Multi-Network Access to IEEE P1451 Smart Sensor Information Using World Wide Web Technology

Rick Schneeman
Computer Scientist
Sensor Integration Group
NIST
rschneeman@nist.gov

Kang Lee
Leader
Sensor Integration Group
NIST
Kang.Lee@nist.gov

Abstract

NIST and a consortium of companies are organizing a public demonstration of the features of IEEE P1451, a *Draft Standard for a Smart Transducer Interface for Sensors and Actuators*. The demonstration is set in a multivendor transducer and control network environment during the May ‘97 Sensors Expo in Boston. This paper presents the design of the demonstration system which uses a server architecture based on the industry standard UDP/IP protocol and the gateway software module that bridges the UDP Ethernet-based network and the specific control network technology. The definitions of a common set of ASCII-based message protocol units are defined to allows cross network and transducer device communication. Combining these approaches using Internet-based technologies such as World Wide Web browser software, Java, HTML Web pages, and the Internet Protocol suite, the demonstration shows the functionality and interoperable capability of the proposed interface for connecting sensors and actuators to control networks. It also shows that the implementation of the interface based on the draft specification is realizable.

1. Introduction

During the May, 1997 Sensors Expo in Boston, the National Institute of Standards and Technology (NIST) would like to demonstrate the salient features of IEEE P1451, a *Draft Standard for a Smart Transducer Interface for Sensors and Actuators* (herein referred to as Draft), in a multivendor transducer and control network environment. Both parts of the Draft, P1451.1[1] and P1451.2[2], are currently moving through the IEEE balloting process with approval expected for one and possibly both parts of the Draft within the next 6 to 8 months. To highlight the popular features of the Draft in a multivendor supported atmosphere, NIST has organized a public demonstration with participation from sensor and transducer manufacturers, control network vendors, system integrators, and users.

This paper outlines a strategy to demonstrate the interoperable capability and functionality of the IEEE P1451 in an interesting yet educational venue that leverages the phenomenal growth and excitement of the Internet and World-Wide Web. Using World Wide Web-based browser technology as the IEEE P1451 information delivery and dissemination tool, Sensors Expo attendees will be able to access IEEE P1451 devices from a convenient, ubiquitous, and understandable vantage point. For example, we envision the NIST demonstration area at Sensors Expo to include multiple laptops or personal computers that would allow conference attendees to launch and use a World-Wide Web browser to navigate IEEE P1451 specific links from the NIST IEEE P1451 Demonstration Web-Site Home page. Using the Web in combination with browser technology, one can quickly imagine interesting scenarios of how to demonstrate IEEE P1451 to the public.

The NIST demonstration to be held at the Sensors Expo ’97 in Boston and Detroit, as well as the ISA Tech ‘97 in Anaheim, will provide attendees with hands-on interactive information regarding the emerging IEEE P1451 Standards Draft capabilities and usefulness to the transducer community. Using a familiar Internet-based World

Wide Web browser as the information delivery tool, various standardized information from IEEE P1451 networked devices can be queried. Figure 1 illustrates the overall configuration and topology of the demonstration.

The demonstration provides access to all networked P1451 devices via a common 10 megabits per second (Mbps) Ethernet backbone. This Ethernet backbone will provide the demonstrations Intranet at Sensors Expo that will allow various control network vendors and transducer manufacturers to “connect” their technology into one complete demonstration setting. All devices and control networked P1451 standardized nodes are connected to the network, providing direct access via the NIST Internet browser shown in Figure 1. This is the first time one demonstration has shown the integration of both P1451.1 and P1451.2 based devices in a multitransducer and multinetwork environment. At any one time, several Internet-based browsers may execute queries to the P1451 devices. Therefore, multiple demonstration computers will be available for attendees to use and discover P1451 capabilities and benefits.

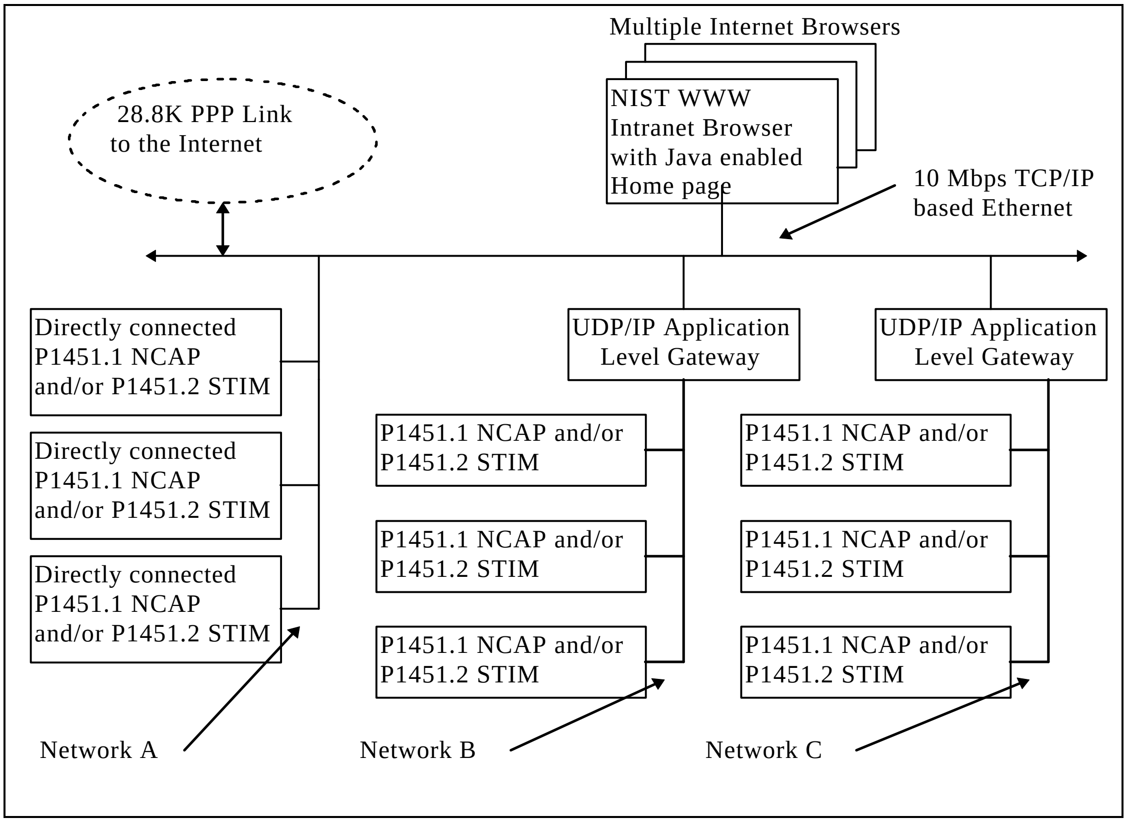
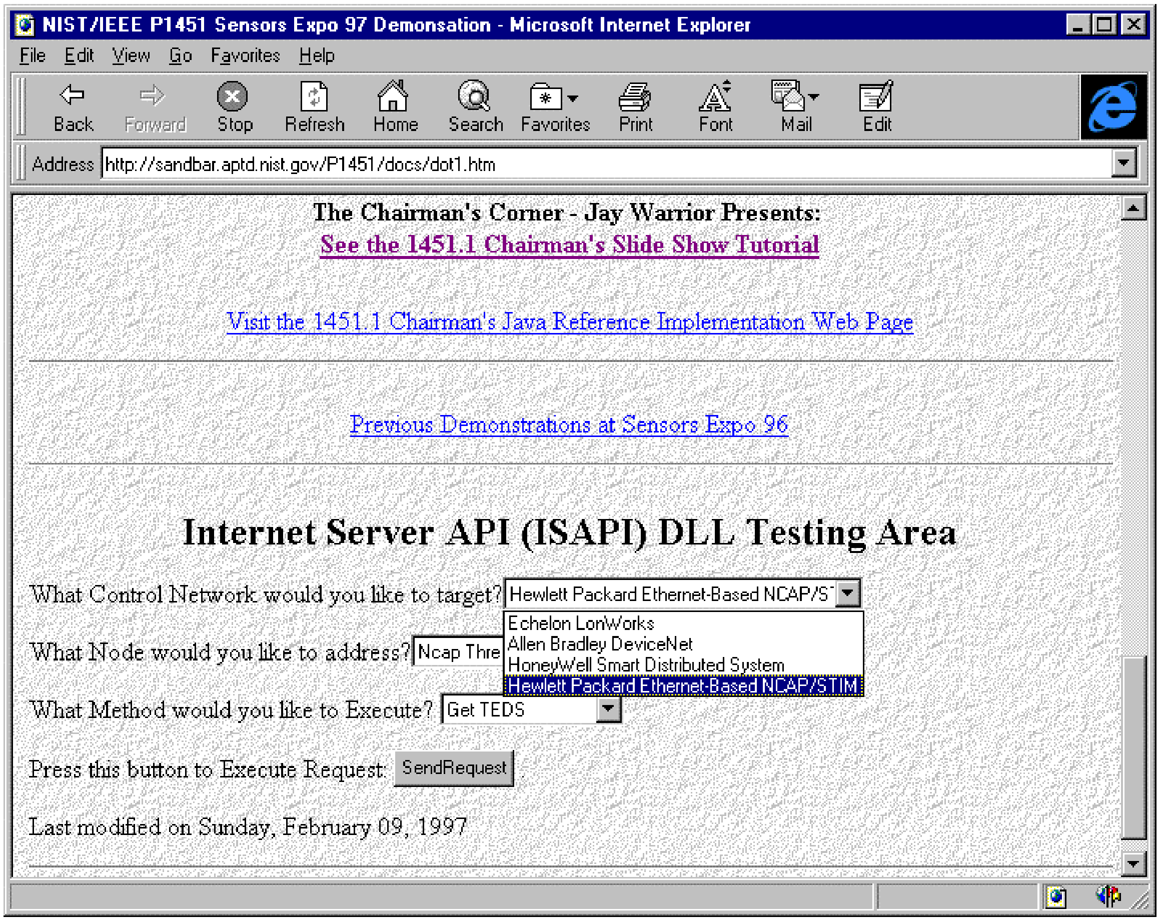

