RESEARCH REPORT

VTT-R-08256-08

# Embedded systems in portable devices – research report

Authors: Tero Haatanen, Kristiina Hytönen, Jarkko Leino, Mikko Metso, Timo Niemirepo

Confidentiality: Public

## Contents

## Abbreviations

| | |
|---|---|
| AN | Application Node<br><br>Application nodes are parts of a NoTA (sub)system that can interact with other nodes, but do not offer services of their own |
| SN | Service Node<br><br>Just as application nodes can be thought of as clients, service nodes are analogous to servers. Service nodes can accept connections from application nodes or other service nodes, and can also initiate contact with other service nodes |
| NODE | In this document, the term node alone can refer either to AN or to SN. |
| HIN, H_IN | High Interconnect.<br><br>The top layer of the interconnect stack, which communicates with the nodes in the network |
| LIN, L_IN | Low Interconnect.<br><br>The bottom part of the interconnect stack, responsible for communicating with the physical network |
| HIF | High Interconnect Interface.<br><br>The part of H_IN visible to application developers |
| SIS | Service Interface Specification.<br><br>A description of the services offered by a service node. |
| SID | Service Identifier.<br><br>A unique identifier for each service node that is registered in the network |

# 1 Introduction

The EMSYS (Embedded Systems in portable devices) project was started at VTT in order to deepen the know-how of today's software technologies in embedded programming. A new architecture called NoTA (Network on Terminal Architecture) was introduced to the public during 2008. It is anticipated that NoTA will gain wide acceptance among telecom industry partners in the near future. The purpose of NoTA is to seamlessly interconnect devices offered by different manufacturers in order to integrate them into a system consisting of a network of devices. The EMSYS project focuses on embedded systems, and specifically on applying the NoTA protocol stack to such systems.

NoTA is a service based modular device architecture framework [1] as shown in Figure 1-1. The Device Interconnect Protocol (DIP) is the backbone of the NoTA architecture. DIP is also modular and its parts are High Interconnect (H_IN) layer and Low Interconnect (L_IN) layer. L_IN provides a uniform socket based communication mechanism between any two network endpoints. H_IN handles service registration, discovery, access and security. H_IN uses L_IN through the L interface (L_IF). The application node (AN) and service node (SN) use the H_IN through the H interface (H_IF).

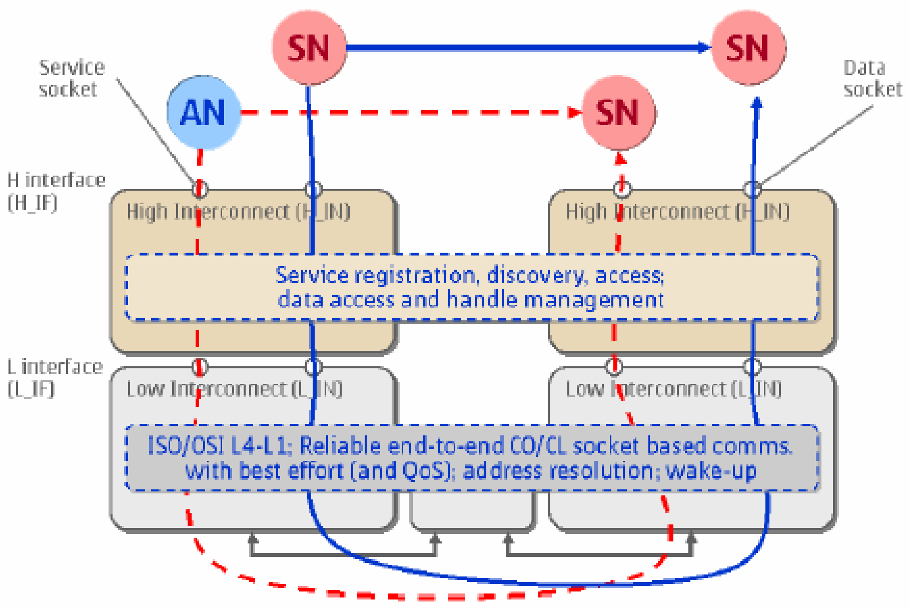

SN
SN
Service socket
AN
SN
Data socket
H interface (H_IF)
High Interconnect (H_IN)
High Interconnect (H_IN)
Service registration, discovery, access; data access and handle management
L interface (L_IF)
Low Interconnect (L_IN)
Low Interconnect (L_IN)
ISO/OSI L4-L1; Reliable end-to-end CO/CL socket based comms. with best effort (and QoS); address resolution; wake-up

*Figure 1-1 NoTA Interconnect* [1]

## 2 Goal

In this project, a small group of devices were to be integrated into a logical entity functioning via a communication channel between the devices. The communications channel was to be implemented using the NoTA stack. Another objective was also to research the feasibility of an FPGA (Field Programmable Gate Array) as one type of a NoTA implementation core component. A NoTA demonstration platform was to be ready by the end of the project (30th of September, 2008).

## 3 Description

At the beginning of the project, the NoTA demonstration platform was sketched as shown in Figure 3-1. It consists of a stereo microphone system connected to a printed circuit board with an FPGA circuit on it. The FPGA calculates the direction of the stronger sound amplitude measured by the microphones in real time, and sends that information to a computer acting as a NoTA service node. Also, a video camera with tilt control is connected to the same PC. By default, the computer steers the camera in the direction where the higher sound amplitude was observed.

The camera tilt angle can also be controlled from a laptop PC having a NoTA application node software implemented on it. The application node is connected to the service node through a NoTA interconnect that uses TCP/IP for data transport. In addition to video control command transmission, also video stream can be transferred all the way from the video camera to the NoTA service node and through the NoTA interconnect to the NoTA application node.

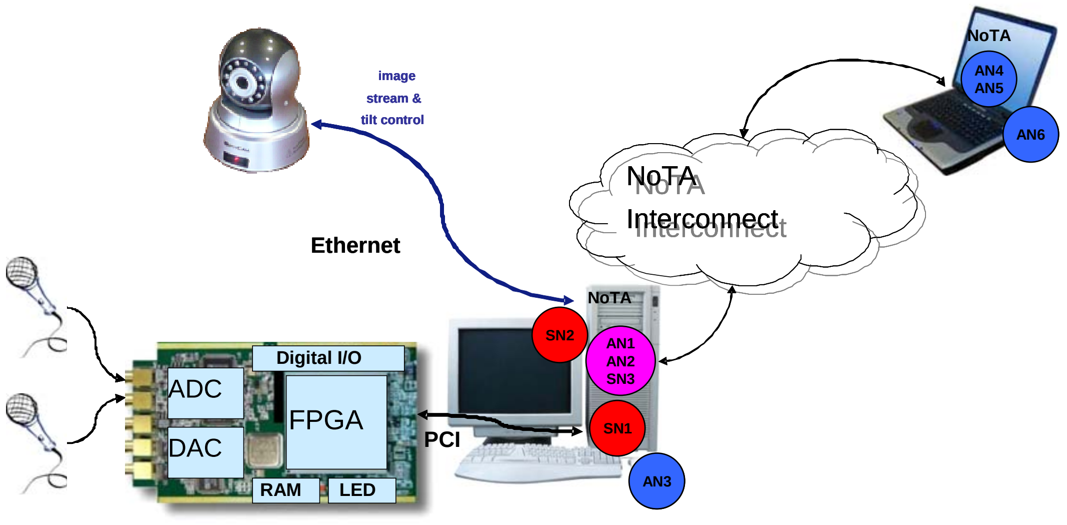


*Figure 3-1 EMSYS demonstrator platform setup*

This chapter is divided into several sub-chapters. The first sub-chapter explains the general structure of the demo application. The second sub-chapter describes the implementation of the demonstrator (FPGA and video camera control). The third sub-chapter describes running the demo in more detail. The last sub-chapter studies NoTA development tools.

