IMTC 2005 – Instrumentation and Measurement
Technology Conference
Ottawa, Ontario, CANADA, 17-19 May 2005

Using a Commercial Framework to Implement and Enhance the IEEE 1451.1 Standard

Vítor Viegas[1,2], J.M. Dias Pereira[1,2], P. Silva Girão[2]
[1]Escola Superior de Tecnologia do Instituto Politécnico de Setúbal, 2910-761, Setúbal, Portugal
[2]Instituto de Telecomunicações, Av. Rovisco Pais, 1, 1049-001, Lisboa, Portugal
Phone: +351-265790000, Fax: +351-265721869, Email: vviegas@est.ips.pt

Abstract – *In 1999, the 1451.1 Std was published defining a common object model and interface specification to develop open, multi-vendor distributed measurement and control systems. However, despite the well-known advantages of the model, few have been the initiatives to implement it. In this paper we describe the implementation of a NCAP – Network Capable Application Processor, in a well-known and well-proven infrastructure: the Microsoft .NET Framework. The choice of a commercial framework was part of our strategy: to take advantage of several "of the shelf" technologies and adapt them to produce a NCAP prototype, called NCAP/XML. In addition, a solution to enhance the 1451.1 Std is presented by proposing a new format for inter-NCAP communication based on XML (eXtended Markup Language).*

Keywords – *IEEE 1451.1, NCAP, Smart Transducer, .NET Framework, .NET Remoting, Web Service.*

1. INTRODUCTION

We start by introducing the two foundations of our work: the 1451.1 Std [1] that describes the NCAP, and the Microsoft .NET Framework that we choose as an implementation base for our prototype. A complete description of the framework is beyond the scope of this paper, but those interested on this subject can easily find additional information in the literature [2].

1.1. IEEE 1451.1 Standard

The 1451.1 Std defines an object model suitable to represent any networked smart transducer. The model, illustrated in figure 1, must be implemented in a processor with two communication ports, one that interfaces the network and another that interfaces the transducer. The network processor, denominated NCAP, acts like a bridge between the smart transducer and the network. It exports the transducer functionalities over a standardized model and hides the details of implementation. The NCAP includes four major parts:

- **Operating System**: The operating system manages NCAP hardware resources offering an execution environment for applications. This environment provides basic services like memory management, interrupt handling, and multi-task execution. The execution of multiple concurrent tasks requires additional support that includes: a real-time scheduler to launch tasks according to their priority; mechanisms for inter-task communication like shared memory and message pipes; and primitives for inter-task synchronization such as mutexes and semaphores.

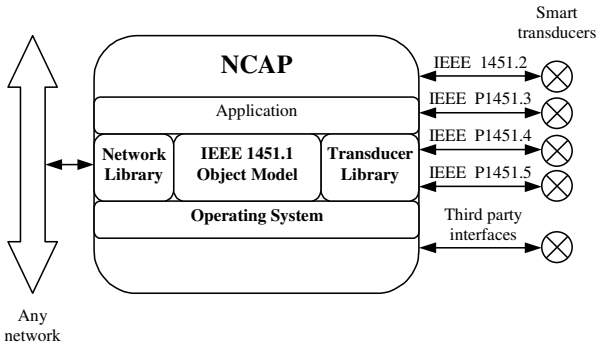

Fig. 1. IEEE 1451.1 model layout.