Multiple Internet Browsers
28.8K PPP Link to the Internet
NIST WWW Intranet Browser with Java enabled Home page
10 Mbps TCP/IP based Ethernet
Directly connected P1451.1 NCAP and/or P1451.2 STIM
Directly connected P1451.1 NCAP and/or P1451.2 STIM
Directly connected P1451.1 NCAP and/or P1451.2 STIM
UDP/IP Application Level Gateway
UDP/IP Application Level Gateway
P1451.1 NCAP and/or P1451.2 STIM
P1451.1 NCAP and/or P1451.2 STIM
P1451.1 NCAP and/or P1451.2 STIM
P1451.1 NCAP and/or P1451.2 STIM
P1451.1 NCAP and/or P1451.2 STIM
P1451.1 NCAP and/or P1451.2 STIM
Network A
Network B
Network C

Figure 1: Conceptual Layout of the NIST Sensors Expo '97 Demonstration.

The NIST Internet browser will use Java applets embedded in a Hyper Text Markup Language (HTML) based home page as the platform Expo attendees will use to query information from various IEEE P1451 transducer devices. Various concepts from the P1451 draft standard will be shown throughout the demonstration. Through the use of various user interface components (i.e., buttons, pull-down menus, check/radio boxes, and text edit dialogs) from the Java-enabled home page, a series of queries will be constructed and sent to the appropriate control network or any directly attached device via the common Ethernet-based backbone. Figure 2 illustrates an example of what a possible home page would look like to construct such queries.


NIST/IEEE P1451 Sensors Expo 97 Demonsation - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Favorites Print Font Mail Edit

Address http://sandbar.aptd.nist.gov/P1451/docs/dot1.htm

The Chairman's Corner - Jay Warrior Presents:
See the 1451.1 Chairman's Slide Show Tutorial

Visit the 1451.1 Chairman's Java Reference Implementation Web Page

Previous Demonstrations at Sensors Expo 96

Internet Server API (ISAPI) DLL Testing Area

What Control Network would you like to target? Hewlett Packard Ethernet-Based NCAP/S

Echelon LonWorks
Allen Bradley DeviceNet
HoneyWell Smart Distributed System
Hewlett Packard Ethernet-Based NCAP/STIM

What Node would you like to address? Ncap Thre

What Method would you like to Execute? Get TEDS

Press this button to Execute Request: SendRequest

Last modified on Sunday, February 09, 1997

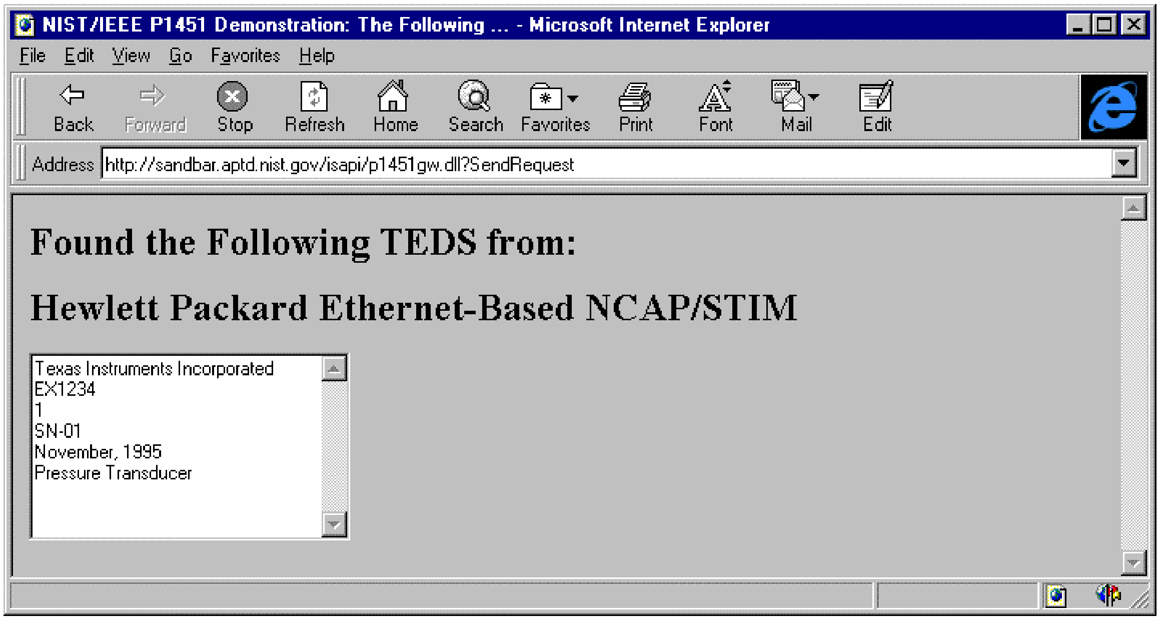
Figure 2: Internet Explorer interface illustrates the query construction process.

To facilitate sending the queries and receiving the responses from the remote devices, the industry standard User Datagram Protocol over the Internet Protocol (UDP/IP) will be used as the common transport over the Ethernet medium[1]. Each application level ***gateway***[2] or directly attached device (i.e., the vendor may have a device which can directly attach to the Ethernet backbone), shown in Figure 1 will have the ability to receive these queries from the Internet browser software and route the request to the appropriate node/device. The end result being that the Java Applet will update the Internet browser user interface with the requested information. This information will take the form of (1) an actual one-shot (i.e., not a continuous reading) sensor reading from the device, (2) retrieve the Transducer Electronic Data Sheet (TEDS) from the device, (3) initiate sending data events (continuous sensor readings) from the device, and finally (4) change the rate at which the sensor is sampling the sensed entity. Figure 3 illustrates what the Microsoft*** Internet Explorer 3.01 user interface would like after the results of the META-TEDS query was requested from the remote Network Capable Application Processor (NCAP)/Smart Transducer Interface Module (STIM)[3].

[1] UDP/IP refers to the connectionless mode protocol used in the Internet. This mode is opposite of the Transmission Control Protocol (TCP) connection-oriented mode that most think of when referring to Internet protocols. UDP provides an unreliable “datagram” based transport to the client server entities. This means that messages or datagrams can be sent at anytime in any order to any host without making an actual connection to that host. This mode of service works well in short data transaction situations such as this demonstration environment requires.

[2] The term gateway in this demonstration refers to a software process that will translate messages received from the Ethernet into requests for the specific control network or directly connected device’s own native communication format.

[3] The examples shown here are for illustrative purposes only.



Address http://sandbar.aptd.nist.gov/isapi/p1451gw.dll?SendRequest

Found the Following TEDS from:

Hewlett Packard Ethernet-Based NCAP/STIM

Texas Instruments Incorporated
EX1234
1
SN-01
November, 1995
Pressure Transducer

Figure 3: Retrieving the TEDS from the Remote NCAP/STIM.