## 3.1 Overview of the demonstration software

The demonstration platform consists of a desktop PC offering microphone and camera services and a laptop client using them. Figure 3-2 shows NoTA subsystems and message flows between the subsystems.

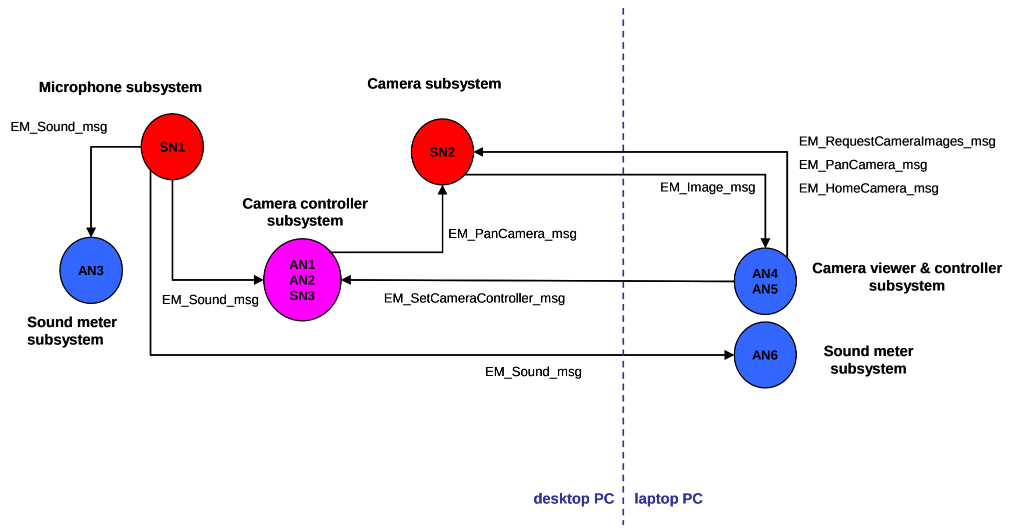


*Figure 3-2 Message flows between software components*

The desktop PC has three NoTA subsystems. The first subsystem is a microphone service node. The second subsystem is a camera service node. The third subsystem steers the camera in the direction where the stronger sound amplitude was perceived. The laptop PC has a camera viewer and controller subsystem. The sound meter subsystem can be simultaneously run in the desktop and laptop PCs. All NoTA service nodes implemented in the demonstration can serve multiple NoTA clients (ANs) at the same time.

### 3.1.1 Microphone subsystem

The microphone subsystem contains a microphone service node (SN1) that offers a simple service of sending a sound amplitude difference to all connected clients automatically. The amplitude difference is a signed 32-bit integer. Negative values mean that sound is coming from a microphone on the left and positive values that the sound is coming from a microphone on the right. The message is sent every 100 ms.

The microphone service node operates also in a role of a resource manager for the NoTA interconnect.

### 3.1.2 Camera subsystem

The camera subsystem contains a camera service node (SN2) that offers control and image stream services. For the demonstration, camera panning operations (home, left, right) were implemented. An application node can also request a stream of images from the camera service node. Images are sent at a constant speed (by default 10 frames per second). The application node can also stop the image stream if images are not desired any more.

### 3.1.3 Camera controller subsystem

The camera controller subsystem is responsible for steering the camera into the direction of a microphone where a louder sound is detected. The subsystem contains the functionality of three NoTA nodes, one service node (SN3) and two application nodes. The microphone application node (AN1) receives sound difference messages from the microphone SN (SN1). The camera control application node (AN2) sends camera panning messages to the camera SN (SN2). The service application node allows turning the automatic camera control ON or OFF. When the automatic control is disabled, the camera can still be controlled manually from a remote AN connected directly to the camera SN.

### 3.1.4 Camera viewer and controller subsystem

*Image client* is a graphical GTK2.0+ toolkit application to provide the visual feedback in the demo. It shows the user the Web camera view at a rate of about 10 FPS. The user interface includes buttons to manually turn the Web camera and to activate or deactivate the automatic sound volume based camera control.

Image client connects to two service nodes over the NoTA network, the camera controller subsystem (SN3) and the camera subsystem (SN2). Having two simultaneous client connections is handled by using GTK threading. The main thread runs the UI and takes care of sending button-initiated command messages to the two servers. Another thread receives image messages from the camera subsystem and updates the camera image on the application window. Only two NoTA *Hsocket*s are needed as the same camera subsystem socket can be used to send data while waiting or reading data back from the socket in another thread. The other socket is used just to send control mode commands to the camera controller subsystem.

### 3.1.5 Sound meter subsystem

The sound meter subsystem is a simple GUI that shows the current sound amplitude difference graphically. It consists of a microphone application node (AN3, AN6) that receives sound difference messages. The subsystem was written in C++ using the *gtkmm* library. This demonstrates that a NoTA stack can easily be also used in C++ applications. It also runs simultaneously between multiple clients.

### 3.1.6 EMSYS NoTA demo service messages

The actual binary form of the EMSYS demo service messages is shown below. They are similar to the *Stubgen* tool defined data messages even though we didn't use Stubgen to generate the messages.

The message that one party sends is received by the other. We define the interface here from the sending point of view.

See NoTA documentation for full details on the service messaging syntax. uns8...64 define unsigned integers of the specified bit size. int8...64 define signed integers of the specified size.

The general format of the NoTA service message is shown first, followed by the EMSYS demo specific messages.

```
uns8            Signal id length code 0xA1 (one byte) or 0xA2 (two bytes).
uns8/uns16      Signal id, one or two bytes, can be chosen freely to label messages
<argument token type>         Defines the type of the following data.
<argument token value(s)>     The actual data bytes.


                LAPTOP CLIENT (AN4-AN5) SENDS

  REQUEST WEB CAMERA IMAGES FROM THE IMAGE SERVER (SN2)

uns8            0xA1                       Short id.
uns8            0x12                       Message number.
true/false      0x01/0x02                  Start (true) / stop (false) receiving).

The server responds by starting or stopping to send the "Send camera image"
messages (at a rate about 10 FPS).

  HOME CAMERA (IMAGE SERVER)

uns8            0xA1                       Short id.
uns8            0x40                       Message number.
null            0x00                       No parameters

The server does not respond.

  TURN CAMERA (IMAGE SERVER)

uns8            0xA1                       Short id.
uns8            0x41                       Message number.
uns8            0x21                       Arg. type int8.
int8            -128…127                   The number of steps to turn the camera.

The server does not respond.

  SET AUTOMATIC/MANUAL CAMERA CONTROL FOR THE CAMERA CONTROLLER

uns8            0xA1                       Short id.
uns8            0x31                       Message number.
true/false      0x01/0x02                  Automatic (true)/manual (false) control.

The server does not respond.

                CAMERA CONTROLLER (AN1-AN2,SN3) SENDS

  TURN CAMERA (IMAGE SERVER)

uns8            0xA1                       Short id.
uns8            0x41                       Message number.
uns8            0x21                       Arg. type int8.
int8            -128…127                   The number of steps to turn the camera.

The server does not respond.

                SOUND SERVER (SN1) SENDS

  SEND SOUND DIRECTION

uns8            0xA1                       Short id.
uns8            0x51                       Message number.
uns8            0x24                       Arg. type int32.
int32           -2147483648…2147483647     The amount of sound difference between
                                           the microphones.

The client does not respond.

                CAMERA SERVER (SN2) SENDS

  SEND CAMERA IMAGE

uns8            0xA1                       Short id.
uns8            0x61                       Message number.
uns8            0x42                       Arg. type binary data 2.
uns16           0…65535                    The image size in bytes.
uns8[]          Image data                 JPEG image from camera (~ 30 KB max).

The client does not respond.
```

