# Implementing IEEE 1451.1 in a Wireless Environment


Rick Schneeman, Computer Scientist

*rschneeman@nist.gov*

US Department of Commerce

National Institute of Standards and Technology (NIST)

Gaithersburg, Maryland 20899 USA


NIST CENTENNIAL 1901-2001

# Introduction

- Who we are: NIST mission is to help increase US industry competitiveness through advanced research, standards, and technology collaboration

- Member of the Sensor Development and Application Group (SDAG) within the Manufacturing Engineering Laboratory (MEL) at NIST

- Member of the Working Group on the IEEE Standard for a Smart Transducer Interface for Sensors and Actuators — Network Capable Application Processor (NCAP) Information Model, or IEEE 1451.1 ("dot1")

# Topics of Discussion

- Part 1: Provide a brief object-based overview of the IEEE 1451.1 components, services, and block classes
- Part 2: Discuss the technical and architectural solutions for the development and deployment of the NIST IEEE 1451.1 reference implementation
- Part 3: Illustrate the use of IEEE 1451.1 in an example application using both the NIST C++ and Java reference implementations
- Part 4: Describe an IEEE 802.11b (11Mbps) wireless environment used for application testing and demonstration

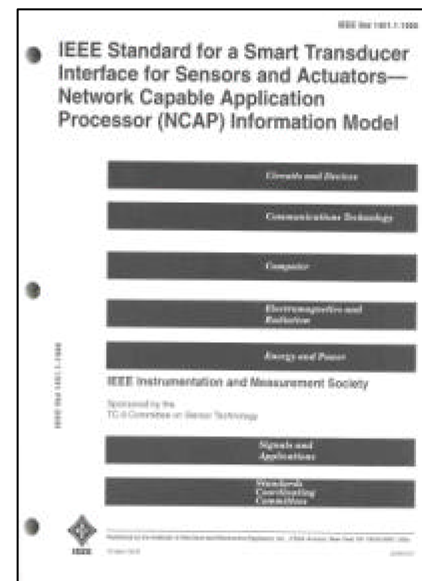
# IEEE 1451 Overview/Goals

- Provide standardized communication interfaces for smart transducers, both sensors and actuators. In the form of a standard hardware and software definition/specification.
- Simplify the connectivity and maintenance of transducers to device networks through such mechanisms as common Transducer Electronic Data Sheet (TEDS) and standardized Application Programming Interfaces (API)
- Allow plug-and-play with 1451 compatible transducers among different devices using multiple control networks
- Give sensor manufacturers, system integrators, and end-users the ability to support multiple networks and transducer families in a cost effective way

# Part 1: IEEE 1451.1 Overview/Goals

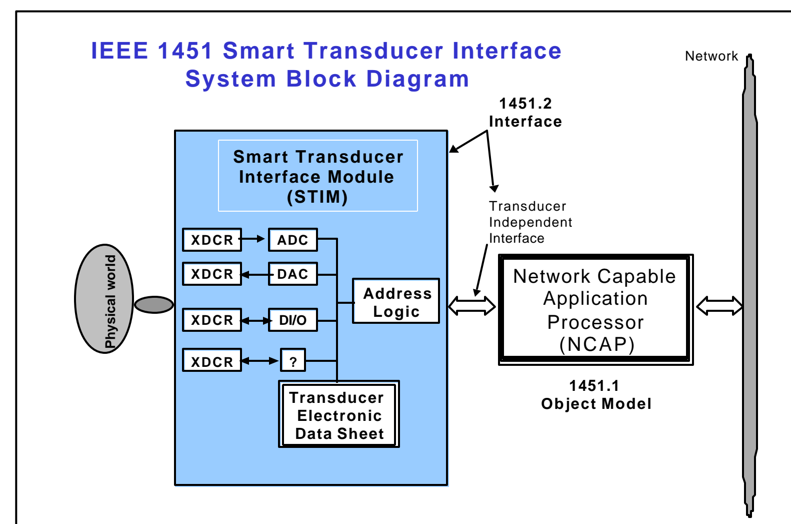
1901 - 2001
NIST CENTENNIAL

- “*The specifications provide a comprehensive data model for the factory floor, and a simple application framework to build interoperable distributed applications*…” Dr. Jay Warrior, Agilent Technologies, Chair IEEE 1451.1 WG
- In general, IEEE 1451.1 accomplishes this by providing:
  - Transducer application portability (software reuse)
  - Plug-and-play software capabilities (components)
  - Network independence (network abstraction layer)


IEEE Standard for a Smart Transducer Interface for Sensors and Actuators—Network Capable Application Processor (NCAP) Information Model

IEEE Instrumentation and Measurement Society

- The standard specifies these capabilities by defining software interfaces for:
  - Application functions in the NCAP that interact with the network that are independent of any network
  - Application functions in the NCAP that interact with the transducers that are independent of any specific transducer driver interface