- **Network Library**: The network library provides services to handle network requests, including client/server and publisher/subscriber ports. Through these ports a remote client can call an object on the local NCAP and use its properties, methods and events just as simply as if the object was on the client itself.
- **Transducer Library**: The transducer library provides services to implement the interface between the NCAP and the smart transducer. The library includes functions to auto-detect the transducer, to read sensors, to update actuators, to configure trigger settings, to get status registers, to handle interrupts, and so on. The 1451.1 Std strongly recommends the implementation of generic interfaces but other alternatives still remain open. By generic interfaces we mean those described by the IEEE standards: Transducer Independent Interface (IEEE 1451.2), Transducer Bus Controller (IEEE P1451.3), Mixed Mode Interface (IEEE P1451.4), and Wireless Transducer Independent Interface (IEEE P1451.5).
- **IEEE 1451.1 Object Model**: The IEEE 1451.1 object model specifies the object classes used to design and implement application systems. As shown in figure 2, the object model is presented as a hierarchy of classes divided in three main groups: blocks, components and services. Three block classes are specified: the NCAP Block class that provides software interfaces for supporting network communications and system configuration; the Transducer Block class that provides standard software interfaces between transducers and application functions; and the Function Block class that encapsulates

application-specific functionality. Component classes provide common application building elements such as: structured information like measurements and files; collections of related application-specific objects; and actions with state where the action takes place over a relatively long period of time. Finally, service classes support communications between objects on distinct NCAPs and system-wide synchronization.

Root
- Entity
  - **Block**
    - NCAP Block
    - Function Block
    - Base Transducer Block
      - Transducer Block
        - DotX Transducer Block
  - **Component**
    - Parameter
      - Parameter With Update
        - Physical Parameter
          - Scalar Parameter
            - Scalar Series Parameter
          - Vector Parameter
            - Vector Series Parameter
      - Time Parameter
    - Action
    - File
      - Partitioned File
    - Component Group
  - **Service**
    - Base Port
      - Base Client Port
        - Client Port
        - Asynchronous Client Port
      - Base Publisher Port
        - Publisher Port
        - Self-Identifying Publisher Port
          - Event Generator Publisher Port
    - Subscriber Port
    - Mutex Service
    - Condition Variable Service

Fig. 2. IEEE 1451.1 Std class hierarchy.

1.2. Microsoft .NET Framework

The Microsoft .NET Framework, or .NET Framework for short, is a pre-fabricated infrastructure to develop desktop and Internet applications. The infrastructure is divided in four main parts as represented in figure 3:

- **Common Language Runtime (CLR)**: The CLR acts like a virtual machine that runs managed code, offering advanced features such as automatic memory management (also known as garbage collection),

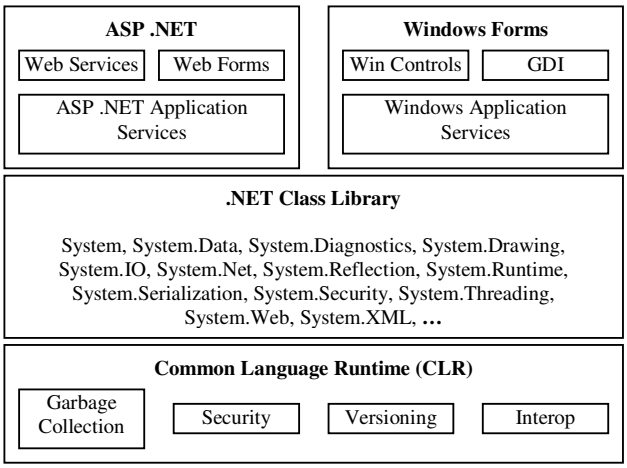

Fig. 3. .NET Framework components.

standardized versioning, code access security, and seamless inter-operability with Common Object Model (COM) components and Dynamic Link Libraries (DLLs). Managed code is written in a high level CLR-compliant language (such as Visual Basic .NET or C#) and then is compiled into an Intermediate Language (IL). The IL code itself can't run directly on any computer: it has to be interpreted by a Just-In-Time compiler, or JITter. This intermediate step slows down performance but provides .NET Framework with a certain amount of platform independence, as each platform can have its own JITter.

