THE STYX IP-CORE FOR UBIQUITOUS NETWORK DEVICE INTEROPERABILITY

N.C. Audsley, R. Gao and A. Patil
University of York, York UK

ASBTRACT

Application level interoperability between ubiquitous networked communication devices (e.g. Mobile phones, PDA, CCD camera, etc.) poses many problems. In this paper we consider the issue of efficient application level access to resources on remote devices whilst achieving both network and distribution transparency. Provision of such transparency is difficult as low-resource devices are usually limited to one or two standard communication mediums (e.g. WiFi, Bluetooth, ZigBee). Thus, it is unlikely that an application node can communicate directly with all other nodes, with the requirement for some to act as intermediaries. Also, direct control of remote devices (potentially via some intermediary) in the same manner as local devices is not usually provided by conventional OSs.

In this paper we consider the Styx protocol (from the Inferno OS) as a solution to these problems. Styx is defined to provide a file based interface to devices, within a namespace that provides distribution transparency (coping with intermediary devices). However, Styx currently is only available as software, requiring a OS (and CPU). We define and implement a (hardware) Styx IP-core Module [1] to provide both network and distribution transparency for applications that control physically remote devices. For low-resource devices, such an approach removes the need for a CPU (to execute a software OS and Styx implementation). The implementation of the hardware Styx IP-core (and subsequent demonstration) presented within the paper show the efficacy of this hardware Styx approach.

INTRODUCTION

Application level interoperability between ubiquitous networked communication devices (e.g. Mobile phones, PDA, CCD camera, etc.) poses many problems. One key issue is that of uniform accessibility and control – ideally, an application should be able to access any (available) remote device in the same manner as it addresses local devices, e.g. via file commands (i.e. open, read, write). This is problematic for the low-resource devices increasingly seen in ubiquitous systems, e.g. a smart home. Usually, such devices support only one (or perhaps two) wireless communication standards – unless all devices within some system support a common standard, control of all devices by an application is difficult. One solution is to enable some devices to act as an intermediary, where A communicates via communication standard X to intermediary device B, which in turn communicates using standard Y to device C. Whilst a communications path has been established, uniform accessibility of devices on A, B and C by an application running on A, remains a significant problem.

In this paper we consider the issue of efficient application level access to resources on remote devices whilst achieving both network and distribution transparency. To achieve transparency, we adopt the Styx (2,3) network protocol. Styx is an application layer protocol that can run over any reliable communications link. It allows a file based interface to be provided for any device, coping with the problem of intermediary connections via a global namespace across all nodes and devices.

However, efficiency is a problem for existing Styx implementations, as they are software, requiring a supporting OS and hence CPU. In this paper, we propose and implement a hardware IP-core to solve the efficiency problem. The hardware IP-core Styx component can operate without a supporting OS or CPU, so enabling low-resource devices (that potentially do not have a CPU, or at least no spare CPU capacity) to be remotely accessible by applications via the same file interface.

This paper is structured as follows: in the next section we describe the Styx software component and its operation in detail. Section 3 describes the design of the hardware Styx IP-core. Section 4 discusses the implementation while Section 5 provides a detailed performance analysis between the software and hardware Styx components. Conclusions are offered in section 6.

BACKGROUND

Styx is the network protocol developed for the Inferno OS (2,4), designed by Bell Labs and now a product of Vita Nuova (5). It is an application layer protocol over any network protocol like TCP/IP (1), ATM, PPP, etc. The only requirements that Styx places upon the underlying network is of in-order and reliable send/delivery to/from the Styx layer. Hence Styx

[1] This work is part of the DEMOS project undertaken by the AMADEUS Research Centre, University of York, York, UK (http://www.cs.york.ac.uk/amadeus/)

provides network transparency to applications, providing a high level of abstraction of the network devices/resources. Additionally, Styx represents each device/resource on the network as a single or multiple file(s), so providing distribution transparency over all resources to applications.