Notice the Uniform Resource Locator (URL) address in Figure 3 has called a C++ Dynamic Link Library (DLL) based on the Microsoft Internet Server API (ISAPI), and uses the Microsoft Foundation Class Extensions to support Internet HTTP based application gateways. In addition to the Microsoft method shown here, queries can be constructed using Java Applets, or various scripts based on the Common Gateway Interface (CGI), Perl, or any other secure method for invoking services on the local host.

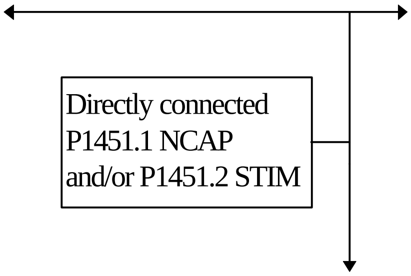
For the purpose of this demonstration multiple NCAPs are needed for each network to accommodate all the sensors from the manufacturers. The final component layout for the demonstration shown in Figure 1 includes a 28.8K or greater speed Internet Point-to-Point Protocol (PPP) modem link to either the NIST Gaithersburg Campus or to a local Boston-based Internet Service Provider (ISP). This link may be available for companies that wish to provide links in the NIST Home Page to other in-house corporate demonstrations of P1451 functionality. If companies supply in-house based P1451 URLs, then it is expected that attendees could use this link to get external access to the Internet. In addition, some collaborators at the Sensors Expo may want to provide URLs for the demonstration Web Page. This would provide a means of accessing them if they too supported such an Internet connection. It is important to note that at the moment the internal Boston based demonstration that NIST shows at the Expo will be a separate entity (Intranet). It is not expected that the participants will use this external link for demonstration purposes, unless explicitly requested. Using the Intranet based solution minimizes network logistics and reliance on third-party communications support.

Demonstrating the IEEE P1451 capabilities with the networked environment as we have defined here will essentially require four major areas of hardware and software to come together. These include: (1) a software gateway, or protocol translation software layer, (2) an agreed upon common messaging syntax or protocol for network requests, responses, and event messages, (3) the actual software and hardware implementation of the NCAP, and 4) the STIM.

We begin our discussion with the software portion of the demonstration by focusing on the capabilities and implementation guidelines of the gateway. We then provide a discussion on the various interactions and expected startup behavior between the gateway and the nodes on each network. Finally, a detailed discussion outlining the protocol encoding rules for the common messaging system is provided.

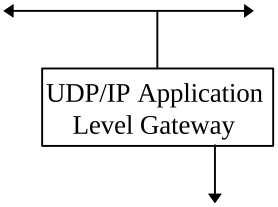
2. Demonstration Software Requirements and Guidelines

The capability to bridge a control network vendors technology with the UDP/IP based Ethernet environment is needed as part of this demonstration. We discuss in this section what we consider to be the minimal implementation guidelines for successful gateway deployment. The gateway, as discussed here will consists of a UDP/IP based connectionless server implementation. Using various socket-layer application programming interface calls, this style of server can be implemented rather quickly. It is our goal in this section to provide enough information and guidance to allow skilled network application writers to develop and integrate such a gateway.



Notice in the box to the left that the P1451.1 NCAP and/or STIM on Network A from Figure 1 does not require a gateway. In this case, each node that is directly attached from Network A may communicate directly with the Internet browser software without an intermediary. The directly connected devices do not need gateways; however, they MUST implement the equivalent functionality of a *half-gateway*. A *half-gateway* is defined as only the UDP/IP message receipt, translation, formatting, and subsequent reformatting of the response for transmission. However, developers of these types of devices should still follow the gateway implementation details found below. On the other hand, control network vendors that do not physically attach directly to the Ethernet (i.e., Network B and C shown in Figure 1) must use alternative means of providing access from the UDP/IP based network to their respective control network technology or transducer hardware. This alternative can be provided by means of a software gateway or router mechanism. The following discussions are provided to help the developers implement this capability in relation to the demonstration requirements described in this document.

2.1 Gateway Design Concepts



Any combination of hardware and software can be used to provide this gateway capability. It is envisioned that a PC or Notebook computer would be used as the physical gateway, providing both an Ethernet card interface for the UDP/IP network side and a PC Card (PCMCIA), ISA BUS, or equivalent control network specific card to their respective network. Software modules would then need to be developed to bridge the two disparate networks forming a gateway. In the box shown to the right, the gateways from Figure 1 are referred to as application-level gateways because the UDP packets would be received, parsed and interpreted at the application level. Communicating the request through to the control network would most likely be done also using an application-level access mechanism implemented as dynamic link libraries (DLL), dynamic data exchange (DDE) servers, or whatever vendor specific product allows application-level node access from their control network board in the PC.

NCAP suppliers could provide any node-based device on their network, access mechanism, and data acquisition scheme they deem suitable to fulfill the service requirements that the messaging interface requires (i.e., get sensor reading, TEDS, etc.). They would then simply develop a small piece of code (gateway) executing on a PC to interact with their hardware nodes to request this information. Most vendors we have come in contact with have this capability. Referring to Figure 4, Echelon is used as an example to illustrate this point.

Echelon developers would then receive these requests and in turn inject the LON-specific equivalent of this using the LON API calls to interact with their nodes. Once the information from the nodes has been gathered, they would then repackage this information in a form corresponding to the ASCII response packet format we have defined and ship it out to the client awaiting the return. Once the client program (Java Applet) gets this information from its UDP socket, it them takes the string information and places it in a standard Web page format.

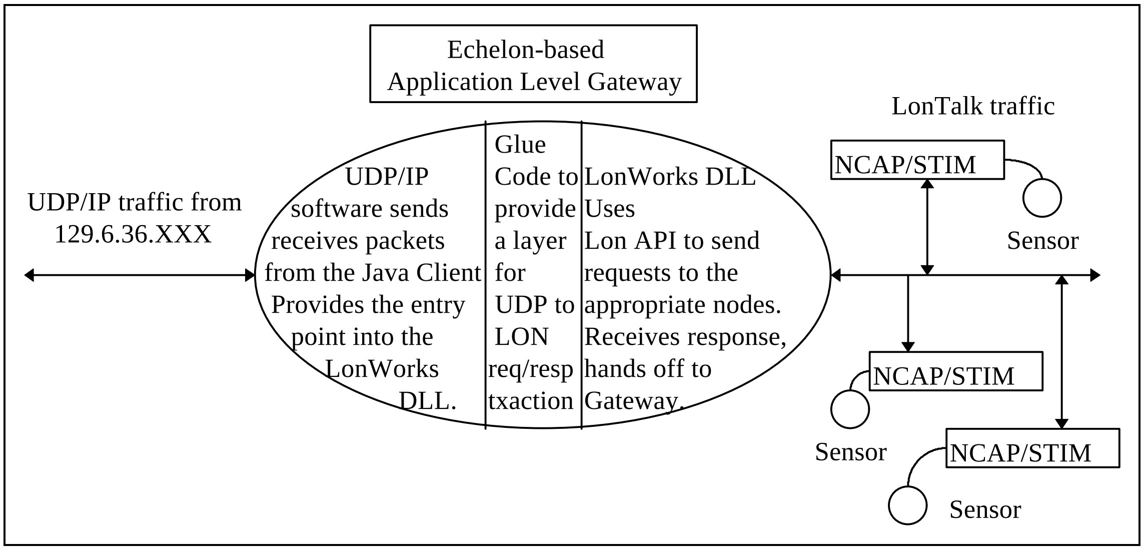


Figure 4: Example of a possible Echelon Application-Level Gateway.

Essentially, the gateway software would need to be able to act as a connection-less iterative server. An iterative server receives a requests from a client and handles it directly without creating any concurrent processing activities such as a thread, another process, or co-routine to handle the transaction. This means that requests are handled serially, new requests are not honored until the outstanding requests completes. This was purely a technical decision based on the types of transactions likely to occur and to reduce the amount of software development overhead on the collaborators, as the iterative approach is less time consuming to implement. The type of server implemented, whether iterative or concurrent, makes no difference to the client software. In fact if collaborators would rather implement a concurrent server for reasons of efficiency or general preference, then they are free to do so.