- **Class Library**: The class library contains the foundation classes used to build applications. The library has a tree structure where each class inherits the functionalities of its parent. The developer can extend the .NET Framework by creating custom classes that inherit from those of the pre-built tree.
- **Windows Forms**: Windows forms is a package that provides ready to use user interface elements to build powerful front ends for desktop applications. Such elements, also called Windows controls, include windows, buttons, dialog boxes, tree views, data grids, and so on. The package also includes a new version of the Graphics Device Interface (GDI) that enables applications to use graphics and formatted text on both a video display and a printer without worry about hardware details.
- **ASP .NET**: ASP .NET provides support to build and run Web applications. It's main functions include: pre-fabricated controls that do for HTML pages what Windows controls do for desktop applications; a Web server run-time environment that dynamically generates HTML pages in response to input received from the client; and advanced services such as data caching to speed up documents that are often downloaded, session state to personalize clients, and security to block malicious clients.

In the next section we stress the distributed technologies offered by the .NET Framework in order to use them afterwards for inter-NCAP communication.

2. MICROSOFT .NET DISTRIBUTED TECHNOLOGIES

The .NET Framework includes two distributed technologies to deal with remote objects: .NET Remoting [3-4] and Web Services [4]. In this section we present both technologies to understand where each one fits into the overall picture of a distributed application.

2.1. .NET Remoting

The .NET Remoting is the replacement of DCOM (Distributed COM, an older distributed technology released by Microsoft in the 90s for Windows environments). Like DCOM, .NET Remoting allows client applications to instantiate objects on remote computers and use them like local objects. However, unlike DCOM, .NET Remoting is simple to configure and easy to scale. It also uses a "pluggable" architecture that makes it very flexible.

As shown in figure 4, the communication process in .NET Remoting begins with a proxy object that "mimics" the remote object. When a method is called on the proxy, it calls the remote object, waits for the response, and then returns the result. In the background, the proxy communicates with a format layer that packages the client request or server response in the appropriate format. The format layer then communicates with the transport layer that transmits the information using the appropriate protocol. This layered structure is so flexible that developers can code their own layers and still use the same model without re-compiling the code. These custom layers can intercept the communication and provide additional services such as encryption, logging or compression.

The .NET Remoting supports three types of remote objects:

- **SingleCall**: Objects that are automatically created with every method invocation and live for only the duration of that method. The client can keep and use the same reference, but every call results in the creation of a new object. These objects are completely stateless.
- **Client-Activated**: When invoked for the first time, a new instance of the object is created and travels to the client becoming local.
- **Singleton**: This type of object retains state but shares it with clients. No matter how many clients are connected, there is always only one remote object instance.

Although .NET uses a "pluggable" architecture that allows custom layers, by default the .NET Framework includes two pre-built channels:

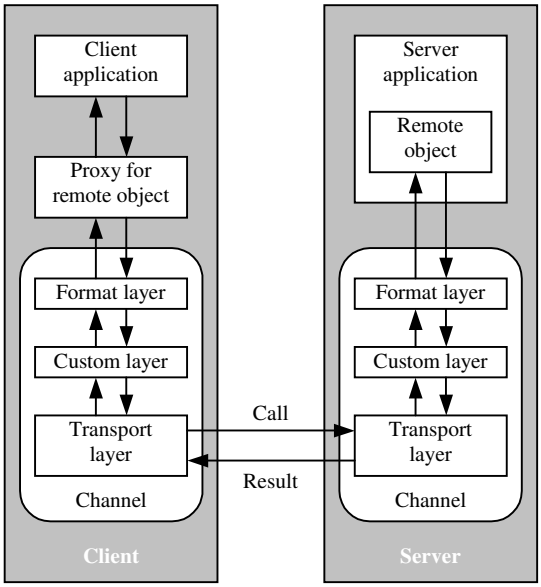

Fig. 4. .NET Remoting architecture.

- **TcpChannel**: This channel uses TCP/IP (Transport Control Protocol/Internet Protocol) with proprietary binary protocol as transport layer. It is compact and fast, ideal for using inside a firewall with maximum performance.
- **HttpChannel**: This channel uses TCP/IP with HTTP (Hypertext Transfer Protocol) as transport layer. It is a text based channel, not so compact, not so fast, but can cross firewalls making possible the communication across the Internet.