Files provide a fixed and definite way of handling data. i.e. there are only open, read, write, and close operations that can be performed on files. Styx makes use of this fact and allows remote devices to open, read, write and close network devices/resources represented by it same as files. For example: a device, A can access a device/resource, R on the network in the form of file, F. The device can then simply use the other device/resource as if it were a local file. Any open/read/write/close operations performed by the device on file, F directly affects the actual network device/resource, R.

Remote access via filesystem abstractions has been utilised in many OSs. However, this is conventionally restricted to true data files rather than devices. In typical Unix style implementations (e.g. SunOS, Linux, BSD (10,13)) and Windows (i.e. Samba (11)), devices are not exported and hence are not available to remote applications via the virtual file system. The usual work around is the construction of local applications (or kernel level) servers to handle remote accesses to devices.

Styx and Inferno Namespace

The Styx software component is tightly coupled with the Inferno namespace. Each device is represented in the namespace as a single or multiple files. For example, consider a digital camera connected to a system running Inferno OS. This camera is represented by three files:

1. “camerastatus” – read-only, reporting the status of the camera.
2. “cameractl” – to send commands to the camera.
3. “camera” – acts as a read/write buffer depending on the command sent via file “cameractl”.

To take a picture from the camera, a user would open the file “cameractl”, write “click” and close the file. All the complexity that lies in the actual communication and execution of the command on the camera is handled by the Styx component. To read an image stored in the digital camera, a user would send a read command to the camera using “cameractl” file and as a result of which the image is received by reading data from the file “camera”.

Each node in the system, and each application on a node, can have a different view of the overall namespace. Thus an application uses its own copy of the namespace and can move the device-file location in its namespace anywhere in the hierarchy without affecting other applications. This gives greater flexibility to the applications without imposing unnecessary restrictions in the way they want to use the files, which are essentially network devices/resources.

A key feature of Styx and the Inferno namespace is that of chaining device accesses across multiple nodes. Thus, an application can access a device via an intermediary node (acting as communication bridge) – removing the need for total direct connectivity of all nodes (and connected devices) in the system. Thus, if intermediary serves the third device’s attached namespace via its Styx server, the application can access the device files of third device (via the intermediary) using the Styx protocol – initially, it connects to the intermediary and then starts using the device files of the third device as its local files. The overhead is a two-level of indirection, which is inevitable without total connectivity.

Though the namespace has been encapsulated into the Inferno OS, it is possible to implement the namespace alone without the OS. Essentially, the Styx protocol and Inferno namespace are separable from the overall Inferno OS, and can be used in isolation. Hence, indeed, both Styx and the associated namespace are lightweight and suitable for implementation in hardware.

The Styx Protocol

The Styx Protocol is designed to handle everything in terms of files; following is a brief description of protocol messages with functional descriptions:

- **Tauth/Rauth**: used to exchange authorisation information like the username and password. Note that the authorisation message may or may not be encrypted. When encrypted, it may use any encryption standard (e.g. MD5) that is agreed by both – the client and the server.
- **Tattach/Rattach**: once the connection has been authenticated, Tattach message is sent by the client to attach itself to the root node of the server’s namespace; Rattach message constitutes the reply sent by the server. For every “T” message (e.g. Tversion) sent by the client, the server must reply with a similar “R” message (e.g. Rversion) or an error message (e.g. Rerror). Note: in the message descriptions below, the “R” message is assumed to be sent by the server in reply and has not been included in the descriptions.
- **Twalk/Rwalk**: after a namespace has been attached, the client uses Twalk message to navigate within the attached namespace.
- **Topen/Ropen**: the client uses Topen message to open a network device/resource file present in the namespace attached by the server. The client has options to open the device/resource file in read-only or write-only or both read-write mode.
- **Tread/Rread**: client uses Tread message to read data from a previously opened network device/resource file.