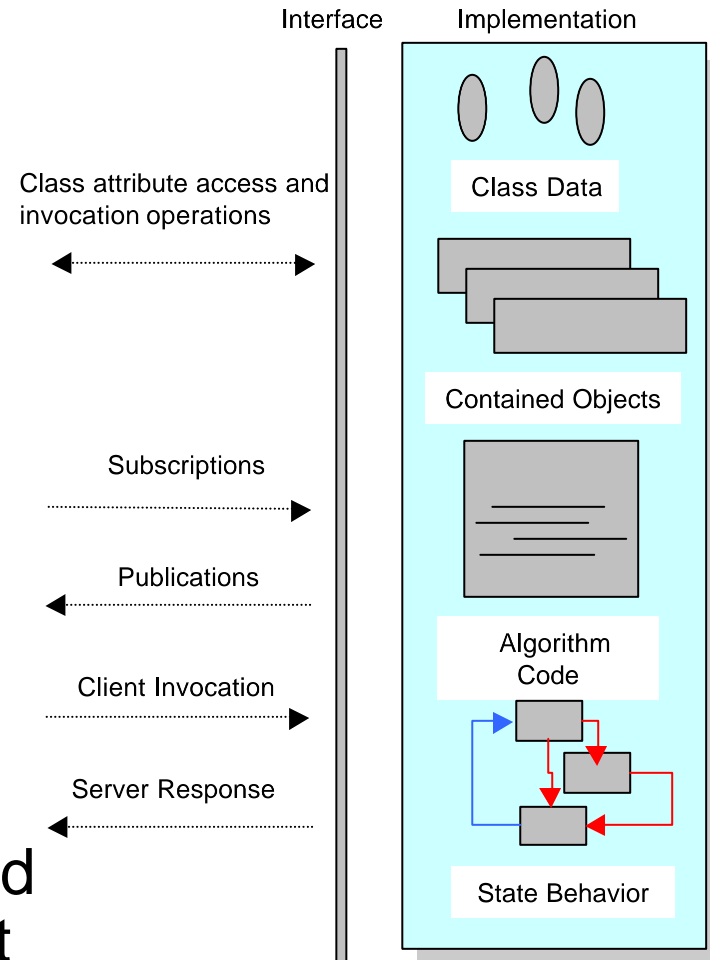


# IEEE 1451.1 Overview/Goals (Cont.)

- IEEE 1451.1 software architecture is defined using three different models or views of the transducer device environment:
  - An Object Model, defines transducer device specific abstract objects – or, classes with attributes, methods, and state behavior
  - A Data Model, defines information encoding rules for transmitting information across both local and remote object interfaces
  - A Network Communication Model, supports a client/server and publish/subscribe paradigm for communicating information between NCAPs

# Inside an IEEE 1451.1 Object

- IEEE 1451.1 objects look similar to other object-oriented class definitions; they are comprised of instance data, other classes, code for implementing internal operations or methods, and state behavior machines
- Defines services and interfaces required for distributed smart devices (i.e., object discovery, invocation, synchronization) via attribute access and operations
- Specifies Object interfaces and behavior using the Object Model
- The Object Model is formally described using the implementation independent Interface Description Language (IDL)



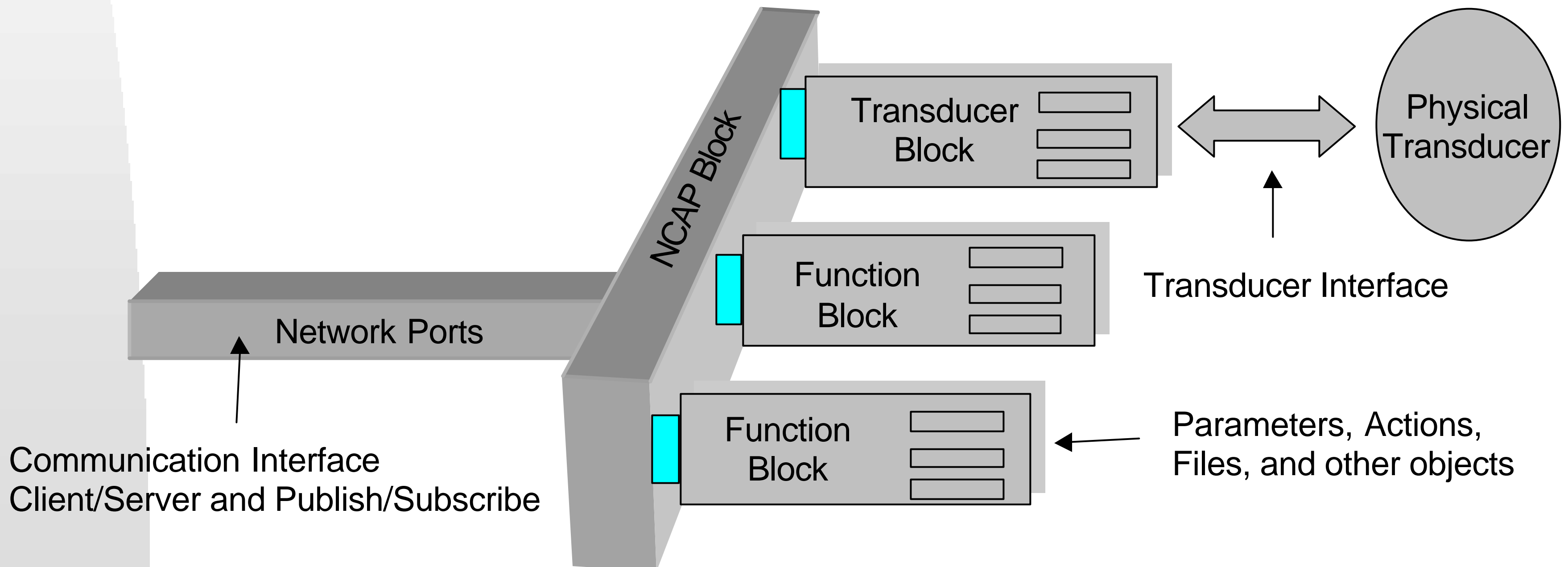
# Types of IEEE 1451.1 Classes

NIST CENTENNIAL 1901-2001

- Four Object classes are found in an IEEE 1451.1 system:
  - Block Classes (building blocks of the system)
    - NCAP Block (network communication and configuration)
    - Function Block (application-specific functionality)
    - Transducer Block (transducer device driver interface w/app)
  - Component Classes (common application constructs)
    - Parameter (contains structured information, network variables)
    - Action (time-based system state altering activity)
    - File (supports downloading new code to device)
    - Component Group (addressing collections of related entities)
  - Service Classes (system and network services)
    - Client Ports (implements client-side communication endpoint)
    - Publisher Ports (implements publishing endpoint)
    - Subscriber Ports (implements subscription endpoint)
    - Mutex/Condition Service (provides application/NCAP synch)
  - Non-IEEE 1451.1 Classes