## 3.2 Demonstrator Implementation

### 3.2.1 FPGA implementation

In order to inform a NoTA service node about the direction of the sound, various tasks need to be performed that are mainly related to digital signal processing in the FPGA. Analog sound signal from the microphones is first pre-amplified with an external preamp module before feeding the signal into the actual FPGA processing board. Inside the board, the sound is first sampled by an A/D converter and the 14-bit samples are then fed into the FPGA. A Xilinx XC2V2000-FG676 FPGA is used as the main data processing unit on a BenAdda module, included in an Extreme Development Kit manufactured by Nallatech. The FPGA does low-pass filtering in order to filter out high frequencies, which are considered noise. The filters were first modeled in Matlab with a pass-band frequency of 20 kHz (see Figure 3-3).

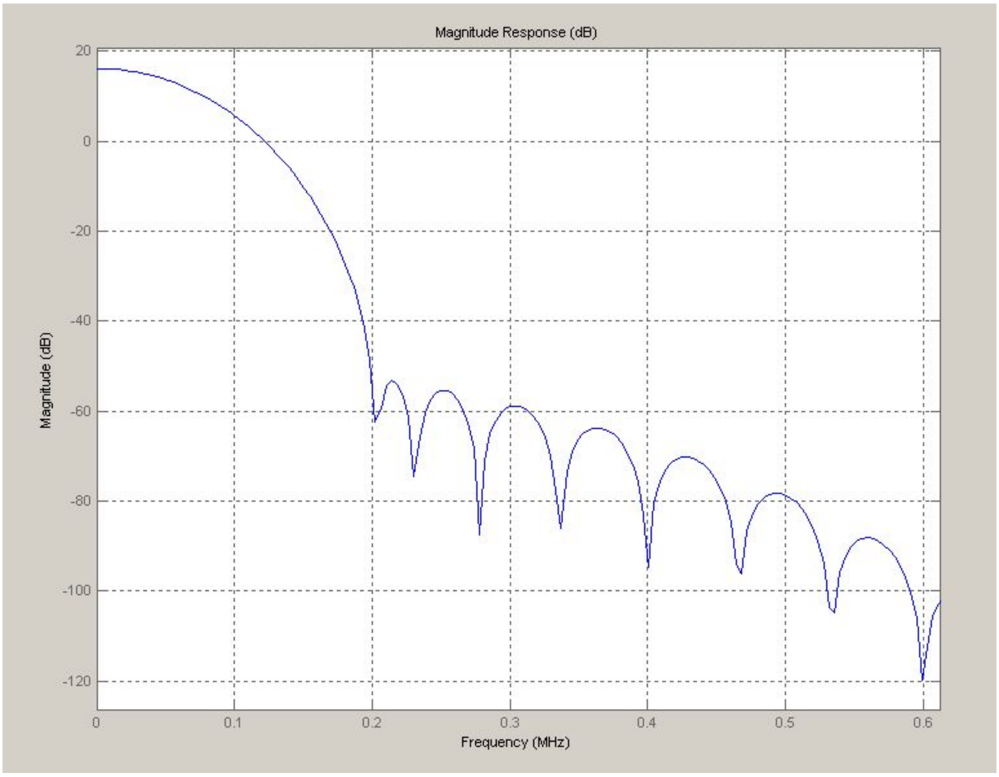


*Figure 3-3 CIC-FIR filter chain spectrum*

The filters were then described in VHDL, consisting of primary building blocks such as adders, multipliers and shifters. Two types of filters were used (see Figure 3-4): At first, two decimating CIC filters are applied in series. CIC filters do not require the use of multipliers and are simpler than a corresponding FIR implementation, so with the high FPGA sampling frequencies this is a good option. After decimation, the data is forwarded to a FIR filter with a steep cut-off. The filter does use the multipliers but now that the sampling frequency is already brought down from 65MHz to 1 MHz, a large amount of multiplier logic is saved as new results are not expected at every 65MHz clock cycle. The number of filter taps for the FIR filters is 13. The FIR filter coefficient width is set to 18 bits due to the fact that the FPGA has 18-bit multiplier cores performing the multiply operations of the FIR implementation. During the CIC filtering process, the 14-bit wide input data is transformed to 45-bit values due to a huge pass-band gain of CIC filtering. Those values are truncated into 18 bits by a ‘Gain’ block in order to meet FIR input data width specifications.

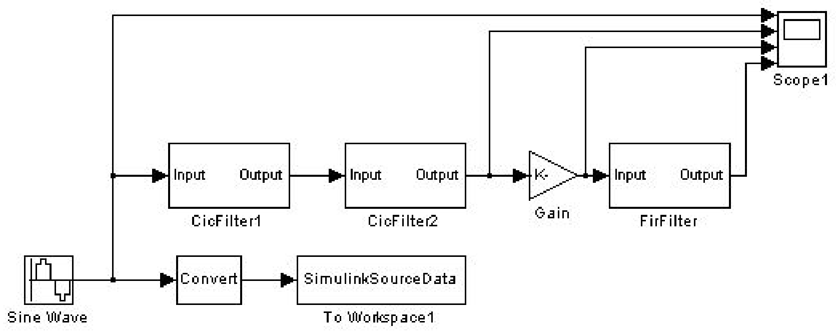


*Figure 3-4 Filter chain model*

After filtering, absolute values are calculated of the data values by subtracting every negative value from 0. Then, a 14-bit counter (running at 1MHz) is used to accumulate the absolute values in order to gather a longer period of data stream into a single register.