- **Twrite/Rwrite**: client uses Twrite message to write data to a previously opened network device/resource file.
- **Tclunk/Rclunk**: client sends the Tclunk message in order to close a previously opened network device/resource file.
- **Tstat/Rstat**: client sends a Tstat message to get the statistics of the current namespace. For instance, when sent while the client's current-remote-working-path in the attached namespace is the root node, the reply to this message would contain the a list of network device/resource files present in that namespace along with their access rights and other statistics like file size, etc.
- **Rerror**: this message is generated by the server in case of any error encountered during its operation. Note that there is no Terror message as the server is not concerned about the errors occurring on the client side.

Connection. Figure 1. (a) shows the exchange of various Styx messages between a Styx client and server in order to establish an initial connection. The client sends a Tversion message containing its Styx protocol implementation version number. After verification, the Styx server responds by sending a Rversion. The Tversion message also gives the server the information about the maximum length of a Styx message that the client is capable of handling. If this message is longer than the maximum a particular Styx server is capable of handling, the server sends its maximum message length to the client in the Rversion message. In this way both the client and the server synchronise and agree upon a common maximum message length.

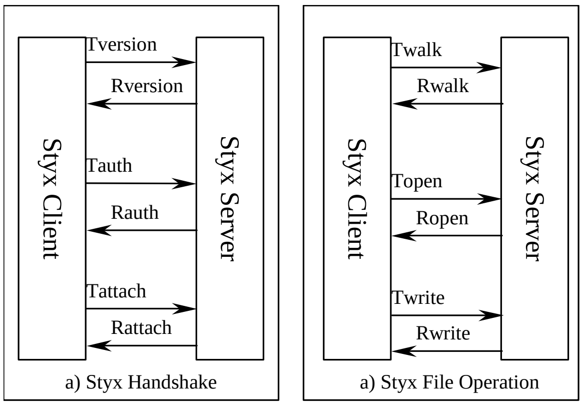

Figure 1 The Styx Protocol

The Client then sends a Tauth message containing the user name and password required to connect to the Styx server. This information may also be encrypted using any encryption standard agreed upon by both client and server during implementation. The server verifies the username and password and responds with a Rauth message on success.

Finally, the client issues a Tattach message requesting the server to attach it to the server's namespace. The server responds with a Rattach message that contains a handle to the root node of its namespace. This handle called QID is a 13-byte identification string containing the node/device/file identification number (FID), the version and the path in the namespace. Each device file or node in the server's namespace has a unique QID. Once the client receives a Rattach message from the server, it is then ready for communication (read/write device files) with the devices connected to the server.

File Read / Write. Figure 1. (b) describes the Styx messages exchanged between client and server depicting a file write operation by a Styx client. The client issues a Twalk message to navigate to the node that contains the required file that it wants to write to. The server changes the client's working node to the requested node if it exists and the client has sufficient permissions to access it. On success, the server replies by sending a Rwalk. The client then issues the Topen message containing the identification number (FID) of the file to be opened. This FID is unique and local to every client. On the server side, the FID is associated to a QID in its namespace. Thus, on the client side, it is possible that two or more FIDs point to the same QID on the server side. This is how every client has its own copy of the namespace by using their own FIDs and manipulating them as per their requirements. On receiving Topen message, the server associates the client's FID to the device files QID and replies with a Ropen message.

To write data (either command to the associated device, or mere data) to the opened file, the client now issues a Twrite containing the data to be written. It is possible for the client to issue more than one Twrite message if the length of data to be written is greater than the maximum Styx message length allowed. Each such message is tagged by a message identification to provide information to the server about the order in which it has to write the data to the file. The server may not literally write the data it receives. It will decode the data field for commands and carry out the required operation(s) on the actual device accordingly. If appropriate the server now changes the information present in the device's status file (e.g. camerastatus). The server sends a Rwrite message containing the information about the number of bytes written to the file. When there were multiple Twrite messages sent by the client, the server also replies in equal number of Rwrite message carrying the same message tags.