In any event, the server must have the ability to create a UDP socket at a port address assigned for this demo and to receive UDP/IP packets that contain standardized formatted ASCII command strings from the NIST home page user interface. Once the UDP/IP server has been established and is executing, then a standard set of agreed upon ASCII commands can be sent back and forth to request the desired P1451 services from the control network or nodes. The gateway must receive and interpret the packet, parse the standard ASCII command, and route the *action* to the appropriate node on their network or to the directly attached transducer hardware. Conversely, the gateway or directly attached device must be able to accept the response that their hardware or control network node provides and package the response into a UDP/IP packet and send the response back to the requester. The messaging system we have defined here effectively provides a client/server relationship between the Java program executing as part of the Internet Browser environment and the gateway or directly attached devices located on the demonstration's Intranet. As long as the gateway or physical device can (1) setup a UDP/IP server, (2) accept UDP/IP packets, and finally (3) parse and respond accordingly to the requests, then the demonstration entities should interact well with this environment.

Implementers are free to develop the gateway module in any language or environment that find it easiest or quickest to do so. As long as the environment presents an available UDP network transport socket endpoint for a connectionless server, the language environment used is irrelevant. That is, standard UDP/IP socket layer calls for connecting to the Gateway server should be provided. This means, Distributed Component Object Model (DCOM), Network Dynamic Data Exchange (NetDDE), Object Linking and Embedding (OLE), or other methods for tunneling data through a TCP/IP network will not be acceptable. We do not want to provide client support for multiple data exchange protocols. We only want to be presented with a socket level interface for server communication. For the demonstration, Microsoft's 32-Bit implementation of the TCP/UDP/IP protocol stack referred to as WinSock Version 2 in a Windows 95/NT 4.0 environment is used for the network software development. Figure 5 shows the relative positioning of both TCP and UDP in relation to the seven-layer Open Systems Interconnection (OSI) protocol hierarchy[3]. Notice that the Internet Protocol suite only defines four layers of the OSI model.

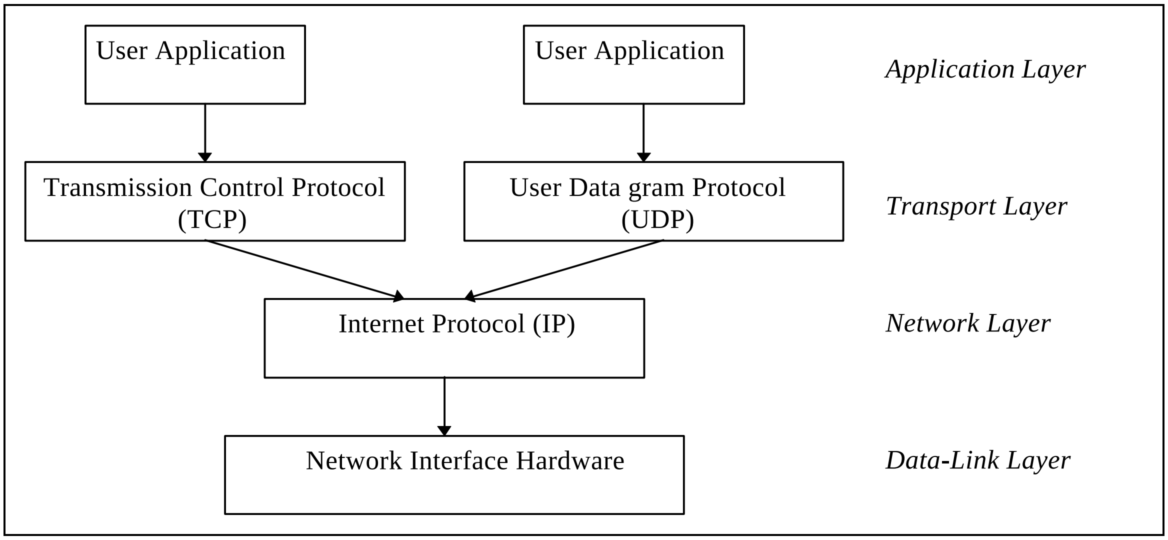


Figure 5: UDP placement in the four Layer OSI Protocol Model.

In addition, the Java software we are developing for the client side communication uses the Java Socket API. In a Windows NT Workstation 4.0 setting for instance, the Java Socket layer would also use the Microsoft WinSock Version 2.0. If time permits, because we have experience in using UDP/IP in a variety of settings, we will offer assistance to any vendor who requires help integrating the skeletal server framework described in this document. However, integration with vendor specific control network software is outside the scope of this assistance provision. The next section provides a more detailed discussion of the implementation of the gateway.

2.1.1 Gateway Implementation Information

In general, each gateway or directly connected device will be assigned an unique IP address in order to participate in the demonstration Intranet. NIST will assign these addresses as we need to coordinate the addressing semantics within the client/server Java communication software located on the Demonstration Web server's home page. We will allocate addresses for the local demonstration Intranet using a subnetted Class B address of the form 129.6.36.XXX, where XXX is the host identifier that we will assign to each participant. There is no particular reason for using the 129.6.36 subnet address, only that this is the address we use at NIST for our local network and it is therefore convenient for us during testing. We envision assigning IP addresses(not presently used) such as the SDS gateway will be 129.6.36.215, the DeviceNet gateway will be 129.6.36.216, and so on. Of course, when developers are testing their gateway code within their organizations, they can choose any address that is appropriate for their domain, local network, or individual machine. Until such time as the actual integration, then the IP addresses will need to be changed to reflect the actual demonstration scenario. Nodes located on the particular control network on the other side of the gateways will not be individually assigned IP addresses, as it will be up to the gateway software using internal tables to maintain the mappings between node addresses referred to in the ASCII commands and the node id's on their physical networks. Only gateways and directly attached devices to the Ethernet will have IP addresses assigned to them.

Below in Figure 6, we outline the basic steps required to develop an iterative server using UDP/IP. This code is written in the C language and conforms to the Berkeley Software Distribution's (BSD) version of the socket layer application programming interface.

```
#define  UDP_SERVER_PORT          4000                 /* demo assigned port address */
int         sock, status;
int         client_len;
struct    sockaddr_in         *client;
struct    sockaddr_in         server;

            sock = socket( AF_INET, SOCK_DGRAM, 0);              /* datagram socket*/
            server.sin_family   = AF_INET;
            server.sin_addr.s_addr       = htonl(INADDR_ANY);
            server.sin_port                  = htons(UDP_SERVER_PORT);   /* demo port */
            bind(sock, (struct sockaddr *) &server, sizeof(server));

            for (; ; ) /* forever loop *.
            {
               receivefrom(sock, mesg, MAXMESG, 0, client, &client_len);
               /* Process message */
               sendto(sockfd, mesg, n, 0, client, client_len);
            }
```

Figure 6: Setting up a Server-based UDP socket connection.

Implementers may use other versions of the socket implementations just as we use the Microsoft version in our work for this demonstration. In any event, Figure 6 describes the most basic steps needed to provide the iterative server capability in the gateway. The server simply uses the *receivefrom()* and *sendto()* socket calls to receive and send messages, respectively. There are a couple subtleties that need to be noted in this example. Namely, in the socket() call, notice that the SOCK_DGRAM parameter is being used. This creates a handle to a UDP-based or datagram socket, as opposed to using SOCK_STREAM for a TCP-based connection-oriented socket. In addition, when the UDP/IP server software residing on the gateway calls the *bind()* socket call shown in Figure 5; notice that it uses the demo-assigned port address of **UDP_SERVER_PORT** (or 4000) in the *bind()* call that we defined in this demonstration to be the P1451 service port[4] This way the client software will not have to broadcast or guess what port will be in use for the P1451 service. UDP servers setup a socket for accepting messages using the socket layer application programming interface calls outlined in Figure 6. After this code is executed, the server will wait in an infinite loop for any client message on the IP address given to the machine and at port 4000. The next section discusses further the startup behavior that is expected in the server portion of the gateway.

2.2 Gateway Startup Behavior

Upon initial execution, the UDP/IP based gateway or directly attached device that emulates the gateway network functionality should create a UDP socket, bind to it at the servers address and well-known port address (4000). Now in server mode, the gateway should simply wait to receive any message from a Java-enabled Web Browser. There will be no connection phase, only messages from the client will be received and messages sent by the server will go to the destined client. When a message is received by the server using the *receivefrom()* system call (or similar incarnation) the server must process the message iteratively (serially) in the order it is received. Once a response can be obtained (events not included) by the device from the server, a response message is placed on the outgoing socket using the *sendto()* call. Graphically, the client/server interaction using the UDP based socket system calls is illustrated below in Figure 7.

[4] Example code is provided as an informative tool.