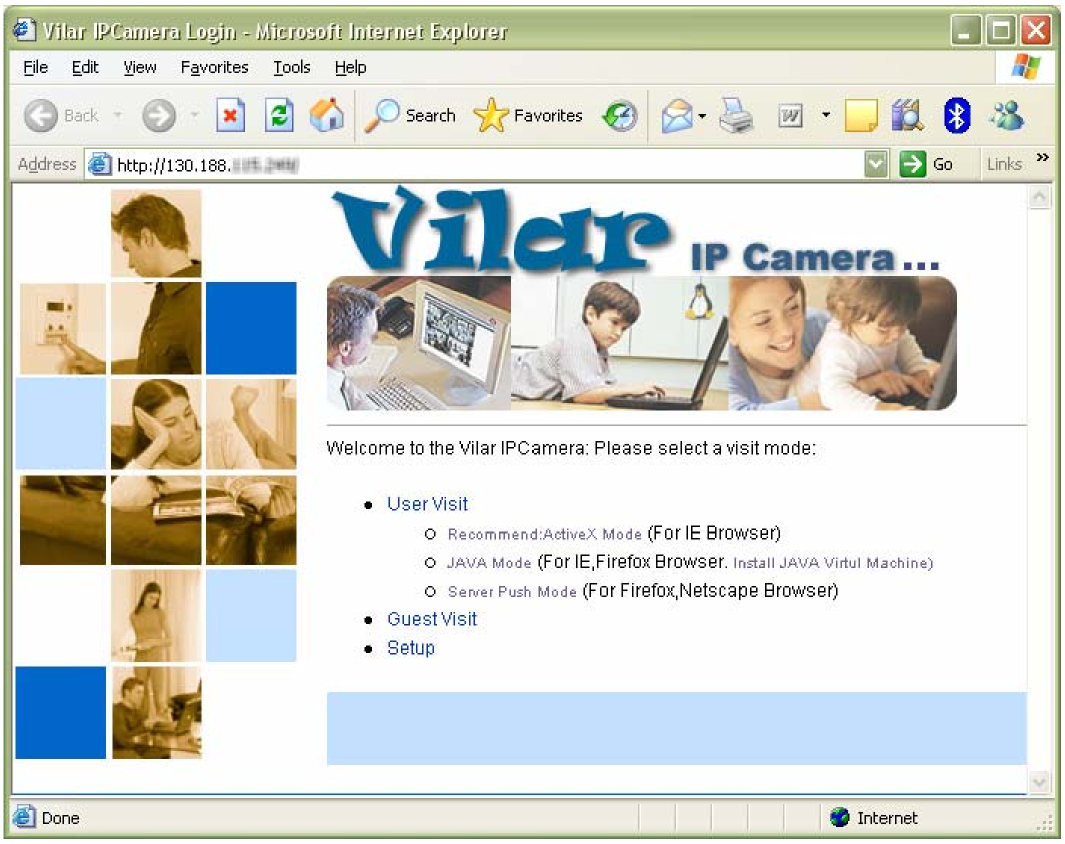
All the above is done in parallel for two sound channels, left and right. After accumulation, a new data value is registered once in every 16ms for both channels. If the accumulated value from channel 2 is now subtracted from the accumulated value of channel 1, we get a value corresponding to the accumulated difference in sound amplitude from the last 16ms period. A separate register control interface was implemented on the FPGA to transfer the difference value from the FPGA board to the desktop PC.

Initially we planned the FPGA board to transfer channel amplitude difference value to a laptop PC via a USB connection. During the project, it turned out that the USB interface of the FPGA board was not supported in a Linux PC. So, we needed to change the laptop PC to a desktop PC and settle for a connection where the FPGA board was plugged directly into a PCI slot of the computer. This change of plans did not affect the actual NoTA implementation, so it was acceptable.

The FPGA board is controlled by using a DimeSDL library. The library offers a C/C++ API for accessing the FPGA board functionality. DimeSDL library and the related files are only available in RPM-packages (for Red Hat Enterprise Linux), so there was a little work getting them to work in Ubuntu Linux. The required kernel-driver was available in source code form, so it was easy to get working with a newer kernel version. The implementation then was straightforward and consisted of three phases: initialization, register reading and cleanup.

### 3.2.2 Video camera control

The video camera used in the demo was an OptiCam IP Web camera. This camera can be controlled using a web interface. There is a different URL for each control operation (i.e. setting an image size/quality or turning a camera). The images are transferred using HTTP server push protocol. The image stream contains a series of small JPEG files.


Vilar IPCamera Login - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back
Search
Favorites
Address
http://130.188.[redacted]
Go
Links
Vilar IP Camera ...
Done
Internet

Welcome to the Vilar IPCamera: Please select a visit mode:

- User Visit
  - Recommend:ActiveX Mode (For IE Browser)
  - JAVA Mode (For IE,Firefox Browser. Install JAVA Virtul Machine)
  - Server Push Mode (For Firefox,Netscape Browser)
- Guest Visit
- Setup

*Figure 3-5 Vilar IP camera Web browser login screen*

The camera interface was implemented using a libcurl library. It is a free and easy-to-use client-side URL transfer library, supporting HTTP. The video camera module implementation has a separate thread for reading the image stream and another thread for control functions.

## 3.3 Running the demonstration system

### 3.3.1 Physical setup

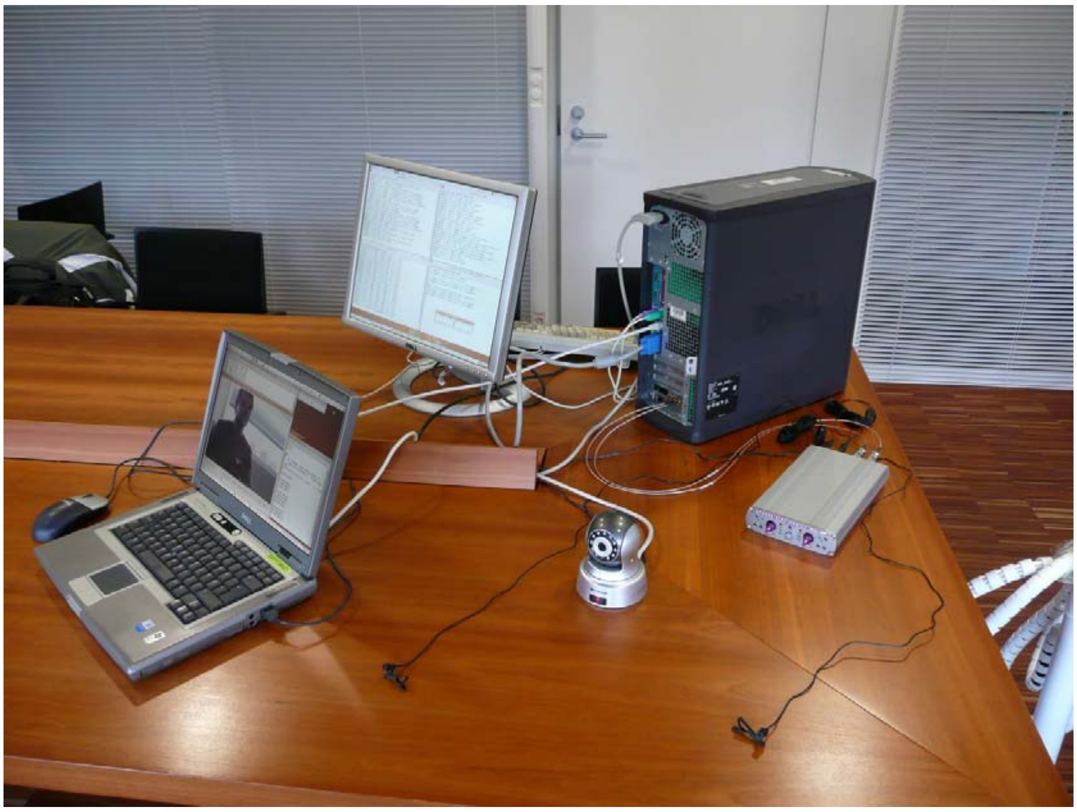


*Figure 3-6 Demo hardware setup*

### 3.3.2 About The Nota Stack

Get the NoTA stack from http://www.notaworld.org and compile and install it according to the instructions. At least versions 3.0-307...3.0-434 should work. Apply the *Hselect* patch from **.../emsys/trunk/patches** to the stack before compiling and installing it, if you want to have a faster image and sound refresh speed in the demo (recommended).