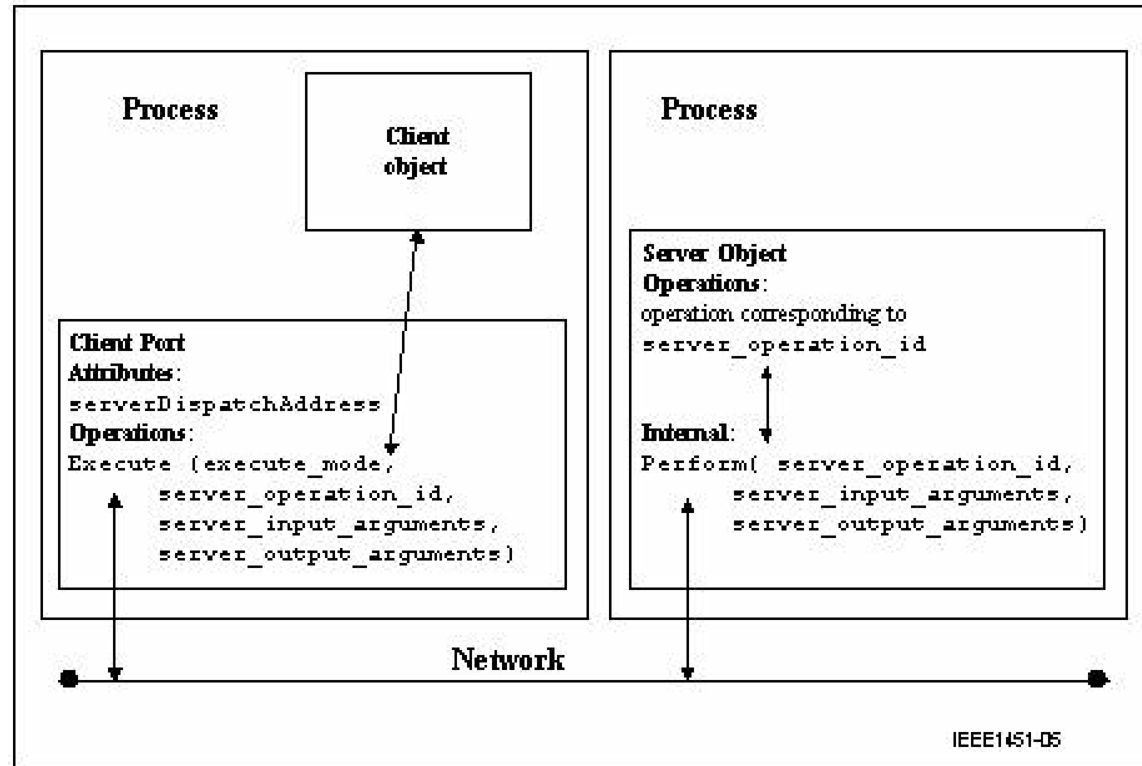
# Conceptual View of an IEEE 1451.1 NCAP

- Uses a “backplane” or “card cage” concept
- NCAP centralizes and “glues” all the system and communications facilities together
- Network communication viewed through the NCAP as ports
- Function block application code is “plugged” in as needed
- Transducer blocks map the physical transducer to the NCAP



# IEEE 1451.1 Communication Model

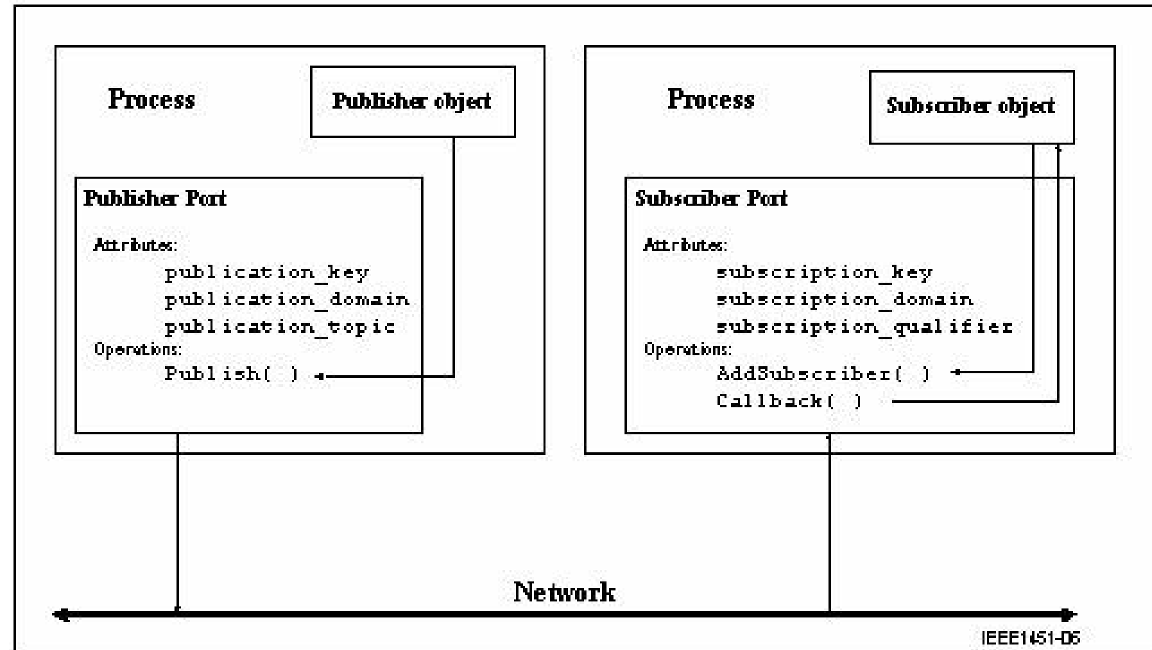
- Provides two styles of inter-NCAP communication
- Client/Server: A tightly coupled, point-to-point model for one-to-one communication scenarios – typically used for configuration, attribute accessors, and operation invocations.