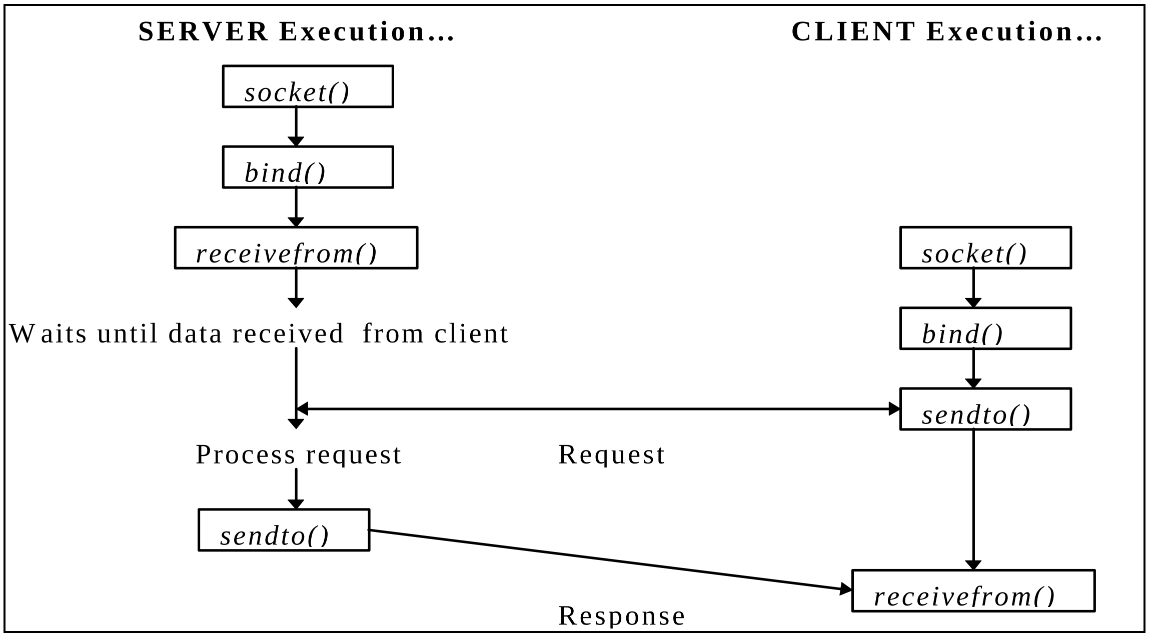


Figure 7: Server startup using Socket calls in a connectionless Protocol mode.

2.3 NCAP/STIM and Gateway Interaction

Understanding what UDP server messages will be received, and the interactions needed from the NCAP and/or STIM to respond to these messages is critical for developers to know what kinds of functions their nodes will need to support during the start-up phase and throughout normal operations. Summarizing the possible message requests from the client helps to frame the discussion about required services needed by the nodes. Requests from the client fall into four general categories:

1. Request for polled sensor readings (not continuous readings).
2. Requests for various TEDS structures from the transducers.
3. Requests to change event generation time offsets and set sampling rates.
4. And finally, requests for starting and stopping the nodes event generation mechanism.

These services (if all supported) can be done in various places behind the server scenes. Clearly, most of the service implementations should closely follow the P1451 standardized practices within reason for this demonstration's time frame. However, the implementation details are left up to the NCAP and/or STIM implementers and gateway developers. The goal in this section is to give the developers a heads up on what behavior the gateway should expect from the NCAP and/or STIM in order to fulfill the requests coming in from the clients.

The NCAP or node implementations for this demonstration clearly should be very simple to design and implement. Upon startup, the node should be in a ready state willing to respond to request from the gateway. The actual interaction after the UDP message is decoded from the gateway and injected into the various control networks is beyond the scope of this document. However, a simple one-to-one relationship with requests at the server to requests on the particular control network is a realistic one. Assuming this relationship exists then, the node should be capable of handling a request for a sensor reading. After executing the actual reading from the transducer, the node should send the answer/value back to the server (the server most likely will act as a participating node on the specific control network). The server in turn will need to translate the network specific answer/value into the appropriate response ASCII message as we have defined. After the translation process is finished and the message is ready for transmission, the server will then send the message back to the client who originated the request.

The server will always have the correct client return addressing information to return the response because when the original request was received, the receive-from address is provided as part of the *receivefrom()* UDP socket call. Some maintenance of this address may occur for out of sequence processing of messages (i.e., events), but for the most part, a tight receive/send loop can be maintained as discussed in the iterative server implementation approach from the previous sections.

The server may receive requests for TEDS information. To provide this, the node targeted to retrieve the TEDS must be capable of accessing the STIM using the 10-wire interface protocol defined by the P1451.2 draft. In addition, the TEDS format must also be preserved by the node in order to ship the correct image back to the server. The server must then reformat the answer into the TEDS encoding message as discussed in Section 3.2.1.

For this demonstration, there are only three requests for operational changes that each node application needs to be aware of. The node application should be able to respond to a change of start time offset in the case of event generation. The node must also respond to changes in the sampling rate requests for transducer read requests. And finally, the node's application must be capable of starting and stopping the generation of events. The startup state of the node should again be in the ready state. The state of the nodes will not be checked by the clients' software. It will assume it can communicate requests or it cannot. The node should not pump out any extraneous information upon start-up to the server that might be construed as response message processing as this may confuse the state of the demonstration. For example, earlier demonstrations would upon power-up, send out the TEDS and continuous sensor readings to the destination node. In this particular poll system of message requests, we do not want the nodes to respond unless directed via the request procedure outlined here. There is no other node-application specific start-up information required. Events will be generated by the node only upon requests to do so; therefore, no additional traffic on each control network or among each node should be taking place during this demonstration.

3. Common Messaging Support and Implementation

In this demonstration, the query process will construct and send ASCII messages using a single communication scheme across all devices. This will provide a common reference point for the integration of the network technologies. We chose to send UDP/IP based ASCII messages over the Ethernet to communicate with the various gateways/devices. This message format and complete descriptions of their meaning are discussed below.

The set of standard commands that the device or gateway must respond to will be few. In the hopes of keeping this demonstration simple yet effective, we envision only four standard types of commands will be required. The commands include: (1) request for sensed data from the transducer (i.e., temperature, pressure, etc.), (2) request for different types of TEDS from the transducer, (3) request to change or alter the transducers sampling rate (i.e., setting time period between sensor readings), and finally (4) starting and stopping event generation. In this demonstration, there will only be one type of event generated, a data event. This means, when a request from the client is received to start generating events, the sensor or actuator will take a new reading. At every interval defined by either the default or user specified start time offset variable, a sensor data event will be generated by the NCAP to the gateway. The gateway will in turn translate the reading from the NCAP into the appropriate event in ASCII encoding and send the packet to the client.

In addition to these types of requests, the encoding of the responses returned from the devices will also need to be standardized as the Java program executing from the Internet browser will need to accept the response, interpret what was received from the gateways/devices, and update the user interface accordingly. Using an ASCII encoding scheme will minimize the machine dependencies found in this multivendor hardware environment. Formatting of TEDS fields and sensor readings will all be encoded as ASCII strings.

Because vendor software for this demonstration is only required to respond to standardized ASCII string messages, they will not have to be concerned with how the final user interface is organized, operated, or displayed. This decouples the user interface and other software activities we are doing at NIST from the software development approach the vendors must accomplish to integrate their demonstration hardware and software.

3.1 ASCII Message Format

An ASCII command message encoding will be used to communicate information between the browser software and the end-system gateways/devices. We have defined below the minimum required set of ASCII messages that the direct-attached device and gateway implementers must support. The encoding rules described below are a modified subset of a richer superset developed originally by Dr. Jay Warrior, Leader of the IEEE P1451.1 Working Group. Essentially, the ASCII messages will take the form of tokenized strings that can easily be parsed using either String class methods in C++ or by using the *strtok()* C language standard library routine[4]. All messages will be encoded using a series of tokens. The table below illustrates the tokenized message format used in this demonstration[5].

MSG_TYPE	NODE_ID	TRANS_ID	METHOD	ARGC	ARGV

Table 1: Tokenized ASCII Message Format

Each message will provide a message header starting with a message type MSG_TYPE token. This token describes what type of message it is. The three message types that are legal in this demonstration include: REQ, RSP, and EVT which represent a request, response, and event message, respectively. A NODE_ID provides addressing information for the server to target the request to. The NODE_ID is assumed to be zero based, meaning 0 is the first node on the network. For directly attached devices, the NODE_ID token is extraneous but mandatory. IP addresses will be assigned to all directly accessible devices. Therefore, it is assumed that directly connected NCAP and/or STIM devices implicitly contain one node, and no more levels of addressing indirection are necessary. Specific channel information when requested will be encoded using a channel number in the ASCII header of the message; therefore the IP address and node addressing granularity is reasonable enough for accessing the desired end point.