### 3.3.3 Building the Demo Applications

The demo application is meant for *Ubuntu 6.06 LTS Linux* or newer but should work on most flavors of the operating system.

Required packages and libraries that need to be installed before making or running the demo (older versions may work too):

- build-essential
- libgtk2.0-dev
- libcurl3-dev or newer
- Nota 3.0 stack version 307 or newer
- g++ (for sound_ui)
- gtkmm-2.4 (for sound_ui)

- Nallatech's DimeSDL-library (see trunk/doc/FPGA.txt for more information)

Check the common/config.mk compile and linking flags. The top level Makefile can invoke a compile in all sub-folders. The Makefiles in the sub-folders can be used to compile each subsystem separately.

Compile whole project:

```
$ make depend
$ make
```

To clear all compiled and linked files, run the target clean:

```
$ make clean
```

The make command should create all the demo applications into their own directories. By default, the applications use the single process (sp) version of the NoTA stack. This means that each application has its own copy of the stack embedded into its executable. An alternative way would be to share the stack among applications by using a **nota-ind** process (with a resource manager) in each computer.

### 3.3.4 Running the Demo

There are two ways to use the NoTA stack by an application: via a NoTA daemon and via an embedded stack (single process nodes). This demo uses the latter way, because it is more robust although not as efficient as the daemon method. There needs to be one resource manager enabled node in the NoTA network to handle node identities, to manage node discovery with the specified transport, and to control routing and addressing of NoTA messages between nodes. That node is chosen by specifying the environment variable **RM=1** when launching the node application.

The LD_LIBRARY_PATH environment variable tells the application where the NoTA stack library is located. Make sure that LD_LIBRARY_PATH is set correctly if libraries are installed in a non standard location.

The Vilar IPCamera Web camera needs to have its IP address correctly set for the demo network. See camera documentation and test the settings by accessing the camera with a Web browser.

### 3.3.5 Running the Demo When Using the Embedded NoTA Stack

**Set up the network**

NoTA messaging over TCP/IP uses multicast to locate nodes from each other. This discovery works only within a subnet. The following (admin) commands may be needed for each computer in the network:

```
$ sudo route add -net 224.0.0.0 netmask 240.0.0.0 dev eth0
```

The above enables the multicast routing.

NoTA implementation (3.0-434) contains a bug that can affect for TCP/IP transport if multiple network interfaces are available. Unnecessary interfaces can be disabled using the following command for each interface.

```
$ sudo ifconfig eth1 down
```

**Workstation**

```
$ RM=1 ./sound_server/sound_server_sn

$ ./camera_server/camera_server_sn

$ ./camera_control/camera_control_sn

$ ./sound_ui/sound_meter
```

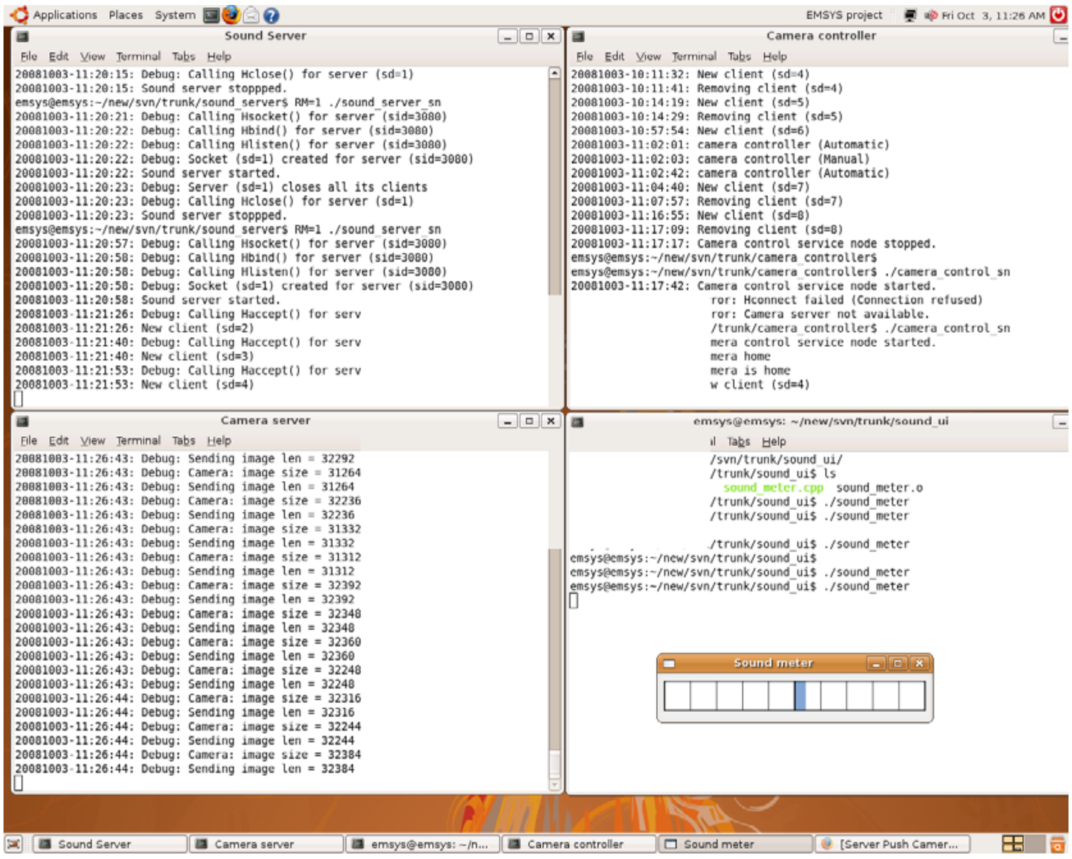
**Laptop**

```
$ ./laptop_client/laptop_client_ui_sp

$ ./sound_ui/sound_meter
```

Everything should work automatically. The laptop client launches a Gnome user interface for viewing the camera images and for manually controlling the camera.