Finally, the client issues a Tclunk message to close the opened file. The server carries out the required operations (e.g. disabling a device, putting it to sleep, etc.) on the device and replies with a Rclunk message. For any error encountered by the server during its operation, a Rerror message is sent to the client describing the error.

Observations. A Styx aware node/device in the network can choose to be either a client and/or a server. By removing the complexity involved in network communication from the applications or devices, Styx provides a standard file interface, providing network and distribution transparency over devices to applications. Hence, from an operational perspective, Styx provides interoperability of various ubiquitous devices. However, given that Styx has to handle all the network complexity and also maintain interoperability between different kinds of networks as well as devices, the software implementation can become a significant overhead, particularly for a low-resource device that wished to provide remote access (e.g. the camera described above).

This provides a key motivation for implementing Styx in hardware, in the form of a IP-core. Such a core can operate in parallel and independent of the device's CPU there by providing better performance. Also, it can free a low-resource device from having a CPU, if it is not necessary for the other functionality of the device. Given the advantages that Styx can provide, it is certain that hardware Styx IP-core will help different unrelated devices to communicate easily with each other. The remainder of this paper describes the design of Styx IP-core, together with its evaluation.

DESIGN OF STYX IP-CORE

The design of the Styx IP-core is multi-faceted. A device can choose to be either a client and/or a server. For example: a digital camera, which can take pictures, store them or send them across the network does not need assistance from any other device in the network. Thus, the camera can choose to be Styx server only that responds to other client requests like taking picture, retrieving picture, etc. On the other hand a touch screen present to interact with user can choose to be a Styx client only. Its only purpose is to get users request and connect to other Styx servers to retrieve the required information (e.g. to acquire a picture from the digital camera Styx server) or perform the desired operation. The combination – both a client and server is chosen when a device acts as a bridge. In this case it acts as a server to the client (that cannot communicate directly with the concerned server) and as a client to the server (that the client wants to connect to). Both the Styx client and the server need to implement the namespace. In the following sub-sections we describe the design of Styx hardware namespace, Styx client IP-core and Styx server IP-core.

Styx Namespace

In a software Styx implementation, the namespace is completely embedded into the OS, with no limit on the file size. However, the hardware approach varies as the IP-core component can be used as a standalone or together with a general purpose CPU. We chose to put certain bounds on the hardware namespace as follows:

- **File Type**: the namespace would represent only files related to the devices shared/accessed by the Styx server/client respectively. Representation of data files of a particular file-system is not allowed.
- **File Size**: depending on the nature of the device being represented, the file size is chosen accordingly. However, the maximum file size is limited to 256 bytes. In case of certain devices (e.g. camera) which require a larger file size, we split the file into several parts each of which is not more than 256 bytes and the Styx server/client hardware has been designed to consider all these parts as a single file. This limit has been put in place to improve performance by reducing the time delay in accessing the namespace.
- **Number of Files**: there is no limit on the number of files in the namespace. Since the namespace would be implemented in RAM, it is limited by the amount of RAM available at the time.
- **Depth of namespace tree**: for simplicity the depth of the hardware namespace tree is restricted to only one. Thus, all the device files are direct descendents of the root node of the namespace.

The structure of hardware namespace is organised as a set of records where each record represents a file or part of a file (as explained above). Each file has a unique name and a unique identification string called QID. A typical file record would be as follows:

QID (13 bytes)	Filename (8 bytes)	Length (1 byte)	File Data (Length bytes)

It is important to note that these files are device files and are in direct relation to those specific devices. i.e. the data contained in them represents the state of the device at any time and any change made to the file would affect the device directly. This functionality is handled by a hardware component called Device Control Logic. It is tightly bound to the physical devices represented in the namespace.

Styx Client IP-core

The design of Styx client IP-core is simpler than the server IP-core. Figure 2 shows the Styx client IP-core component. It is directly wired to the system bus. Note that the arrows – one pointing towards the input buffer and one going away from the output buffer are both connected to a system bus through which all the other communication components exchange data. The Client IP-core consists of four units:

- Input buffer,
- Decoder,
- Encoder and
- Output buffer.