- Client objects "Execute()" or invoke operations over the network against a Server NCAP. Server NCAP objects "Perform()" the operation based on the ID and return the results to the client.

# IEEE 1451.1 Communication Model (Cont.)

- Publish/Subscribe: A loosely coupled, model for many-to-many and one-to-many communication scenarios – typically used for broadcasting or multicasting measurement data and configuration management (i.e., node or NCAP discovery) information




- The Publisher is the sending object, it invokes "Publish()" method and does not need to be aware of any receiving objects. Subscribers issue AddSubscriber() method to register interest in something on that subscription.

# Part 2: Implementing IEEE 1451.1

NIST CENTENNIAL 1901-2001

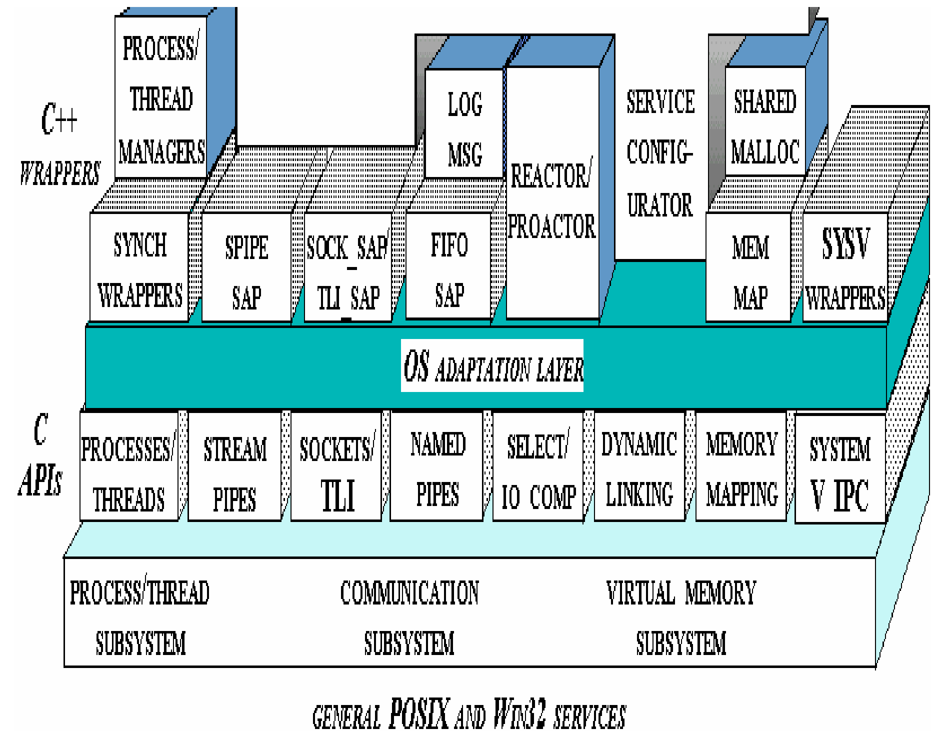
- An IEEE 1451.1 C++ Reference Implementation provides a concrete representation of the abstract Smart Transducer Information Model (IEEE Std 1451.1-1999, Dated 18 April 2000). The NIST implementation is called “1451.1 Lite”, as it is a subset of the complete specification.

- A subset of the IEEE 1451.1 implementation has also been developed in Java to provide an architecture neutral NCAP configuration tool.

- The C++ implementation uses the open-source Adaptive Communication Environment (ACE) from the Washington University at St. Louis.

# Using ACE Framework for IEEE 1451.1

- ACE is an object-oriented framework for implementing portable real-time communication software patterns in C++.
- All core distribution, concurrency, and communication patterns that underlie the NIST developed IEEE 1451.1 implementation are derived from the ACE library.
- Provides an object-oriented abstraction of operating system services for real-time network and communications support.
- Presently used by major communication companies including: Hughes, Lucent, Ericsson, Siemens, and Boeing among others



# ACE Framework Components

- The Adaptive Communication Environment (ACE) is an object-oriented framework that implements core concurrency and distribution patterns for real-time communication software. ACE includes the following components:
  - Concurrency and Synchronization
  - Interprocess communication (IPC)
  - Memory Management, Timers and Signals
  - File System management
  - Thread Management
  - Event demultiplexing and handler dispatching
  - Connection establishment and service initialization
  - Static and dynamic configuration and reconfiguration of software
  - Distributed communication services –naming, logging, time synchronization, event routing and network locking. etc.
- Notice that the ACE components mimic the object requirements found in the IEEE 1451.1 specification; therefore, the mapping was clear but cumbersome because of the specificity of the standard.