Both channels can choose between two pre-built formatters:

- **Binary**: The binary formatter serializes data to a compact, proprietary .NET format. This formatter offers the best performance but can be used only by .NET applications.
- **Extended SOAP** (Simple Object Access Protocol): Pure SOAP is a cross-platform XML-based plain-text format supported by the W3C (World Wide Web Consortium [5]), which includes all the major software companies around the world (such as Microsoft, Sun Microsystems and IBM, among others). However, .NET Remoting uses an extended SOAP formatter with proprietary logic that makes possible the reproduction of any .NET object but compromises inter-operability with non .NET frameworks. SOAP format (pure or extended) requires large XML text messages and can therefore reduce overall performance.

The main idea to retain is that .NET Remoting is more suitable as a high-speed solution for binary communications between proprietary .NET applications, usually over an internal network or inside the same machine.

2.2. Web Services

Web Services are, in simple terms, object methods exposed via HTTP using pure SOAP messages. Web Services are always stateless: the server instantiates an instance of the Web Service on demand and then destroys that instance after it finishes servicing a call. In fact, Web Services behave much like the SingleCall objects supported by .NET Remoting. The sequential communication steps, represented in figure 5, can be summarized as:

- Step 0: at programming time, the developer generates the proxy object code from meta-data that describes the Web Service.
- Step 1: at run time, the client generates the proxy object.
- Step 2: the client calls a method on the proxy.
- Step 3: the proxy converts the call to a SOAP message over HTTP and sends it to the server.
- Step 4: ASP .NET creates a new instance of the Web Service.
- Step 5: ASP .NET calls the specified method on the Web Service.
- Step 6: the Web Service returns results to ASP .NET and then dies.
- Step 7: ASP .NET converts results to a SOAP message and returns to client via HTTP.
- Step 8: the client receives the return value from proxy.

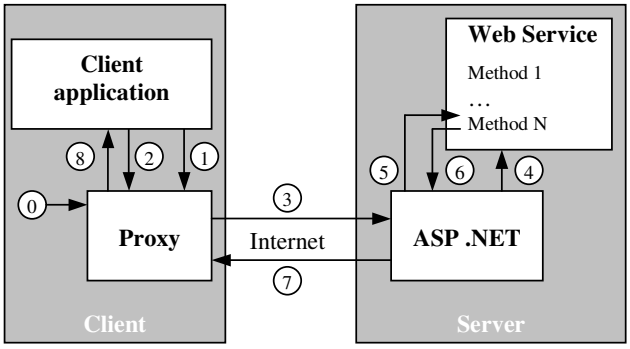

Fig. 5. Communication steps and Web Services architecture.

It's useful to evaluate the major differences between Web Services and .NET Remoting technologies:

- Web Services are more restricted than objects exposed over .NET Remoting (it isn't possible to create a Singleton or a Client-Activated objects). However, Web Services are generally easier to design and set up than .NET Remoting objects.
- Web Services support only pure SOAP message formatting, which targets cross-platform use. Any client that can parse the XML meta-data describing the service can connect over an HTTP channel and use that service, even if the client software is written in Java and hosted on a UNIX computer.
- Web Services are always hosted by ASP .NET, which means that they gain access to some powerful platform services, including data caching, session state management and security. These features, if required, can be very difficult to re-create by hand in .NET Remoting.
- Web Services work through a Web server and ASP .NET using the default HTTP channel (usually port 80). This means that clients can access Web Services just as easily as they can download HTML pages from the Internet. There's no need for an administrator to open additional ports on a firewall.

The main idea to retain is that Web Services are fine-tuned for Internet and cross-platform scenarios, offering a simpler model for distributed applications than the one provided by .NET Remoting.