The Styx client IP-core receives requests through the system bus (e.g. to transmit data to a network device) from other components (e.g. the application process, network stack, etc.). These requests are stored in the input buffer of the client which are then decoded by the request Decoder unit. The Decoder decodes the requests into instructions to the packet Encoder unit. Subsequently, the encoder generates one or more Styx messages that carry out the request according to the Styx protocol. These Styx messages are then buffered within the client IP-core's output buffer. Finally, the network communication device is signalled of the generated message that it then transmits on the network. The Styx IP-core is completely transparent to the communication component allowing it to work on any underlying network transport medium (e.g. serial, ethernet, WiFi, etc.)

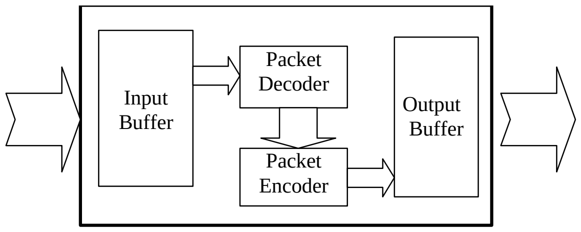

Figure 2 The Styx Client IP-core

Styx Server IP-core

The Styx server IP-core component has more complexity built into it in order to handle issues related to security and the namespace. In addition to the units present in the client, the Styx server IP-core component includes the following units:

- Authentication Unit,
- Namespace Control logic.

As described before, the namespace control logic further consists of the "device control logic" unit and the RAM based namespace itself. The server remains passive until a Styx message is received from the client. This message is transferred to the input buffer via the system bus either by the software running on the CPU (if present) or directly by the network communication unit (e.g. serial, ethernet, etc.). The server then decodes the received Styx messages into either requests for local data (e.g. status of a particular device) or control signals for devices that are represented by files. All such requests are handled by the namespace control logic unit. Figure 3 shows a Styx server IP-core component (note that the system bus is not shown).

During the initial phase of a connection from the client, on receiving a "Tattach" message the decoder unit authenticates the client using the authentication unit. The Authentication unit verifies the client and either allows or disallows the connection. It can make use of complex encryption standards like MD5, etc. the decision of which is left until the implementation stage. On successful authentication the encoder unit is signalled to send out a "Rattach" message consisting of a handle to the root node of the namespace tree. Furthermore, the decoder uses the authentication unit on every file access (e.g. open, read, write, close operations) made by the client. Unlike the previous case, the authentication unit this time checks for access rights on the files being accessed and the corresponding operations (e.g. read/write) being performed on them.

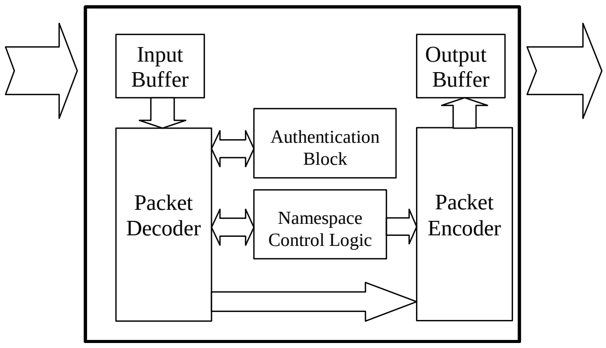

Figure 3 They Styx Server IP-core

The Namespace control logic is the main component of the Styx server IP-core. It is responsible for representing each sharable network resource in the form a single/multiple files. Furthermore, it also has to convert the file operations received from the client into the corresponding device operations. The Styx namespace component helps Styx provide a uniform "file-based" interface to all the shared network devices/resources. Finally, the Styx message encoder is used to build the Styx reply message to be sent to connected client.