All requests sent to directly connected nodes will be assigned the NODE_ID of 0 (zero) by default. After the NODE_ID, the message must contain a transaction identifier, or TRANS_ID. This is a locally generated random number that is encoded as an ASCII string. This number is generated by the client's sending side (i.e., the Internet Browser software) and is used to match responses with requests. For consistency, the transaction identifier must also accompany the event message, and it must be the same as the original event start request message transaction identifier.

Following the transaction identifier, the message must also contain a method identifier or METHOD. This token establishes what action needs to be taken on the server. The method identifier typically needs arguments associated with it to distinguish what parameters may accompany the method. The total number of arguments to be associated with the method are encoded in the ARGC (argument count) token. The arguments following the ARGV (argument vector) token will provide the type encoding for each argument pair along with the actual argument encoded as a string. For example, assume a method required two integers as arguments, 5 and 6, then the ARGC would equal 2 and the ARGV token would consist of two pairs of tokens describing each parameter or argument. The encoding for the two integers would then look like: INT 5 INT 6. It is requested that all ASCII characters be encoded in their capitalized form shown in this document.

3.1.1 Data Type Encoding Rules

[5] This is a representation of the tokens that delimit the string; it is ***not*** the actual message sent. Actual ASCII messages are shown in *italics* in sections 3.1.2 to 3.1.4

All data is transferred as ASCII encoded strings. To permit a range of data types and values to be transferred as strings, the following encoding rules shall be followed in order to convert the data type into its ASCII string representation[5]. Currently, it is envisioned that only a few type encoding for this demonstration will be required.

INT

The integer token type INT as briefly described earlier provides the encoding hint that an integer is the next argument in the request or response message. An integer value shall be encoded by the ASCII string representation of its decimal value. An optional negative (-) character may precede the string to represent a negative value. There will not be any encoding tokens used for unsigned integers in this demonstration.

STRING

In case a string is required or sets of text such as needed for the TEDS, then a type designator of STRING will be used.

FLOAT

Also, during responses, most sensor readings will be encoded with corrected float values; therefore a FLOAT type is used as well. A float value shall be encoded as the string representation of the decimal number followed immediately by a decimal point character "**.**" and optionally preceded by a negative character (-) for describing the negative value domain.

BOOLEAN

True and false conditions in the request/response message shall be encoded using a zero (0) to represent a false condition, or with a one (1), representing a true Boolean condition. Most Boolean encoding in this demonstration are used to designate failure or success during certain request messages.

At this time, these are the only data types that will be used during this demonstration. The complete message protocol, encoding rules, examples, and the required messages for this demonstration will be described in detail below. We will begin the discussion by detailing all the required request messages along with their descriptions for this demonstration. Each message breakdown and discussion will provide a table format that introduces the actual ASCII message with the token delimiters as table headings. Actual ASCII messages as they would be encoded in the message are italicized for clarity. A brief description of the message will follow as text below the table.

3.1.2 Request Messages (REQ)

There is a small set of request messages that will be needed. The messages shown here are simply ASCII representations of the P1451.1 draft standard application programming interface subset of concepts. Therefore a great deal from P1451.1 is used here to execute the various requests on the P1451.2 compliant hardware. Several past demonstrations used methods similar to these for low-level transducer reading of the sensor data, accessing the TEDS, and turning on or off the sensor sampling engine. Therefore it may be possible to reuse some prior demonstration node software and design techniques, as these areas of the standard have remained reasonably constant over time. For this demonstration, the server process must support the following defined request messages:

Request for sensor data (reading) using the **IO_READ** ASCII METHOD.

MSG_TYPE	NODE_ID	TRANS_ID	METHOD	ARGC	ARGV
REQ	*1*	*82347843*	*IO_READ*	*2*	*INT 0 INT 1,2, or n*

The METHOD *IO_READ* Transducer Block message format used here requests that a new sensor reading on NODE_ID *1* be sent back to the requester (client). If the device is an actuator, then the last value sent to the actuator will be returned to the client. Again, the transaction identifier TRANS_ID is a random number generated from the client and will be propagated back to the client in the outstanding response message by the server copying the identifier to the response(s) message just prior to transmission. This is a one-shot read; the value (pressure, temperature, etc.) is expected to be a corrected reading so that the client will not have to manage correction engines or require parsing the Calibration-TEDS for each transducer manufacturer. The arguments in this particular *IO_READ* message mean the following: the first of the ARGV pair, *INT 0*, indicates to the server process that this will be a sensor data read. This argument is constant for the sensor read request. The second ARGV pair, *INT 1, 2, or n* argument is not constant, in that it represents the channel number that the sensor reading will be taken from. Its values can range from channel 1, channel 2, or channel *n*; where *n* represents a channel number (greater than zero) of a multichannel STIM device.

Request for Meta-TEDS using the **IO_READ** ASCII METHOD.

MSG_TYPE	NODE_ID	TRANS_ID	METHOD	ARGC	ARGV
REQ	*1*	*82347843*	*IO_READ*	*2*	*INT 1 INT 0*

The METHOD *IO_READ* Transducer Block message format used here requests that the Meta-TEDS on NODE_ID *1* be sent back to the requester (client). This request is relayed to the server process by encoding the arguments in the following manner: the first ARGV pair, *INT 1*, tells the server that some form of TEDS information is being requested. This command encoding is also constant. The second ARGV pair, *INT 0*, describes to the server what exact kind of TEDS is being requested; that is, a 0 (zero) represent the Meta-TEDS information which describes the overall sensors/transducer configuration that is attached to the STIM.

Request for Channel-TEDS using the **IO_READ** ASCII METHOD.

MSG_TYPE	NODE_ID	TRANS_ID	METHOD	ARGC	ARGV
REQ	*1*	*82347843*	*IO_READ*	*2*	*INT 1 INT 1,2, or n*

The METHOD *IO_READ* Transducer Block message format used here requests that the transducer Channel-TEDS on NODE_ID *1* be sent back to the requester (client). This request is relayed to the server process by encoding the arguments in the following manner: the first ARGV pair, *INT 1*, tells the server that some form of TEDS information is being requested. This command encoding is also constant. The second ARGV pair, *INT 1, 2, or n*, informs the server that the Channel-TEDS from one of the channels shown here is being requested; that is, only one value representing which channel is to be used to retrieve the Channel-TEDS is provided here. It is important to note that the channel numbers in relation to requesting TEDS is NOT zero relative; the first channel begins at one (1).

Request for Calibration-TEDS using the **IO_READ** ASCII METHOD.

MSG_TYPE	NODE_ID	TRANS_ID	METHOD	ARGC	ARGV
REQ	*1*	*82347843*	*IO_READ*	*2*	*INT 2 INT 1,2, or n*

The METHOD *IO_READ* Transducer Block message format used here requests that the transducer Calibration-TEDS on NODE_ID 1 be sent back to the requester (client). This request is relayed to the server process by encoding the arguments in the following manner: the first ARGV pair, *INT 2*, tells the server that some form of TEDS information is being requested. This command encoding is also constant. The second ARGV pair, *INT 1, 2, or n*, informs the server that the Channel-TEDS from one of the channels shown here is being requested; that is, only one value representing which channel is to be used to retrieve the Channel-TEDS is provided here.

Request for Changing Sensor Sampling Rate using the **IO_CONTROL** ASCII METHOD.

MSG_TYPE	NODE_ID	TRANS_ID	METHOD	ARGC	ARGV
REQ	*1*	*82347843*	*IO_CONTROL*	*3*	*INT 0 INT n FLOAT x*

The METHOD *IO_CONTROL* Transducer Block message format used here requests a change in the sampling rate used to read a sensed quantity from the transducer device on NODE_ID *1*. This request is relayed to the server process by encoding the arguments in the following manner: the first ARGV pair *INT 0* tells the server that a change in the sampling rate is being requested. This command encoding is also constant. The second ARGV pair *INT n* informs the server that the new sampling rate is being requested for the channel represented in the *n* parameter. The third and final *FLOAT x* argument represents the amount of time in *seconds* that the sampling rate is being changed to. The setting of the sampling rate will occur in units of 0.1 secs.

Request for Enable/Disable Sensor Events using the **ENABLE_OPERATIONS** ASCII METHOD.