Exit the demo by issuing CTRL-C to the console applications. Press the close button [x] in the client window.



```
20081003-11:20:15: Debug: Calling Hclose() for server (sd=1)
20081003-11:20:15: Sound server stoppped.
emsys@emsys:~/new/svn/trunk/sound_server$ RM=1 ./sound_server_sn
20081003-11:20:21: Debug: Calling Hsocket() for server (sid=3080)
20081003-11:20:22: Debug: Calling Hbind() for server (sid=3080)
20081003-11:20:22: Debug: Calling Hlisten() for server (sid=3080)
20081003-11:20:22: Debug: Socket (sd=1) created for server (sid=3080)
20081003-11:20:22: Sound server started.
20081003-11:20:23: Debug: Server (sd=1) closes all its clients
20081003-11:20:23: Debug: Calling Hclose() for server (sd=1)
20081003-11:20:23: Sound server stoppped.
emsys@emsys:~/new/svn/trunk/sound_server$ RM=1 ./sound_server_sn
20081003-11:20:57: Debug: Calling Hsocket() for server (sid=3080)
20081003-11:20:58: Debug: Calling Hbind() for server (sid=3080)
20081003-11:20:58: Debug: Calling Hlisten() for server (sid=3080)
20081003-11:20:58: Debug: Socket (sd=1) created for server (sid=3080)
20081003-11:20:58: Sound server started.
20081003-11:21:26: Debug: Calling Haccept() for serv
20081003-11:21:26: New client (sd=2)
20081003-11:21:40: Debug: Calling Haccept() for serv
20081003-11:21:40: New client (sd=3)
20081003-11:21:53: Debug: Calling Haccept() for serv
20081003-11:21:53: New client (sd=4)
```

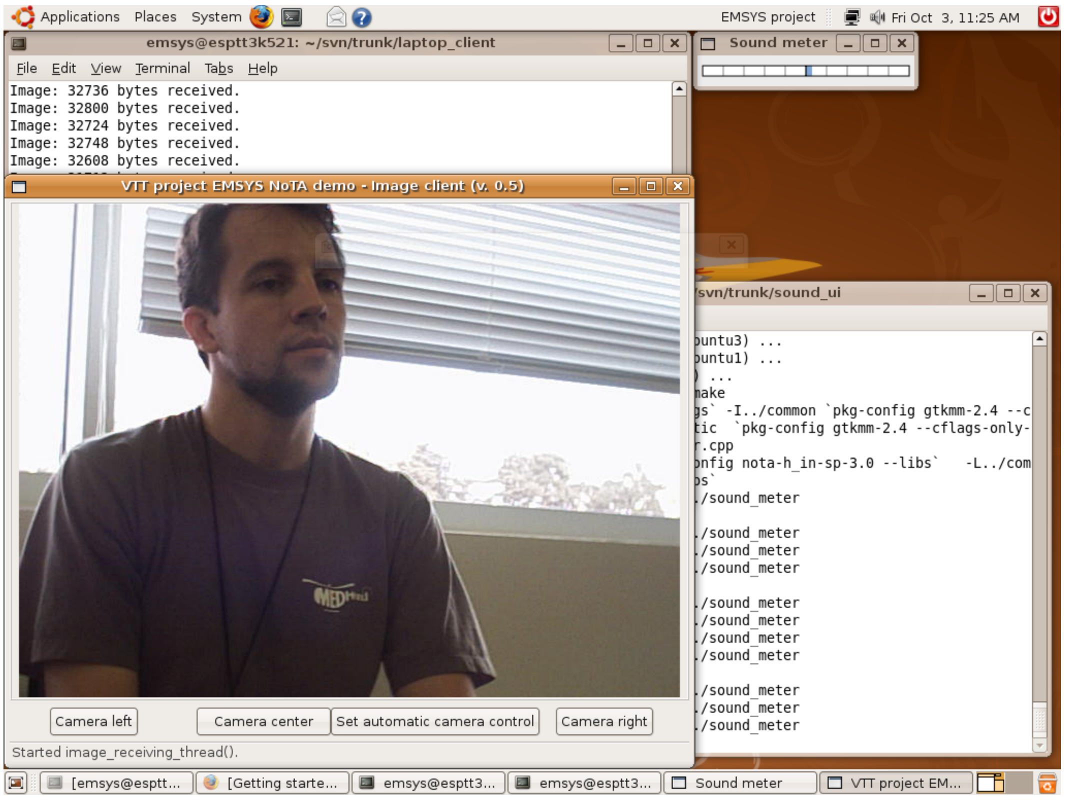
```
20081003-10:11:32: New client (sd=4)
20081003-10:11:41: Removing client (sd=4)
20081003-10:14:19: New client (sd=5)
20081003-10:14:29: Removing client (sd=5)
20081003-10:57:54: New client (sd=6)
20081003-11:02:01: camera controller (Automatic)
20081003-11:02:03: camera controller (Manual)
20081003-11:02:42: camera controller (Automatic)
20081003-11:04:40: New client (sd=7)
20081003-11:07:57: Removing client (sd=7)
20081003-11:16:55: New client (sd=8)
20081003-11:17:09: Removing client (sd=8)
20081003-11:17:17: Camera control service node stopped.
emsys@emsys:~/new/svn/trunk/camera_controller$
emsys@emsys:~/new/svn/trunk/camera_controller$ ./camera_control_sn
20081003-11:17:42: Camera control service node started.
                    ror: Hconnect failed (Connection refused)
                    ror: Camera server not available.
                    /trunk/camera_controller$ ./camera_control_sn
                    mera control service node started.
                    mera home
                    mera is home
                    w client (sd=4)
```

```
20081003-11:26:43: Debug: Sending image len = 32292
20081003-11:26:43: Debug: Camera: image size = 31264
20081003-11:26:43: Debug: Sending image len = 31264
20081003-11:26:43: Debug: Camera: image size = 32236
20081003-11:26:43: Debug: Sending image len = 32236
20081003-11:26:43: Debug: Camera: image size = 31332
20081003-11:26:43: Debug: Sending image len = 31332
20081003-11:26:43: Debug: Camera: image size = 31312
20081003-11:26:43: Debug: Sending image len = 31312
20081003-11:26:43: Debug: Camera: image size = 32392
20081003-11:26:43: Debug: Sending image len = 32392
20081003-11:26:43: Debug: Camera: image size = 32348
20081003-11:26:43: Debug: Sending image len = 32348
20081003-11:26:43: Debug: Camera: image size = 32360
20081003-11:26:43: Debug: Sending image len = 32360
20081003-11:26:43: Debug: Camera: image size = 32248
20081003-11:26:43: Debug: Sending image len = 32248
20081003-11:26:44: Debug: Camera: image size = 32316
20081003-11:26:44: Debug: Sending image len = 32316
20081003-11:26:44: Debug: Camera: image size = 32244
20081003-11:26:44: Debug: Sending image len = 32244
20081003-11:26:44: Debug: Camera: image size = 32384
20081003-11:26:44: Debug: Sending image len = 32384
```

```
                    /svn/trunk/sound_ui/
                    /trunk/sound_ui$ ls
                      sound_meter.cpp  sound_meter.o
                    /trunk/sound_ui$ ./sound_meter
                    /trunk/sound_ui$ ./sound_meter

                    /trunk/sound_ui$ ./sound_meter
emsys@emsys:~/new/svn/trunk/sound_ui$
emsys@emsys:~/new/svn/trunk/sound_ui$ ./sound_meter
emsys@emsys:~/new/svn/trunk/sound_ui$ ./sound_meter
```