Styx Client-Server IP-core

The combined Styx client and server IP-core consists of the client IP-core with the additional server modules such as authentication and namespace control logic. A high performance network intensive embedded device can make use of this dual Styx client/server IP-core to perform all the network communication in parallel, thereby saving clock cycles on the local CPU by several orders of magnitude and hence improving performance. However, only one message (either in the client or server mode) is handled by the hardware at any time.

Standalone Styx IP-core

Importantly, a Styx client, server, or a combined client-server IP-core can be used as a standalone component to establish a Styx connection between several other ubiquitous devices such as digital cameras, sensors, etc. without the need of a CPU. Since the Styx IP-core is directly connected to the system bus, interfacing it directly to the corresponding devices or communication units is fairly easy. In standalone mode the Styx component acts as glue between the application device (e.g. sensor) and its network interface. The standalone Styx IP-core provides a high-performance, lightweight, low-cost and interoperable solution to ubiquitous devices over the network. The next section describes the implementation of all the above units of the Styx server or client IP-core in more detail.

IMPLEMENTATION

All the units of the Styx client/server IP-core have been implemented as VHDL modules and verified on the Xilinx Spartan-2E FPGA (6). The input and output buffers have been implemented as register based byte FIFOs whose minimum length is 64 bytes, which can be varied depending on the requirement.

Styx Client IP-core

Although the client and server IP-core share similar named components (decoder and encoder), their functionalities are slightly different. In the Styx client IP-core, the decoder has the following functionality:

- the input buffer either receives instruction from other components or it receives the Styx reply ("R") messages from a Styx server via the system bus. On receiving an "R" message, the decoder decodes the contents into signals to the encoder alerting it of a successful or unsuccessful operation. In cases where the Styx server sends data through "Rread" message, the decoder takes appropriate action depending on the device functionality. For example: it might alert the CPU about the data being received, or it might display the data on a display device directly in some meaningful form or may even send it over the network to some other device. This can be easily configured during IP-core deployment.
- on receiving an instruction either from the CPU or other devices in the system, it decodes the instruction into appropriate signals to the encoder to generate the required Styx messages to be sent to the server. The instructions consist of only three parts – a one byte instruction code, a two byte data length field and finally a data field which if present would be of length given in previous field. If the length field is zero, then there is no data field. The instruction description along with their code is listed in Table 1.

The Styx client encoder unit on the other is completely controlled by the decoder unit. Its primary task is to generate Styx "T" messages and send them to the Styx server via a network communication device. Depending on the signal received from the decoder, the encoder generates the appropriate Styx messages. In the current implementation we have implemented a Styx client IP-core that communicates over serial line. The client IP-core can establish a Styx connection with a Styx server, navigate through the server's namespace, and perform open/read/write/close operations on the device files presented by the server.

Inst. Code (in Hex)	Description
0x01	Send a Tversion message to the server in data field.
0x02	Send a Tattach message to the server in data field.
0x03	Send a Twalk message given the server and path in data field.
0x04	Send a Topen message given the server and file name in data field.
0x05	Send a Tread message given the server in data field.
0x06	Send a Twrite message given the server in data field.
0x07	Send a Tclunk message given the server in data field.

Table 1 Instructions to the Client IP-core Decoder

Styx Server IP-core

Implementing the Styx namespace in hardware was the first step towards Styx server IP-core implementation. Figure 4 shows the block diagram of the Styx namespace component. It makes use of a simple RAM based file system where each file has a limit of 256 bytes. The RAM-based file-system implements the record structure described in design section. Currently, the namespace implemented on the Spartan-IIE FPGA is contained within block RAM (BRAM) within the FPGA itself. This FPGA is contained within the BurchED B5-X300 board (14) and consists of files for devices connected to the FPGA on that board – LEDs, switches, 7 segment display, bell. This 512 byte BRAM-based namespace can be mounted by a remote Styx client through the serial line.