MSG_TYPE	NODE_ID	TRANS_ID	METHOD	ARGC	ARGV
REQ	*1*	*82347843*	*ENABLE_OPERATIONS*	*2*	*INT 0,1 INT n*

The METHOD *ENABLE_OPERATIONS* event class message format used here requests that event generation on NODE_ID *1* should either be enabled or disabled. When a client sends this message, the user is requesting that the node start or stop the generation of events to the user interface. This request is relayed to the server process by encoding the arguments in the following manner: the first ARGV pair *INT 0,1* tells the server that some form of enabling or disabling of the event generation is being requested. If the ARGV pair is encoded as an *INT 0*, then the request is for disabling event generation from the transducer on channel *INT n*. If the first ARGV pair is encoded as an *INT 1*, then the request is for enabling the event generation at the current or default start time offset event update rate (2 *seconds*) illustrated below.

Note: *This method is supported if and only if the node/transducer device will support data event generation. If the node/device does not support this feature, then a response from the server indicating this to the client shall be sent back to the client. The encoding of this condition in the response message will be discussed in the response encoding section of the document.*

Request to Set the Event Start time offset using the **SET** ASCII METHOD.

MSG_TYPE	NODE_ID	TRANS_ID	METHOD	ARGC	ARGV
REQ	*1*	*82347843*	*SET*	*2*	*INT 1 INT x*

The METHOD *SET* event class message format used here requests that the start offset time for event generation on NODE_ID *1* should be modified. This request is relayed to the server by encoding the arguments in the following manner: the first ARGV pair, *INT 1*, is used to differentiate that the *SET* request is for changing the start time offset in the event generation module. Because we are dealing with higher latency delays in the networks due to multiuser requests and contention, the event update rates need to be normalized into several seconds for this demonstration to work effectively and to be visually discernible. Therefore, the last *INT x* argument pair provides the *seconds* parameter for the event update rate. Note that the default start time offset event update rate is 2 seconds. Any request for a faster start time offset event update rate request should fail.

Note: *This method is supported if and only if the node/transducer device will support data event generation. If the node/device does not support this feature, then a response from the server indicating this to the client shall be sent back to the client. The encoding of this condition in the response message will be discussed in the response encoding section of the document.*

3.1.3 Response Messages (RSP)

There is a small set of response messages that will be needed. For this demonstration, the server process must support the following defined response messages:

Response from sensor data (reading) request using the **IO_READ** ASCII METHOD.

MSG_TYPE	NODE_ID	TRANS_ID	METHOD	ARGC	ARGV
RSP	*1*	*82347843*	*IO_READ*	*2*	*STRING ACTUATOR INT 0,1* *STRING SENSOR FLOAT 00.00*

The METHOD *IO_READ* Transducer Block response message returns the value of the new sensor reading from NODE_ID *1* back to the requester (client). If the device is an actuator, then the last value sent to the actuator will be returned to the client. The transaction identifier TRANS_ID is a random number generated by the client and placed in the packet header. This same transaction number should be propagated back to the client during any return of a response(s) message. The transaction identifier will be used to match responses with requests back in the client. The first ARGV argument in the response packet signifies what type of device the sensor is coming from. Again this is a quick way of providing this information to the client without having to parse through TEDS fields for this information. In this demonstration, it will be assumed that only sensor and actuator style transducers will be attached to the nodes. This information is encoded as *STRING ACTUATOR or STRING SENSOR* type pairs in the argument vector. The sensor value (pressure, temperature, etc.) following the type of transducer is expected to be a corrected reading and is characterized by encoding a *FLOAT* argument type followed by the floating point value encoded as a string. If the transducer is an actuator, then an INT type encoding with the binary value 0 or 1 can be used to describe the actuators condition. Zero (0) means the device has not been actuated, while a one (1) means the device has been actuated.

Response from any TEDS request using the **IO_READ** ASCII METHOD.

MSG_TYPE	NODE_ID	TRANS_ID	METHOD	ARGC	ARGV
RSP	*1*	*82347843*	*IO_READ*	*2*	*INT len STRING "meta-teds", or* *INT len STRING "chan-teds", or* *INT len STRING "cal-teds"*

The METHOD *IO_READ* Transducer Block response message returns any TEDS on NODE_ID *1* back to the requester (client). This requests is relayed to the server process by encoding the arguments in the following manner: the first ARGV pair *INT len* must be present to tell the client how long the stringified TEDS string is. The second *STRING* "*meta-teds*", "*chan-teds*", *or "cal-teds*" pair will contain a string-based description of any TEDS request that was returned by the device to the server. It is important to note that the actual encoding of the *STRING* TEDS argument in the message will ***not*** look like the quoted examples shown in the protocol specification table. That is, they are only placeholders for the actual string-based TEDS encoding discussed further in Section 3.2.1.

Note: *For a detailed breakdown on what the string-based encoding of the TEDS structures will look like, please refer to* *Representing the TEDS in the Response Packets* *section in 3.2.1.*

Response from Changing Sensor Sampling Rate using the **IO_CONTROL** ASCII METHOD.

MSG_TYPE	NODE_ID	TRANS_ID	METHOD	ARGC	ARGV
RSP	*1*	*82347843*	*IO_CONTROL*	*1*	*BOOLEAN 0,1*

The METHOD *IO_CONTROL* Transducer Block response message returns either a *BOOLEAN* argument type pair that describes the success of the change sampling rate request. The allowed values in the call include a zero (0), meaning the request was unsuccessful, or a one (1), meaning the change of sampling rate request succeeded. Because messages can get lost, corrupted, or destroyed when using the UDP protocol, responses may in fact never return to the client. This condition would also constitute failure in the client waiting for the success or failure status to return. In this case the client has the option of reissuing the change in the sampling rate request, or simply return a failure condition to the user interface that invoked the method after a reasonable time-out period.

Response from Enable/Disable Event request using **ENABLE_OPERATIONS** ASCII METHOD.

MSG_TYPE	NODE_ID	TRANS_ID	METHOD	ARGC	ARGV
RSP	*1*	*82347843*	*ENABLE_OPERATIONS*	*1*	*BOOLEAN 0,1*

The METHOD *ENABLE_OPERATIONS* event class response message returns either a BOOLEAN argument type pair that describes the success of the change sampling rate request. The allowed values in the call include a zero (0), meaning the request was unsuccessful, or a one (1), meaning the change of sampling rate request succeeded.

Note: *This method is supported if and only if the node/transducer device will support data event generation. If the node/device does not support this feature, then a response from the server indicating this to the client shall be sent back to the client.*

Response from a Set Event Start time offset request using the **SET** ASCII METHOD.

MSG_TYPE	NODE_ID	TRANS_ID	METHOD	ARGC	ARGV
RSP	*1*	*82347843*	*SET*	*1*	*BOOLEAN 0,1*

The METHOD *SET* event class response message returns either a *BOOLEAN* argument type pair that describes the success of the change sampling rate request. The allowed values in the call include a zero (0), meaning the request was unsuccessful, or a one (1), meaning the change of sampling rate request succeeded.

Note: *This response message is supported if and only if the node/transducer device will support data event generation. If the node/device does not support this feature, then a response from the server indicating this to the client shall be sent back to the client.*

3.1.4 Event Messages (EVT)

If the implementers choose to develop the hardware and software necessary to support the event generation capability in this demonstration, then they must support the event response message format packet defined here. In general, the event message is quite similar to the one-shot sensor read response packet defined in Section 3.1.3.

Event message using the **READ** ASCII METHOD.

MSG_TYPE	NODE_ID	TRANS_ID	METHOD	ARGC	ARGV
EVT	*1*	*82347843*	*READ*	*1*	*STRING ACTUATOR INT 0,1* *STRING SENSOR FLOAT 00.00*

The METHOD *READ* event message returns the current sensor or actuator operating value that has most recently been read. The first argument in the event message signifies what type of device the sensor is coming from. Again this is a quick way of providing this information to the client without having to parse through TEDS fields for this information. In this demonstration, it will be assumed that only sensor and actuator style transducers will be attached to the nodes. This information is encoded as *STRING ACTUATOR or STRING SENSOR* type pairs in the argument vector. The sensor value (pressure, temperature, etc.) following the type of transducer is expected to be a corrected reading and is characterized by encoding a *FLOAT* argument type followed by the floating point value encoded as a string. If the transducer is an actuator, then an *INT* type encoding with the binary value 0 or 1 can be used to describe the actuators condition. Zero (0) means the device has not been actuated, while a one (1) mean the device has been actuated.