# Key ACE Architectural Areas Leveraged

- ACE uses well-established object-oriented “patterns”, or common design elements, including:
  - Reactor pattern (efficient event de-multiplexing and dispatching)
  - Active Object (multi-threaded execution object)
  - Activation Queues (decouples method invocation/execution)
  - Method Objects (queue able objects for execution of commands)
  - Future Objects (resultant objects of method object execution)
- All of these patterns have concrete representations in ACE, such as ACE_Tasks as an Active Object, etc.
- The low-level C++ TCP/IP socket “wrapper” routines were not used because this would mitigate using the advanced features of the object-oriented framework

# Key ACE Architectural Areas Leveraged

- IEEE 1451.1 “Entity” class forms the base class for all network and block services in the standard. The “Entity” class inherits the ACE “service handler” class, which integrates event-driven “Active” object patterns with TCP/IP communication endpoints and synchronization.
- ACE “service handler” interface supports real-time event-driven input/output on multicast/unicast TCP/IP sockets, providing an efficient networking scheme.
- The IEEE 1451.1 standard specifies a data encoding sequence for passing parameters between objects. An on-the-wire network-based data representation for marshaling these parameters is required for each control network used. For TCP/IP networks, NIST used a CORBA compliant implementation of the Common Data Representation (CDR) library found in ACE.

# Key ACE Architectural Areas Leveraged

- Internally, the NIST IEEE 1451.1 code uses the CDR base types as well. This reduces the amount of encoding and marshaling overhead needed from converting base types to the CDR types during marshaling activities.
- Blocks and Ports use ACE "Tasks" for their multithreaded state machine behavior. Each Block/Port class implementation provides a virtualized svc() routine to support each blocks defined state machine.

```
// enter state machine of the fblock
int tempFBlock::svc( void )
{
    while (1)
    {
        // get current state of underlying Block
        ret = IEEE1451_Block::GetBlockMajorState(bms);
        switch (bms)
        {
            case BL_UNINITIALIZED:break;
            case BL_INACTIVE:break;
            // if the block is in the active state, then
            case BL_ACTIVE:
                // get current state of underlying FBlock
                ret = IEEE1451_FunctionBlock::GetFunctionBlockState(fbs);
                // enter the fblock substate state machine
                switch (fbs)
                {
                    case FB_STOPPED:break;
                    case FB_IDLE:break;
                    case FB_RUNNING:break;
                    case FB_RESERVED:break;
                }
                break;
            case BL_RESERVED:break;
            default:break;
        }
    }
    // compiler issue
    return 0;
}
```

# Progress to Date

NIST CENTENNIAL 1901-2001

- Majority of the design in place (using inheritance and composition of ACE services with 1451.1 code base)
- 70-75% of IEEE 1451.1 C++ code complete, all IEEE 1451.1 methods have placeholders, network encoding still under development
- Core subset (.1 Lite) nearly complete, consists of a shared library under Linux/FreeBSD & VxWorks and a dynamic link library (DLL) under WIN32/NT
- Because the ACE framework is being used, a single source code-base that is 100% portable to Linux/FreeBSD, NT, VxWorks, and other POSIX or WIN32-based operating systems has been developed
- At this time, there is minimal access to the underlying STIM or microprocessor hardware via device drivers, (most I/O such as temperature, pressure, or actuator data has been provided by the NCAP via simulation)
- Java implementation consist of most blocks and communication ports. A CDR encoding from the Zen Project at University of California, Irvine has been integrated and is interoperable with C++ version.

# Lessons Learned on Implementation

- The standards' low-level attempt to redefine the RPC (Remote Procedure Call) mechanisms for network communication preclude and limit the implementer's ability to optimize the amount and types of communication patterns that can be used in normal real-time object-oriented network communication software design.
- Many object-oriented patterns are disallowed because of arbitrary inheritance chain and class partitioning.
- Partitioning forces the designers and implementers of the software to create overloaded and convoluted classes in order to pigeon hole the standard into certain middleware and object-oriented frameworks.

# Lessons Learned on Implementation

NIST CENTENNIAL 1901-2001

- Large number of deployed NCAPs could be problematic for system configuration, initialization, and maintenance unless highly sophisticated configuration tools are developed. However, software vendors can provide those tools.
- Modern middleware provides similar service-oriented class characteristics; an OS adaptation layer standard definition approach would ease integrating application functionality.

# Lessons Learned on Implementation

NIST CENTENNIAL 1901-2001

- The backward or legacy approach to preserving other device bus application interaction complicates and makes the standard over specify itself.
- Implementations of the standard (in C++) are cumbersome and very large for many small embedded systems. A subset standard (IEEE 1451.1 Lite) should be considered for the applications that only need a small subset of the current 1451.1 standard specification.
- NIST implementations unique use of CDR streams as the underlying data typing mechanism provides a less complex asynchronous and synchronous communication marshaling routines.

# IEEE 1451.1 Benefits

NIST CENTENNIAL 1901-2001

- Using P1451.1 provides:
  - an extensible object-oriented model for smart transducer application development and deployment
  - application portability achieved through agreed upon application programming interfaces (API)
  - network neutral interface allows the same application to be plug-and-play across multiple network technologies
  - leverages existing networking technology, does not re-implement any control network software or protocols
  - a common software interface to transducer hardware i/o

NIST CENTENNIAL 1901-2001

# Part 3: Looking at an IEEE 1451.1 Application

- A minimal IEEE 1451.1 application consist of a few classes:
  - An NCAP Block (consolidates system and communication housekeeping)
  - A Transducer Block (provides the software connection to the transducer device)
  - A Function Block (provides the transducer application algorithm (i.e., obtain and multicast temperature data every second)
  - Parameters (contains the network accessible variables that hold and update the data)
  - Ports (network communication objects for publishing and subscribing to information or interacting with other NCAPs using client/server