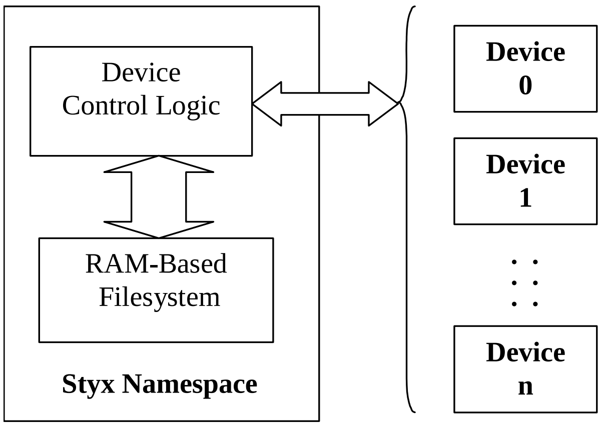

Figure 4 The Styx Namespace Component

In order to receive requests, certain control registers have been implemented within the input buffer. These registers can be accessed by external devices (e.g. CPU, UART) via the system bus. Along with the Styx IP-core component we have implemented a UART module to send/receive data on serial line, and a bus controller for the system bus to coordinate data flow between the IP-core and other devices. The authentication unit implements a simple non-encrypted authorisation mechanism involving string validation against user name/password and file access rights. The output buffer has been interfaced with the UART module via the system bus to send Styx messages through the serial line.

The decoder unit in the server IP-core has the following functionality:

- similar to the client decoder unit, the server decoder can receive instructions from other devices in the system. To maintain compatibility between client and server related instructions the server instruction codes start from 0x80 (hex). This helps distinguish the different instructions when the IP-core is used in dual mode (both as client and server). Table 2 lists the server IP-core instructions with description.
- on receiving any "T" messages from the Styx client, the decoder unit initially validates the messages using the authentication unit and then takes appropriate actions finally giving out signals to the encoder unit to generate and send the required Styx reply ("R") messages to the client. For instance, upon receiving a "Tversion" message, the decoder checks via the authentication unit if the version is same as the server's version and then signals the encoder unit to prepare and send an appropriate "Rversion" message as reply. In case of error in any unit the encoder is signalled to send a "Rerror" message describing the error encountered to the client.

Table 2 Instructions to the Server IP-core

Inst. Code (in Hex)	Description
0x80	Add new device/file in namespace as per information in data field
0x81	Delete a device/file in namespace given the file name in data field.
0x82	Set file permissions in authentication unit as per the data field.
0x83	Set user names/passwords in authentication unit as per the data field.
0x84	Set on-chip verification mode (if present) as per the data field.

On-chip Verification

As added functionality to verify the correct operation of the IP-core, an on-chip verification unit has been implemented. This is an optional unit which when used allows the system designer to verify the correct functioning of the various units in the IP-core. It operates in two modes – display only mode and debug mode. In display mode, the unit displays relevant information and data flow in each unit on a standard VGA display connected to the FPGA board. In debug mode, the system designer/developer can input debug commands to load a particular register, load the input buffer, clear the VGA screen, etc. These commands are given by varying the status of the 8 switches on the BurchED board. This feature helps in easy integration of the Styx IP-core with any kind of devices.

Usage Schematic

As described earlier, the Styx IP-core can be used either as a completely standalone component or along with a CPU. Figure 5 shows the architectural schematic of using Styx IP-core in the presence of a CPU.

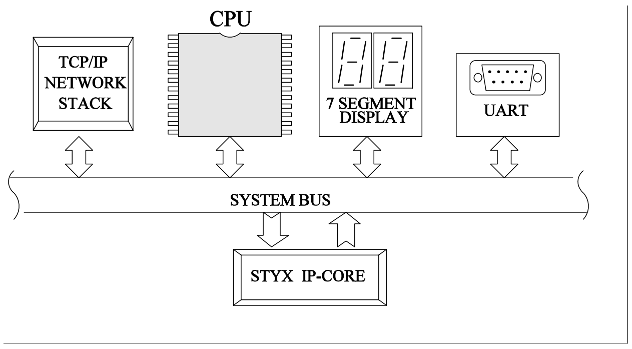

Figure 5 Architecture of Using Styx IP-core with CPU

Figure 6 shows a dedicated standalone Styx server IP-core plugged into an existing ubiquitous device.