Note: *The event message need only be supported if and only if the node/transducer device will support data event generation. If the node/device does not support this feature, then a response from the server indicating this to the client shall be sent back to the client. The encoding of this condition in the response message will be discussed in the response encoding section of the document.*

This completes the required set of request, response, and event messages that the directly attached device or gateways must support for this demonstration. In the next section we provide some guidelines on how to implement the ASCII messaging system in the gateways.

3.2 Implementing the ASCII Message in the Gateway

The encoding of the packet containing the ASCII strings can be done in a variety of ways in the gateway. Because the gateway in this demonstration is assumed to be a fully capable PC-based system, running a complete operating and development systems, there should not be any need to detail compute resource requirements to support the gateway concept. Each language supports various methods for allocating strings, arrays of characters or native byte streams. The most straightforward approach is to allocate a character array of large enough size. The benefit of this is of course is the reduction of network to host and host to network byte order translation overhead found in multibyte and numeric based message payload information. This reduces the errors and integration problems associated with the endianness of the machine architectures with which we must interact[6][6]. For this demonstration, we will be using message sizes of 1024. Therefore, the developer can allocate the message received or sent as:

char message[MAX_MESSAGE_SIZE];

where MAX_MESSAGE_SIZE would be a C/C++ language define for 1024 as:

#define MAX_MESSAGE_SIZE 1024

At this point, this message size represents the largest value we have tested on the receive. It represents the largest TEDS value we have seen on a single read of the socket. There may be others that have larger or smaller sizes, in which case the client may need to block on the receive (which is typically the case anyway). This should not affect the servers, they will simply send the message until there is no more data to send in the case of a larger than normal packet size. If there are many cases where the TEDS will be larger than 1024 bytes, then we can make the MAX size larger to accommodate the majority case. Any string copying procedures that utilize a character array can be used to create the string responses. Below, we begin a more detailed discussion with providing guidelines for the TEDS string encoding.

3.2.1 Representing the TEDS in the Response Packets

[6] Endianness refers to the way in which 8 bit bytes within a multibyte representation are stored in memory between different vendor's computer architecture. The two methods in widespread use today are LSB or least significant byte first and MSB or most significant byte first. The Intel architecture is a little endian, which means it stores multibyte fields in LSB order.

The representation of the TEDS fields in the response packets will need some additional processing from the server before they can be packetized and sent out over the network. There are currently three types of TEDS information that have been standardized. They include the Meta-TEDS, which describe all possible devices that have been connected to each channel of the NCAP and/or STIM node. A Channel-TEDS provides more in depth information concerning the specific device on that particular channel. Finally, a Calibration-TEDS provides any information that algorithms on the transducer might use to correct and calibrate the sensed data. Each TEDS structure is made up of a variety of data types, widths, and precision. In fact, the TEDS structure is a hybrid mix of floating point numbers, strings, arrays, different size integers, enumeration's, and character fields. Table 2 illustrates a snapshot of the Meta-TEDS structure and highlights the variety of data types used.

Field #	Description	Type	# bytes
	Data structure related information data sub-block		
28	Meta-Identification TEDS Length	U32L	4
	Identification related information data sub-block		
29	Manufacturer's Identification Length	U8L	1
30	Manufacturer's Identification	STRING	≤255
31	Model Number Length	U8L	1
32	Model Number	STRING	≤255
33	Revision Code Length	U8L	1
34	Revision Code	STRING	≤255
35	Serial Number Length	U8L	1
36	Serial Number	STRING	≤255
37	Date Code Length	U8L	1
38	Date Code	STRING	≤255
39	Product Description Length	U16L	2
40	Product Description	STRING	≤65535
	Data integrity information data sub-block		
41	Checksum for Meta-Identification TEDS	U16C	2

Table 2: Meta TEDS Structure is a hybrid mix of Data Types.

Although possible to encode the TEDS structures exactly as they are described in the standard for use with our network-based packet/message construction process, it would not be efficient to do so for this type of demonstration. Indeed, because each multibyte encoding would need to be translated into a neutral network byte ordering, a great deal of translation overhead would need to occur. Although the UDP protocol does maintain record boundaries, the result still would not be warranted here for our use. Therefore, the TEDS structures will be encoded by the server using some very simple techniques to essentially convert all entries in each TEDS structures into delimited strings. Using special sentinel strings embedded in the TEDS fields, the client software will be able to convert the strings to the appropriate format for presentation into whatever user interface form is finalized. We discuss this technique further below.

Each individual TEDS fields will be converted to their string equivalent, and then delimited by a End Of String (EOS) special sentinel string value. The EOS character string shall be three asterisks ("***") concatenated at the end of each TEDS field. All strings will then be concatenated together forming one long string vector. All TEDS fields must be present even if they are not used. Any nonimplemented or nonapplicable fields in any of the TEDS structures shall be replaced by the string value "N/A" followed by the EOS sentinel character string.

One last note concerning the use of the special character sequence string delimiter. The EOS mechanism was chosen as different vendors implement end of line conditions differently. In addition, what types of strings (i.e., Visual Basic VBSTR and C strings) that can be passed around can vary significantly as well. What this means is that when the client wants to format the information for display, it must discern what type of end of line/string designator is being used. Once this has been discovered, the developer must retrofit the string anyway with the inherent combination of carriage return linefeed, carriage return, or simply just linefeed, depending on the style of user interface component that the string information will be displayed in. So, we decided to harmonize the string translation process and require vendors to send us only the EOS versions of their string responses to quickly manipulate the string information for display or other purposes. Basically the developers/vendors are responsible for converting the format of the string such that it is compatible with their systems.

4. Summary

A design for demonstrating the IEEE P1451 Draft Standard concepts and capabilities in both a heterogeneous networking environment as well as multivendor transducer manufacturer setting is presented. To provide the infrastructure needed for the demonstration, several pieces of technologies are put into place. A server architecture based on the industry standard UDP/IP protocol suite needs to be developed by network vendors. The gateway software module that bridges the UDP Ethernet-based network and the specific control network technology also needs to be developed. Some vendors will choose to provide direct Ethernet-based transducer capabilities. In this case, only half of the gateway functionality will be needed. That half of the gateway will provide the server implementation that accepts, parses messages, and sends responses back to the caller.

To facilitate final integration, the definitions of a common set of ASCII-based message protocol units are defined. These messages provide for the request/response and the event capabilities. Implementing this common scheme for messaging allows cross network and transducer device communication to be possible.

Combining these approaches using Internet-based technologies such as World Wide Web browser software, Java, HTML Web pages, and the Internet Protocol (IP) suite, the goal of the demonstration - to illustrate the integration of multivendor networks and devices using a common set of interfaces that IEEE P1451 specifies - is achievable. By using an intuitive and interactive environment that the Internet provides, it should be very easy for the Expo attendees to grasp the P1451 functionalities and capabilities.

Ultimately, the demonstration shows companies and attendees the importance and benefits of the open interface standard for connecting sensors and actuators to control networks and that the implementation of the interface standard based on the draft specifications is realizable.

References

1. *IEEE P1451.1 D1.83, Draft Standard for a Smart Transducer Interface for Sensors and Actuators — Network Capable Application Processor (NCAP) Information Model*. Institute of Electrical and Electronics Engineers, Inc., New York, December 16, 1996.

2. *IEEE P1451.2 D2.01, Draft Standard for a Smart Transducer Interface for Sensors and Actuators — Transducer to Microprocessor Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats*. Institute of Electrical and Electronics Engineers, Inc., New York, August 2, 1996.

3. *UNIX Network Programming*. Stevens, W. Richard. Prentice Hall Software Series. Englewood Cliffs, New Jersey. 1990.

4. *The C Programming Language*. Second Edition. Kernighan, Brian W., Ritchie, Dennis M. Prentice Hall Software Series. Englewood Cliffs, New Jersey. 1988.

5. *IEEE P1451.1 Reference Implementation Project. TCP/IP Network Mapping Specification*. Warrior, Ph.D. Jay. Unpublished private communiqué. Draft 0.1. Rosemount Inc., Eden Prairie, MN 55344. 1996.

6. *Microprocessors: A Programmers View*. Dewar, Robert, B.K., Smosna, Matthew, McGraw-Hill Publishing Company. New York, New York. 1990.

Acknowledgment

During the development and integration of the reference implementation, several companies and individuals were very helpful in bringing this demonstration to fruition. We would especially like to thank the reviewers of this document for their advice and encouragement during this manuscript development process.

*** Certain commercial products are identified in this paper in order to adequately describe the proposed standard. Such identification does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the products identified are necessarily the best available for the purpose or the only ones that could be used.