3. NCAP PROTOTYPE

After studying the .NET Framework and its distributed technologies, we thought that it would be possible to use them in order to implement a NCAP prototype. This section describes that prototype, which we called NCAP/XML. We start by describing the general architecture of the prototype, presenting the software components that compose it and the way they establish connection. Then, we focus on the transducer interface, which we implemented using an innovative solution that enhances the 1451.1 Std. Finally, we present results and draw conclusions.

3.1. Architecture

As shown in figure 6, our prototype NCAP/XML is composed by three software components:

- **NCAP Engine**: The NCAP engine implements the IEEE 1451.1 object model, including blocks, components and synchronization services. Communication services are implemented using .NET Remoting and Web Services. The engine starts by creating the top-level NCAP Block and its child objects. The NCAP Block has its own thread that executes a script containing the user application (which implements control routines and provides "intelligence" to the system). All NCAP objects are published as Singleton objects that are shared between all connected clients.
- **NCAP Web Services**: This component is the Web interface of the NCAP application: it exposes NCAP objects as Web Services. Web Services are inherently stateless, but, in this case, they always return consistent results because, behind the scenes, they connect to Singleton objects. This extra layer introduces additional cross-process communication reducing performance.

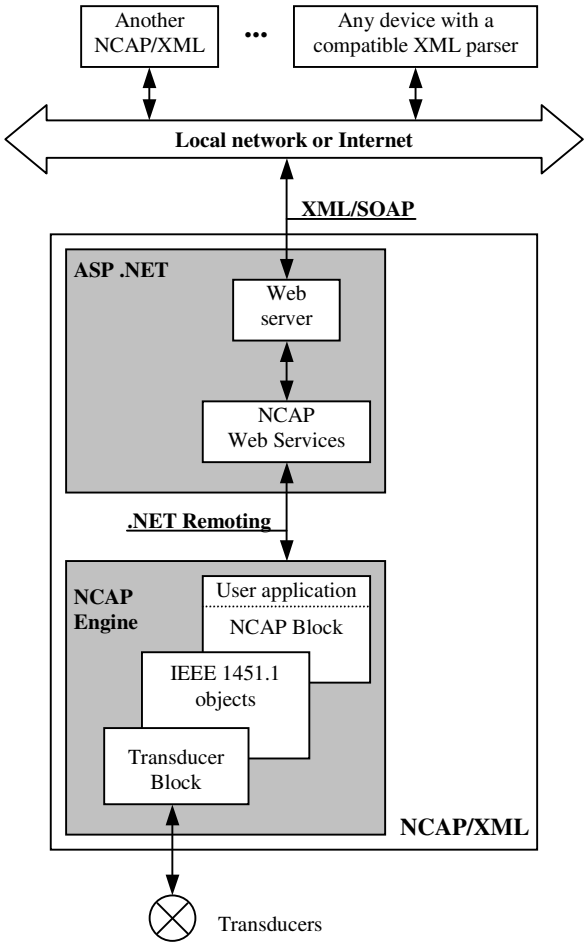

Fig. 6. NCAP/XML architecture.

However, this approach presents definitive advantages: it is open because it uses XML-based communication; it is Internet-oriented; and it is scalable by separating the NCAP application from the ASP .NET infrastructure.

- **Web Server**: The Web server acts like a "doorman", listening for HTTP requests on port 80 and routing them to the application that will serve them. It also handles security by preventing that unauthorized clients access HTML pages and Web Services. Our prototype uses the Web server provided by Microsoft, known as Internet Information Services (IIS).

3.2. Transducer Interface

Our prototype NCAP/XML doesn't implement any generic transducer interface proposed by 1451.X Stds. Instead, it implements a commercial interface focused on test and measurement systems, known as Interchangeable Virtual Instruments (IVI). We choose this interface for two main reasons: it extends our prototype into areas not covered by 1451 Stds; and it can simulate field signals making easier the development and debugging of our prototype.

