment

Mapping to ICE Architecture

Mock Devices
Why Publish/Subscribe?

- Loose coupling of information producers (devices) and consumers (displays, app logic components)...
  - makes it easy to add new components to the system
  - simplifies programming
  - allows flexible combinations of components
- Consumers are activated only when requested data arrives
- Data sources can easily be shared among multiple consumers
- Many different forms of open-source and enterprise quality implementations of “Message Oriented Middleware” are available
MDCF Publish/Subscribe

- MDCF pub/sub layer is abstracted by an interface IMdcfChannelService
- Originally implemented using the Java Messaging Service (JMS) API with OpenJMS and ActiveMQ
  - Note: Cerner uses an enterprise quality JMS implementation for their iBus device integration framework
- Now building out our own implementation (mBus) with real-time and security capabilities
  - Aiming for easy installation, small footprint, and easy experimentation
  - Not advocating our own implementation as ultimate solution; instead using it as a mechanism for gathering requirements
MDCF Channel Service

Interface layer

Implementations
Sender

createTopic(...<my topic name>...);
 ...
 mySender = SenderFactory.createSender();
 mySender.connect(...<my topic name>...);
 ...
 mySender.sendMessage(...<string data>...);

- Current release also includes basic hooks for security credentials
- Next release will also include real-time properties on channel creation
- Current release has all messages as string, with wrappers to convert other types into strings
- Next release will include a more sophisticated type system
MDCF Channel Service Overview

Receiver

class MyListener implements IMdcfMessageListener{
    @Override
    public void onMessage (MdcfMessage message){
        ...<handle message>...
    }
}

creates

Topic

Receiver Factory

Usually auto-generated by higher-level abstractions

Device interface and app writer usually only uses this method
MDCF Component Model

MDCF uses a uniform notion of component to represent units of an app as well as device interfaces.

Currently, component ports are restricted to asynchronous events (with payload), ports will be significantly enhanced in the next version.
Currently, MDCF support three different component roles (device, app logic, and app panel) where the role determines the structure of the code templates that are auto-generated.

MDCF uses a uniform notion of *component* to represents *units* of an app as well as device interfaces.
For example, the app developer simply declares a connection ("wiring") between ports, and the framework will allocate a topic dedicated to that connection.
MDCF Component Model

For each port, auto-generated code templates include the appropriate calls to set up the connections to the topic represented by the port.

Auto-generated code for port `myOutPort`

```java
myOutPortSender = SenderFactory.createSender();
myOutPortSender.connect(
    pubAssign.topicMap.get("myOutPort");
```

Component developer writes the following to send on `myOutPort`

```java
myOutPortSender.sendMessage(...<string data>...);
```
For each port, auto-generated code templates include the appropriate calls to set up the connections to the topic represented by the port.

```java
class myInPortListener implements IMdcfMessageListener{
    @Override
    public void onMessage (MdcfMessage message){
        ...<handle message>...
    }
}
```

MyInPortReceiver = ReceiverFactory.createReceiver();
myInPortReceiver.connect(subAssign.topicMap.get("rawTicktockIn"));
...
myInPortReceiver.registerMessageListener(new myInPortListener());

Component developer writes the body of onMessage
Realizing ICE

MDCF apps are configurations of components. While implementations of **App Logic** and **App Panel** must be included in the app and instances of these components are created at app launch, components of role **Device** represent

MDCF component with role **Device** is analogous to the ICE Device Model (interface) for a device

Auto-generated code forms the basis of building an "adapter" which converts the native signals of a device to a form that the MDCF Network Controller can understand
With Java-based device-side MDCF clients, we are not aiming to be completely realistic (most devices are not designed to run a JVM to host an interface), but we want to be realistic enough for effective experimentation.

We have been able to run an MDCF client on an Android device.

Experimental “Gum Pack” platform (Kejia Li), running embedded Linux with wireless connection.

Run a mock device, or have the MDCF client use platform interface to connect to legacy device, i.e., RS-232.
Conclusion

- MDCF is based on a publish-subscribe middleware
- A component model on top of the pub-sub middleware provides a uniform abstraction for devices, app components, and other services of the ICE infrastructure
- MDCF is designed with an interface layer that allows the underlying pub-sub middleware implementation to be switch relatively easily
- Distributed clients (e.g., devices) of the MDCF Server (i.e., ICE Network Controller) can easily interface if they are running a JVM
  - Alternatively, the code generation facilities of MDCF could be retargeted to generate client code in other languages (e.g., C).
- Looking ahead...
  - Significantly improved app development environment will continue to build on component model concept
  - Incorporating the notion of ICE Device Model within MDCF will attempt to integrate with the notion of *port interfaces* in the current component model