*Figure 3-7 Workstation console views for active SNs and ANs*


Applications Places System
EMSYS project Fri Oct 3, 11:25 AM
emsys@esptt3k521: ~/svn/trunk/laptop_client
File Edit View Terminal Tabs Help
Sound meter
VTT project EMSYS NoTA demo - Image client (v. 0.5)
/svn/trunk/sound_ui
Camera left
Camera center
Set automatic camera control
Camera right
Started image_receiving_thread().
[emsys@esptt...
[Getting starte...
emsys@esptt3...
emsys@esptt3...
Sound meter
VTT project EM...

```
Image: 32736 bytes received.
Image: 32800 bytes received.
Image: 32724 bytes received.
Image: 32748 bytes received.
Image: 32608 bytes received.
```

```
buntu3) ...
buntu1) ...
) ...
make
gs` -I../common `pkg-config gtkmm-2.4 --c
tic  `pkg-config gtkmm-2.4 --cflags-only-
r.cpp
onfig nota-h_in-sp-3.0 --libs`   -L../com
bs`
./sound_meter

./sound_meter
./sound_meter
./sound_meter

./sound_meter
./sound_meter
./sound_meter
./sound_meter

./sound_meter
./sound_meter
./sound_meter
```

*Figure 3-8 Laptop console views and the graphical UI*

## 3.4 NoTA – Development Tools Identification

Currently there is only one tool to help a developer to utilize the NoTA. This tool is a stub generator that generates source code for helper functions to use the NoTA communication platform. The stub generator is implemented as a perl-script. As an input, the stub-generator takes a description of the service's methods defined in a WSDL (Web Services Description Language) file. The used data types may be defined in a separate XSD (XML Schema Definition) -file.

This document discusses some ideas, as a kind of "developer's wish list", of how to make it easier for a software developer to create application and service nodes in the NoTA framework.

### 3.4.1 Stub generator files

The service's methods are defined in a WSDL file. The stub generator takes the WSDL file as an input and generates header and implementation files for the service implementation as well as for the client implementation. Generated stubs require a NoTA stub adapter library to work. The stub adapter contains platform specific parts, and it is included in the stub generator package.

#### 3.4.1.1 Service stub

The service's stub header defines the methods to be implemented in the service's actual implementation: a function to handle connection closing and errors as well as the defined service's methods (`direction="in"` in the WSDL). The stub implements the methods for the sending of the service messages (`direction="out"` in the WSDL).

The service implementation is responsible for creating and binding the service socket and starting to listen and accept client connections. When a client connection is accepted, the service creates a new context for the client using the stub adapter's methods. After the context is created, the service uses the stub's method to tell the platform the client's context and the callback function for the client's signals when it calls the service's methods and the callback functions for the error and disconnect handlers.

When the client's connection is closed, the service calls the remove connection method of the stub, and if no error takes place, then the service calls the stub adapter method to free the client's context.

#### 3.4.1.2 Client stub

The client's stub header defines methods to be implemented in the client's actual implementation: a function to handle connection closing and errors as well as the methods to receive the service's messages (`direction="out"` in the WSDL). The client stub implements methods for calling the methods of the service (`direction="in"` in the WSDL).

The client implementation is responsible for creating and connecting the client socket to the service. When the client socket is connected to the service and context pointer initialized with the help of the stub adapter's method, the client calls the stub's connection method to tell the platform the callback function for the service's signals when it is sending messages to the client and the callback functions for the error and disconnect handlers.

When the client closes its connection to the service, it calls the stub's remove connection method, and thereafter it calls the service adapter's method to free its context.

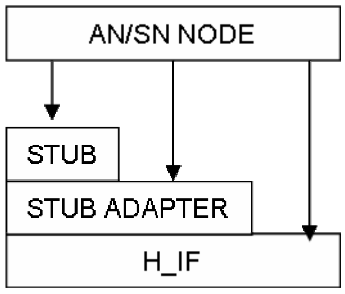
### 3.4.2 Inadequate stubs

The generated stub files use the stub adapter's methods to fulfill the client's or service's requests. However, the service and client implementation must also use the stub adapter's methods and H_IF methods directly.

For example, when the service is removing a client connection, it calls the stub's remove connection method and the stub adapter's method is called to remove the client connection from the platform. After the connection is removed, the service must call the stub adapter's method to free the client's context. The same overlapping exists when a new client connection is accepted: first the service creates the client context with the help of the stub adapter's method and after the context is created the service calls the stub's new connection method where the stub calls the stub adapter's method to add the new connection to the platform. Also the client implementation must use directly the methods in the stub adapter and H_IF.

Stubs generated to be used by the node's implementation are intended to make it easier for developers to implement NoTA based applications and services.

Currently, the stub generator facilitates only a part of the functionalities provided by the H_IF. A developer must learn also the stub adapter's usage and the socket handling in the H_IF (see below).



*Picture 3-9 Using NoTA DIP requires calling methods in different abstraction levels*

### 3.4.3 Developer's wish list

When designing a service, the details of how it communicates with other actors should not be the main question. The main question relates to the content of the service: what does it offer to help other services or users to accomplish their tasks. So, the first thing to do when implementing a service could be to define the descriptive methods and the data types taken as input and output parameters as well as other required attributes, such as the service name and type. Defining methods and data types should not require any additional knowledge of any specific markup language. After the service's methods are defined there should be a tool that generates code to take care of the communication routines and skeleton for the service methods, so that the developer needs only to implement the content of the service's methods.

#### 3.4.3.1 Node definition tool

Currently, the service nodes and their methods are defined with WSDL. Also a separate XSD file may be used to define data types used in the methods. Although both WSDL and XSD are quite well known formats, it is quite tedious to write a service definition from a scratch, because one must first find out the required and optional attributes in order to get the definitions schematically correct.

A node definition tool with a sophisticated graphical user interface could be of great help when defining a service node. With such a tool, the needed data types would be defined first. A name would be given for the data type, and its base type could be selected from a list of primitive types and from data types that the user has already defined. When defining a structural data type, its members' types could be selected the same way. A description for the defined data type could be given in the associated documentation.

Service Interface Definitions i.e. service's methods and their signal IDs would be defined next. A descriptive name and the direction would be defined. The direction would tell if the method is callable in the service or if it is a callback method that the client must implement in order to receive messages from the service. The method's input and output parameters' names and types would be defined here. The types could be selected from the previously defined data types list, including the primitive base data types. Also, other required attributes could be set, e.g. Service ID (SID), whether the service uses blocking calls or it uses asynchronous IO multiplexing, whether it uses Interconnect Daemon or it runs in a single process mode. A description for the method and its parameters might be given in the documentation.

If the defined node needs other services, their WSDL definitions could be imported. When the node's code is generated, the helper functions to call the selected services' methods would be created as well as the callback methods to receive messages from the other services.

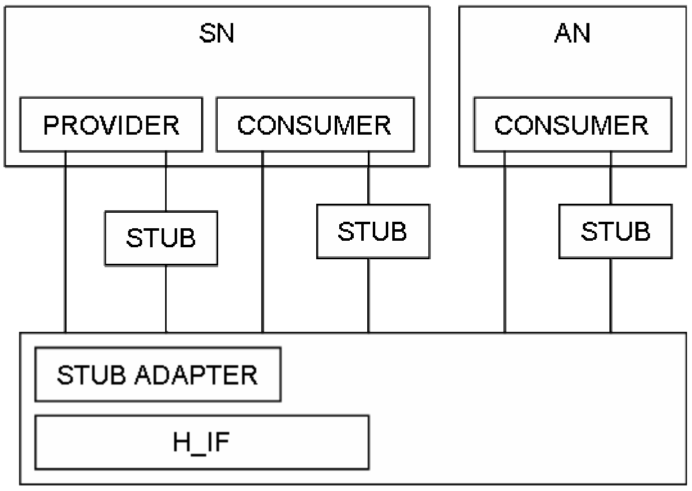
When the node definition is ready, it could be saved as a WSDL file and the defined data types as a XSD file. The saved node definition could also be loaded into the definition tool for changing the definition or for using a previously saved definition as a base for a new node definition.