The IVI interface is supported by the IVI Foundation [6], which includes all the major companies in the field of test and measurement systems (such as National Instruments, Keithley Instruments and Agilent Technologies, among others). The IVI interface is itself an example of inter-operability by proposing a software model that isolates user applications from measurement instruments. Two layers compose the software model: class drivers and instruments drivers. Class drivers encapsulate functions that are common to a specific class of instruments. So far, the IVI Foundation has defined eight instrument classes: power supplies, function generators, RF signal generators, Digital MultiMeters (DMMs), power meters, oscilloscopes, spectrum analysers and switches. Silently, the class driver calls the instrument driver, which in fact "talks" with the instrument. The instrument driver automatically handles low-level communication details. The IVI Foundation provides configuration tools that allow the developer to load and unload instrument drivers without changing class drivers. This way, the user application is always ready to run since it sees the same class driver, no matter the instruments connected.

As shown in figure 7, the NCAP/XML Transducer Block links to a DLL provided by National Instruments (NI) that implements the IVI interface for DMMs. This way, our prototype can call functions on the class driver and communicate with any DMM. At the same time, we can use the configuration tools provided by National Instruments [7] to configure instrument drivers and even simulate measurements.

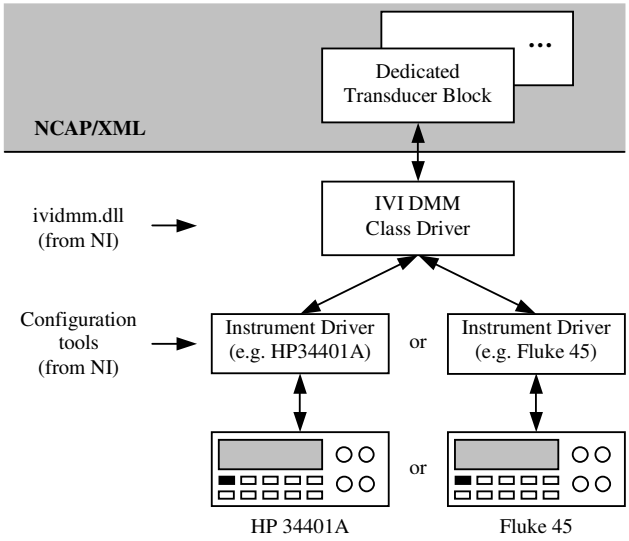

Fig. 7. Transducer interface.

3. RESULTS

The NCAP/XML prototype runs in Windows XP with IIS and .NET Framework v1.1 installed. To use the IVI interface we must install some libraries from National Instruments (namely, NI-VISA v2.6, IVI Engine v1.83 and MAX v2.2.0). The software was developed using Visual Basic .NET and Microsoft Development Environment 2003 v7.1.

As shown in figure 8a, we implemented the NCAP application as a console because it operates faster and requires very little user interface. Using a Web browser, we can access NCAP/XML (figure 8b), call a Web method (figure 8c), and get the result in the form of an XML-based document (figure 8d).

4. CONCLUSION

Our work demonstrates that is possible to implement a NCAP using the .NET Framework. This approach has many advantages because commercial frameworks have good development tools, are robust and are wide accepted.

Our prototype NCAP/XML is completely open because it uses pure SOAP messages to support communications. Any device with a compatible XML parser can communicate with our prototype.

Finally, NCAP/XML can communicate with virtual instruments because its Transducer Block is linked to an IVI interface. This is another example of inter-operability by mixing the 1451.1 Std with a foundation-based technology.

In the future we are planning to transfer the prototype to an industrial PC and test it in an industrial environment. It is our intention to implement and characterize wireless interfaces at the network and transducer levels.