# A C++ IEEE 1451.1 Application (NCAP Block)

NIST CENTENNIAL 1901-2001

- Creating a NCAP object starts with defining a TCP server port assignment.
- Create a Tag, this is used to identify the NCAP to others on the network
- Build a Dispatch Address for clients to use to talk to NCAP
- Instantiate the Temperature NCAP
- Register the dispatch address with this NCAP
- Initialize the NCAP state
- Tell NCAP to go “Active” or start running

```
/////////////////////////////////////////////////////////////
// Start the client NCAP System initialization process....
/////////////////////////////////////////////////////////////

// declare a return code object for 1451 methods
OpReturnCode ret;

// build a local client-server address & port based on this NCAP
ObjectTag localHost;
UInteger16 localPort;
::configLocalAddress(localHost, localPort);

// need an object tag for the client-server NCAP oda
ObjectTag ncapTag("tempNCAP-NCAPBlock");

// create a generic client-server object dispatch address for this NCAP
//
// the port is dummied to 10002 to allow local debugging on the same
// machine of the tempncap <--> configtool
ObjectDispatchAddress ncapaddr(localHost.fast_rep(), 10002/*localPort*/, ncapTag);

// create a set of object properties including the ncap client-server address
ObjectProperties ncapprops(ncapTag, "owner", ncapaddr, "ncapobj");

// create the NIST NCAP
cout << "Creating a NIST Temperature NCAP ...." << endl;
tempNCAP* ptempNCAP = new tempNCAP( ncapprops,
                                    "NIST","N2",
                                    "01a","NIST MfgID# 85073",
                                    "N2","011",
                                    "VxWorks5.4");

// Register NCAP with itself to make operations on it Network Visible (NCAP is owner)
ret = ptempNCAP->RegisterObject(*ptempNCAP, *ptempNCAP, ncapaddr);
if (ret.majorReturnCode != MJ_COMPLETED)
    cout << "Could not RegisterObject()" << endl;

// initialize the ncap state (start the thread/task)
ret = ptempNCAP->Initialize();

// put the NCAP into the Active State!
ptempNCAP->GoActive();
```

# A C++ IEEE 1451.1 Application (Transducer Block)

NIST CENTENNIAL 1901-2001

- Creating a Transducer Block starts with defining a Tag for this object
- Optionally, another Dispatch address can be generated for this NCAP
- Create a set of Object Properties that give this object special identity
- Instantiate the Transducer Block
- Register the Transducer Block with the NCAP
- Initialize the TBlock state
- Tell TBlock to go “Active” or start running

```
// get the state of the NCAP, should be INACTIVE!
NCAPBlockState  ncapState;
ret = ptempNCAP->GetNCAPBlockState(ncapState);

// print state information
if (ret.majorReturnCode == MJ_COMPLETED)
    cout << "NCAP Block State = " << ((ncapState == 4) ? "NB_INITIALIZED" : "OTHER") << endl

// tell ncap to go active now
//ptempNCAP->GoActive();

//////////////////////////////////////
//// create a transducer block
//////////////////////////////////////

ObjectTag tblockTag("tempNCAP-TBlock");

ObjectDispatchAddress tblockaddr(localHost.fast_rep(), 10003, tblockTag);

// create a set of object properties including the ncap multicast address
// first entry of object properties is what registerObject keys on....
ObjectProperties tblockprops(tblockTag, "owner", tblockaddr, "tblockobj");

// create a specialized transducer block for the temperature NCAP
cout << "Creating a generic Transducer Block ....." << endl;
tempTBlock *ptempTBlock = new tempTBlock(tblockprops, ptempNCAP);

// Register NCAP with itself to make operations on it Network Visible (NCAP is owner)
ret = ptempNCAP->RegisterObject(*ptempTBlock, *ptempNCAP, tblockaddr);
if (ret.majorReturnCode != MJ_COMPLETED)
    cout << "Could not RegisterObject()" << endl;

// initialize the ncap state (start the thread/task)
ret = ptempTBlock->Initialize();

// prime the tblock application before starting
ptempTBlock->GoActive();

//////////////////////////////////////
//// create a function block
//////////////////////////////////////

ObjectTag fblockTag("tempNCAP-FBlock");
```

# A C++ IEEE 1451.1 Application (Function Block)

NIST CENTENNIAL 1901-2001

- Creating a Function Block starts with defining a Tag for this object
- Optionally, another Dispatch address can be generated for this NCAP
- Create a set of Object Properties that give this object special identity
- Instantiate the Function Block
- Register the Function Block with the NCAP
- Initialize the Function Block
- Tell Transducer Block to go "Active"
- Manually "Start()" the application or do it later from a network invocation

```
////////////////////////////////////////
//// create a function block
////////////////////////////////////////

ObjectTag fblockTag("tempNCAP-FBlock");

ObjectDispatchAddress fblockAddr(localHost.fast_rep(), 10003, fblockTag);

// create a set of object properties including the ncap multicast address
// first entry of object properties is what registerObject keys on....
ObjectProperties fblockprops(fblockTag, "owner", fblockAddr, "fblockobj");

// create a generic function block for the application
cout << "Creating a Publisher Function Block ...." << endl;
tempFBlock *ptempFBlock = new tempFBlock(fblockprops, ptempTBlock);

//tempFBlock ptempFBlock(fblockprops, ptempTBlock);

// Register the FBlock to make operations on it Network Visible (NCAP block is owner)
ret = ptempNCAP->RegisterObject(*ptempFBlock, *ptempNCAP, fblockAddr);
if (ret.majorReturnCode != MJ_COMPLETED)
    cout << "Could not RegisterObject()" << endl;

// initialize the ncap state (start the thread/task)
ret = ptempFBlock->Initialize();

// prime the fblock application before starting
ptempFBlock->GoActive();

// now actually start the application
ptempFBlock->Start();

// get the state of the NCAP again, should be ACTIVE!
ret = ptempNCAP->GetNCAPBlockState(ncapState);

// print state information
if (ret.majorReturnCode == MJ_COMPLETED)
    cout << "NCAP Block State = " << ((ncapState == 4) ? "NB_INITIALIZED" : "OTHER") << endl

// process events in the ieee environment
NCAP_WAIT_ON_EVENTS
```