#### 3.4.3.2 From stub generator to node generator

The node definition tool would have the functionality to start the code generation process. Also the status of the generation process and possible warnings and errors would be shown in the definition tool, so that the developer can make the corrections right away.

As mentioned earlier (see chapter *3.4.3 Developer's wish list*), the generator should do much more than the current stub generator does. First of all, the developer needs not to know how the message communication is implemented when calling the service's method or when some client calls the service's method. There might be some attributes in the node definition to make some restriction or preference among available communication channels, and these attributes would be taken into consideration when the helper code is generated.

From the service node's point of view, the generated code would include the functionalities currently implemented in the main method of a node. These functionalities include the socket creation, service registration and listening client connections. The component to take care of these routines would be called a **provider** (see picture below). In other words, the provider would implement the functionalities that currently exist in the main method of the service. The provider would also parse the input parameters from a client message call and would call the skeleton of the actual service methods with these parsed parameters and with the output parameters that are instantiated in the actual service method. The service method would return its output to the provider with the given output parameters, and the provider would take care of sending the result back to the client. This arrangement would make it possible to separate all the communication routines from the actual service implementation, and the developer would only need to write the service methods' implementation in the ready made skeletons of the methods with input and output parameters.



*Picture 3-10 Provider and Consumer codes generated to hide the complexity of the NoTA.*

Also for the application node's implementation, the generated codes would include the socket creation and service connection functionalities as well as listening to messages sent from the service. The component to take care of these functionalities would be called a **consumer**. The consumer would wrap the communication routines, so that in the client's implementation the only thing needed to complete would be the message receiving methods' skeletons. To use a method of the selected service, the client would only need to call the service's method offered by the consumer with the input and output parameters, and the service's method implementation in the consumer would take care of the context pointer handling and making the actual call to the service.

The service node might also use other services, so the consumers' code to use the other selected services could also be created with the node generator tool. If the application node used methods of more than one service, then all the selected services would be callable through the created consumer.

The provider/consumer approach would also make it possible to change the communication platform beneath them without any change to the service/client implementation above them. That is, only the generated part of the node's implementation would change if the communication platform changed.

## 4 Limitations

In this report, it has been shown that even though the microphone sensitivity is relatively high, the sound signal amplitude levels vary significantly depending on the direction of voice, and also on the position of different surfaces in the room. Sound waves propagate both directly and via reflections, thus making it difficult to build a real-time system that is able to accurately measure the direction of the original sound source by using only two microphones. Due to this fact, the microphones need to be set close to the persons who are talking in order for the system to recognize which microphone is currently sensing a stronger sound level. This is clearly a limitation in the demo but as the main target of this project was to improve our knowledge on embedded processing and NoTA, we did not want to spend more effort in the sound system design.

The published NoTA stack was found to be still suffering from some limitations and errors. We tried to use SOCK_SEQPACKET type of NoTA sockets that should preserve message boundaries, but they weren't working. The server socket threw the following error:

```
l_if.c:682: Lreceive: Assertion `buf' failed.
```

So, we had to use the normal SOCK_STREAM socket type instead.

Another limitation that was discovered occurs when using **Hselect()** function to switch between connected sockets. The shortest time for the function to return is one second (when specifying a time-out value 0). This was too slow when trying to send and receive camera images ten times per second. As a work-around, the NoTA stack needs to be modified, compiled and reinstalled (see section *3.3.2 About The Nota Stack* for the details). Once that was done, there were no serious limitations on the data transmission speed with multiple connections.

Due to resource and time limitations of the project, only the TCP/IP transport was used for the NoTA protocol. Testing Bluetooth or USB connections could have revealed more about the performance and multitasking capabilities of the NoTA protocol.

The *Stubgen* tool was found to be difficult and complex to use, and was not adhering to clean library interface design principles.

## 5 Results

A demo application consisting of several independent software modules was created. The demonstration shows that we can access microphone sound volume difference calculated by an FPGA card over a NoTA network, and use the value to control a Web camera over normal HTTP protocol. We also managed to transmit JPEG images, read by the Web camera accessing module, over the NoTA network and show them in another module to the user. There was also a possibility to manually turn the camera from the client UI by the user.

All the created software modules in the demo communicated with each other by using the NoTA protocol on top of a TCP/IP transport. The communication between modules (or nodes) consisted of both control and data specific messages. We found out that it is possible to transmit ten 30 KB sized images per second between modules inside the same intranet, as well as sound volume difference messages to two client modules at the same rate. It was also discovered possible to use a NoTA socket simultaneously in reading and writing mode by utilizing two threads. The modules also worked without interfering with the graphical user interfaces of some client modules.

Due to the small size of the Web camera images, we did not encounter problems with having too large messages. The largest binary data fields, defined by the NoTA service message protocol, are of 64 KB in size.

The use of NoTA sockets appeared quite similar to the traditional Unix/Linux BSD sockets. One difference is the return value, when reading from a closed socket. In BSD sockets, the return value 0 indicates that the peer has closed the connection, and the NoTA sockets use the return value -1 indicating the same condition.

## 6 Conclusions

The interdevice Network on Terminal Architecture (NoTA) protocol is a technology to connect software modules, or nodes, utilizing the NoTA stack to each other. The modules may reside either in the same device, e.g. computer, or in several different devices. The NoTA communications may utilize several alternative communication transports, Bluetooth, USB or TCP/IP, but only TCP/IP was used in this project.

The NoTA network uses its own addressing scheme, where accessible service modules/nodes are given pre-defined NoTA network wide 16-bit service identifier SID values and 16-bit port numbers. The NoTA stack handles the actual device and service addresses, and peer discovery needed by the transport automatically. Thus the transport addresses will be invisible to the applications and to the developer. The discovery process may be quite slow, however, with possibly up to 15 seconds with Bluetooth connections due to the slow device and service inquiry processes of that technology. The multicast service discovery used with the TCP/IP transport is much faster, in comparison, and was not observed exceeding a few seconds.

The Stubgen tool can be used to create a higher level skeleton code for NoTA client-server service messaging. However, its current shortcomings result in getting little benefit in the application development. A better, easier to use and more abstract tool utilizing a provider-consumer pattern was proposed and sketched.

Overall, despite the immatureness of the technology, the NoTA technology is a viable solution to loosely coupling software modules and hardware located in different physical devices.

## 7 Summary

A demo setup consisting of two computers, an FPGA board, two microphones and a Web camera was created on a Linux platform. The demo used NoTA protocol over TCP/IP transport to transmit sound or image data, and command messages between several service and application nodes. A graphical user interface was used to monitor and control the demo.

The NoTA protocol and its Stubgen tool were analyzed from the application developer's point of view. An improved toolset was proposed to further ease and abstract the software design and development process.

## References

[1] http://www.notaworld.org/documentation/