REFERENCES

[1] "IEEE 1451.1 Standard for a Smart Transducer Interface for Sensors and Actuators – Network Capable Application Processor (NCAP) Information Model", IEEE, New York – USA, April 2000

[2] David S. Platt, "Introducing Microsoft .NET", 2nd Edition, Microsoft Press, Washington – USA, 2002

[3] David Curran, Andy Olsen, Jon Pinnock, "Visual Basic .NET, Remoting Handbook", Wrox Press, Birmingham – UK, 2002

[4] Mathew MacDonald, "Microsoft .NET, Distributed Applications: Integrating XML Web Services and .NET Remoting", Microsoft Press, Washington, 2003

[5] http://www.w3.org

[6] http://ivifoundation.org

[7] "Getting Started with National Instruments IVI", National Instruments Corp., July 2001

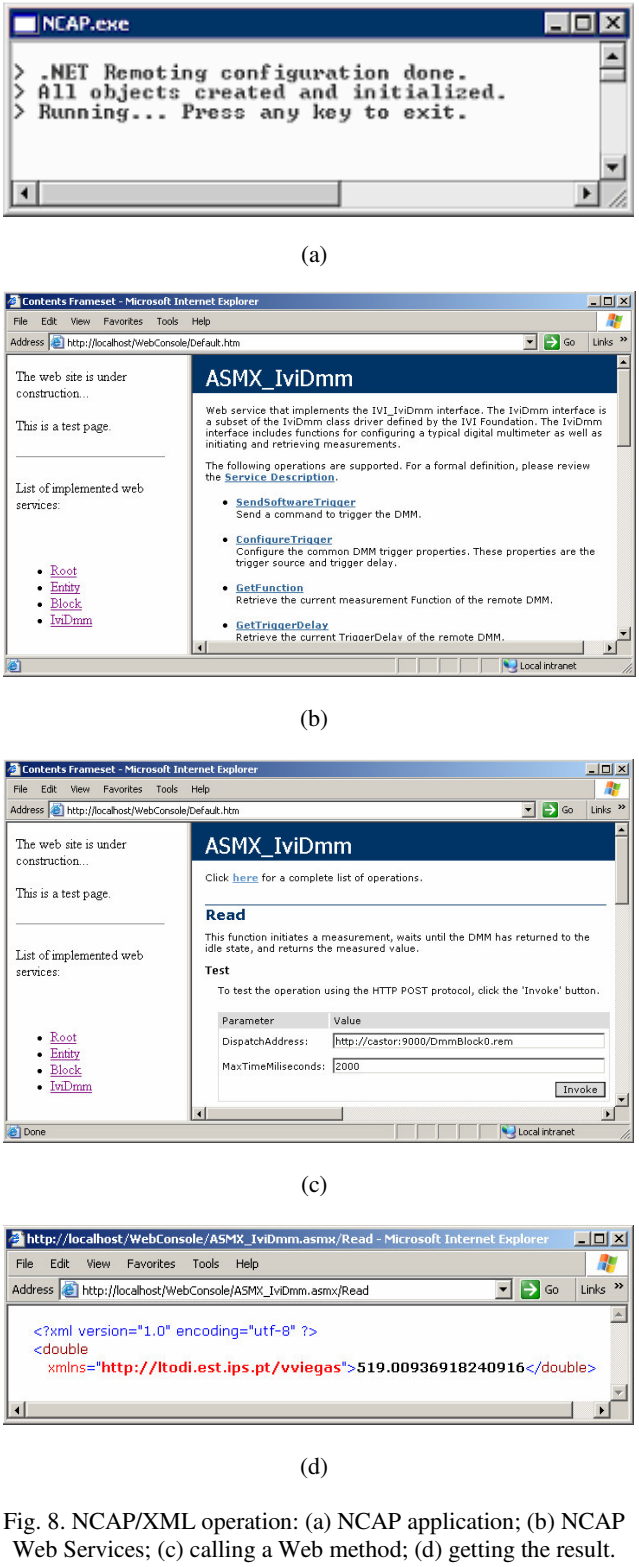

Fig. 8. NCAP/XML operation: (a) NCAP application; (b) NCAP Web Services; (c) calling a Web method; (d) getting the result.