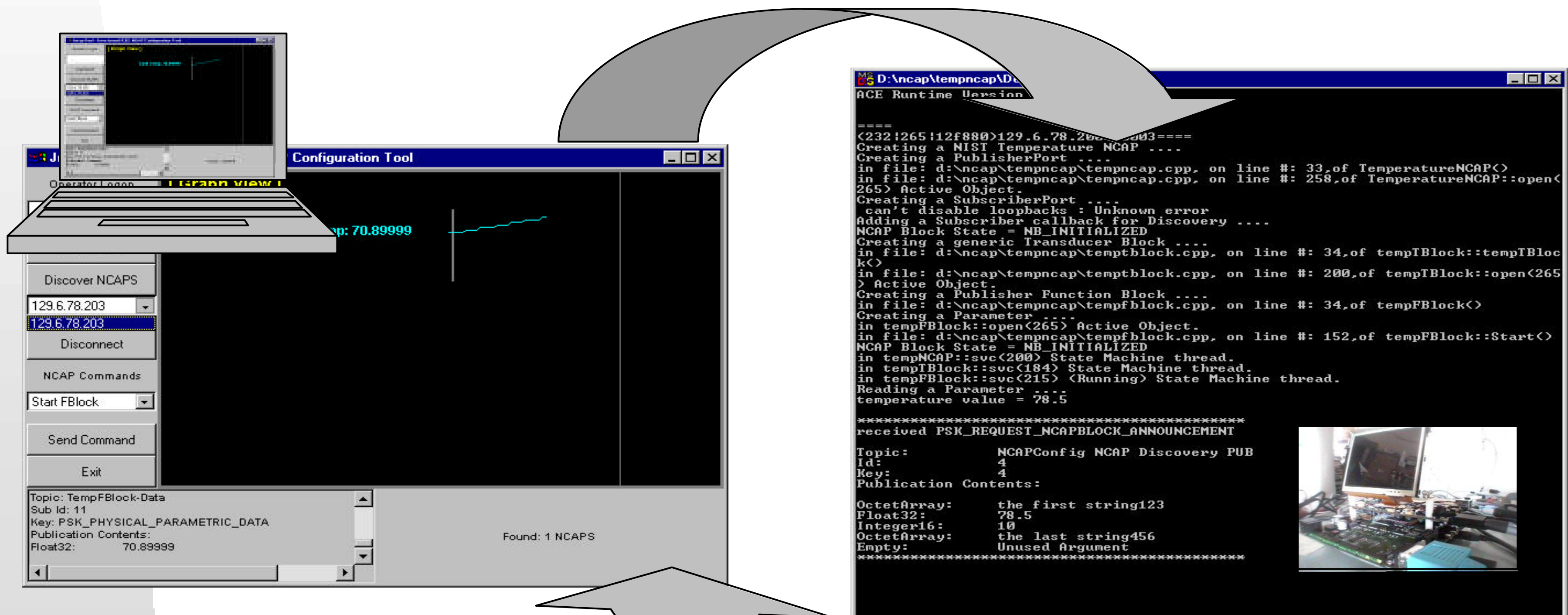
# A C++ IEEE 1451.1 Application (Ports)

- Ports are used throughout the application in the NCAP, Function Block and Transducer Block
- This Port snapshot is used to multicast temperature data from within the Function Block
- A Publication Topic and Key are used to define this on the network
- Instantiate the Port, in this case an Event Generator Port
- Set timer parameters for how often to “fire” this event

```
void tempFBlock::SetupFBlockData()
{
    // declare a return code object for ieee 1451 method calls
    OpReturnCode ret;

    ObjectTag mtag("multicast tag");

    // create a general multicast object dispatch address for this NCAP
    ObjectDispatchAddress mcast_addr(   ACE_DEFAULT_MULTICAST_ADDR,
                                        ACE_DEFAULT_MULTICAST_PORT,
                                        mtag);

    // create a set of object properties including the ncap multicast address
    ObjectProperties    mcast_props("temp fblock", "owner", mcast_addr, "fblockobj");

    ////////////////////////////////////////////////////////////
    // Build PSK_PHYSICAL_PARAMETRIC_DATA
    ////////////////////////////////////////////////////////////

    // create a publisher topic for FBlock Data
    topic = new PublicationTopic("TempFBlock-Data");

    // create a event publisher port for FBlock Data
    cout << "Creating a EventGeneratorPublisherPort ...." << endl;
    fblock_data_pub_port = new IEEE1451_EventGeneratorPublisherPort(mcast_props, *topic);

    // set the PubSub key to PSK_PHYSICAL_PARAMETRIC_DATA, meaning data
    fblock_data_pub_port->SetPublicationKey(PSK_PHYSICAL_PARAMETRIC_DATA);

    // event pub has been retrofitted (non-std) to include time-based event generation

    // create a time period for publishing
    TimeRepresentation timerep;
    timerep.seconds = 1;
    timerep.nanoseconds = 0;

    // set the timer in the eventpublisher
    fblock_data_pub_port->SetTimer(timerep);
}
```

# Executing an IEEE 1451.1 Application

- An embedded Temperature NCAP Application is running from a remote location on the NIST Intranet
  - As part of the system configuration, a NIST developed Java tool on a Notebook issues a discovery multicast, finds the NCAP, and starts the remote NCAP's Function Block
  - The remote NCAP Function Block responds by publishing temperature data every second as the Java tool records the information

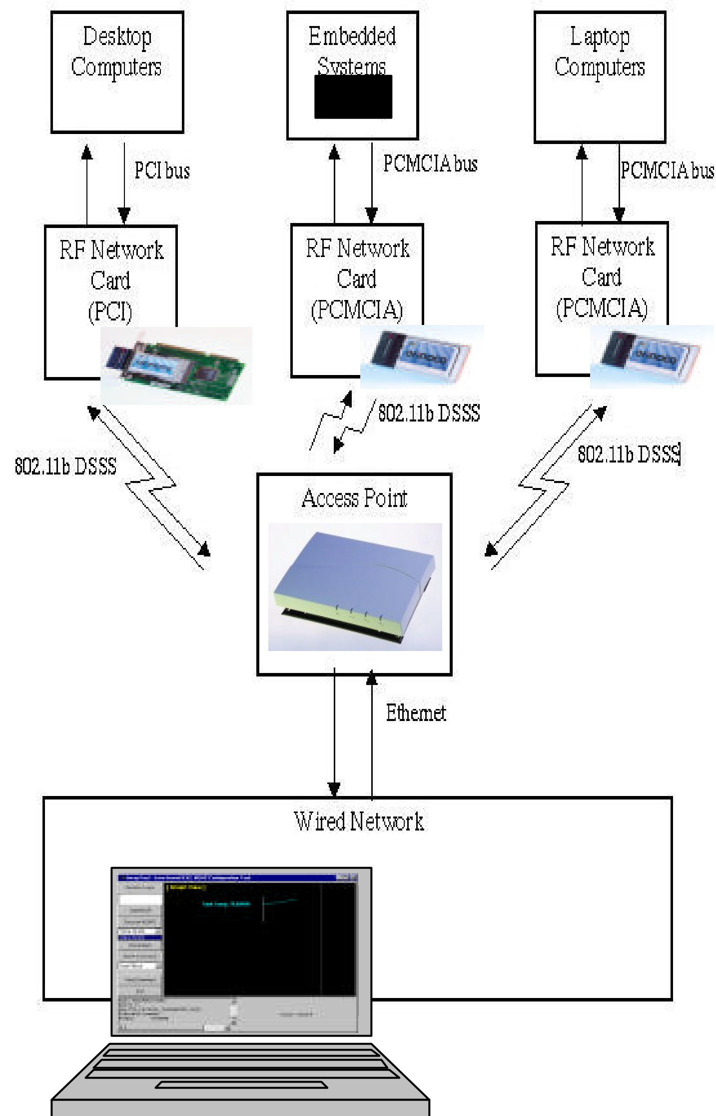


# Part 4: Using IEEE 1451.1 in a Wireless Environment

- The NIST C++ IEEE 1451.1 reference implementation uses TCP/IP as its underlying control network.
- From TCP/IP, IP multicast and TCP unicast features are used to implement publish/subscribe and client/server, respectively
- ACE is used to abstract the networking code from the application; therefore it is highly adaptive to various protocols
- Wired 802.3 Ethernet has been used primarily for testing. No changes were needed in ACE to support this protocol.
- Wireless 802.11b (11Mbps) Ethernet has also been used for testing. Again, no changes were made to ACE as the TCP/IP protocol is compatible with both 802.3 and 802.11b physical mediums.

# Using IEEE 1451.1 in a Wireless Environment

- Testing scenarios included using a wired subnet connected to a wireless extension of the subnet
- Wireless extension uses an Agere (formerly Lucent) Orinoco AP-1000 dual card "access point"
- Range extender antennas are also connected to the access point and each PC-CARD
- Each node on the wireless side executes an IEEE 1451.1 NCAP application
- Java Configuration tool executes beyond the wireless net on the wired subnet



# Using IEEE 1451.1 in a Wireless Environment

- Limited testing scenarios confined to the office environment at NIST
- Experience has not been good due to range problems within our building.
- Structural barriers within the walls severely restrict the range available from NCAP to access point
- Building constructed in the 60’s of solid concrete block with concrete floors, offices
- Range using a wireless subnet is no greater than 20-25’ with antenna
- High-end of 11 Mbps range only reached with 15’
- All applications worked as advertised albeit with limited range.
- Multicasting through the bridge needed to be configured through the access point
- Noticeable delay for multicast depending on how configured at access point

# Summary

NIST CENTENNIAL 1901-2001

- IEEE 1451.1 is a comprehensive and large standard that adequately addresses the smart transducer industry need for portability and network independent access.
- The standard however in addressing all the facets of the smart transducer is complex and quite large.
- NIST has embarked on implementing a good deal of the standard with emphasis on getting the communication and infrastructure code in place in order to start using the code.
- Choosing and implementing the standard with a solid object-oriented framework such as ACE provides a robust environment for real-time network communication.
- Migrating the implementation to other middleware such as CORBA for heavier weight uses will be reasonable to do
- Several projects at NIST will use the implementation for supporting manufacturing related activities

# Summary (cont)

- Continued testing in the wireless space is required to gauge the effectiveness of the implementation.
- Bluetooth trials are forthcoming; however, the lack of multicast support will severely impact the applications – continued research here is a must
- Other lightweight middleware packages are going to be isolated – XML and SOAP, etc; however, these protocols do not support asynchronous messaging or publish subscribe in efficient ways
- Slimmer implementations of the IEEE 1451.1 will need to be experimented with for use with the smaller micro platforms.

# References

- More information about ACE can be found at: www.cs.wustl.edu/~schmidt/ACE.html
- **IEEE 1451.1-1999 Standard for a Smart Transducer Interface for Sensors and Actuators - Network Capable Application Processor Information Model 2000, ISBN 0-7381-1768-4**

NIST CENTENNIAL 1901-2001