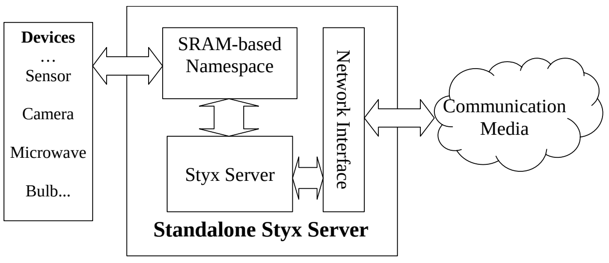

Figure 6 A Standalone Styx Server IP-core

PERFORMANCE ANALYSIS

Previous implementations of the Styx protocol were software, embedded into the Inferno OS. For performance comparison between software and hardware implementation we required a standalone Styx software solution that works independent of the Inferno OS. Hence, we implemented a standalone Styx server/client software component with its own namespace running independent of the Inferno OS on the Intel x86 architecture (Cyrix). This software component serves its namespace on the serial line.

Test Criteria

In order to evaluate the performance of the Styx IP-core over the Styx software component, we conducted the following test cases on both the software (standalone) as well as hardware IP-core Styx components:

- use an Inferno shell running on a different machine connected to the test machine on serial line to connect to the Styx server. Typically, we use the command -"mount /dev/eia0 /n/remote" to connect to a Styx server. This command makes the client send "Tversion" and "Tattach" messages to the server. The Styx server on authentication replies with the corresponding "Rversion" and "Rattach" messages.

- next we traverse through the mounted remote namespace and write to a file. This action makes the client generate "Twalk", "Topen", "Twrite" and "Tclunk" messages. Thus, the server needs to carry out any required action and reply the client with corresponding "R" messages. For better comparison we wrote to the file twice – initially with a short data (8 bytes) and then with considerably large data (256 bytes).

We record the time taken by the Styx server to decode each of the "T" messages from client and time taken to encode a reply ("R") message. The choice of the above test cases is particularly because they make the Styx component generate almost all the possible Styx messages allowing for detailed analysis. Comparing against the Styx standalone software component gives us precise measures of the performance improvements gained by the hardware implementation of Styx.

Styx Software Component

This was implemented on a Cyrix MediaGX 300MHz processor with 64MB SDRAM memory. The design and implementation of this software component is exactly similar to the hardware Styx IP-core described in the previous sections. When compiled, the standalone Styx component is 59KB in size. Table 3 shows the decode/encode time taken by the software only solution. Every received "T" message from the client must have a reply "R" message. Thus, each row in the table describes one complete cycle from "T" to "R" messages. The length field describes the lengths of the message received ("T") from client and the message sent ("R") to the client. The Decode time refers to the time taken by the server to decode the received ("T") message including the the time needed to carry out the required operations (e.g. open/read/write a file/device). The Encode time refers to the time taken by the server to encode and prepare the reply ("R") message. The Misc. field refers to the time spent by the server in doing other miscellaneous activities like book-keeping, device access, etc.

Hardware Styx IP-core

The VHDL modules of the hardware Styx IP-core component have been synthesised using Xilinx ISE (6) software, and implemented on BurchED B5-X300 board (14), containing a Spartan-2E 300 FPGA (6). The Styx IP-core when combined with the on chip verification module uses 63K gates and just 35K gates on the FPGA without the on chip verification module. We tested the Styx IP-core on our test board running at 25MHz. Applying the same test criteria to the hardware Styx IP-core we obtained the results as shown in Table 4. The similar length of messages in both software and hardware implementation confirms that it is compliant with the Styx protocol.

Message Type	Length (bytes)	Decoding Time	Encoding Time	Misc. Time	Total Time
(T/R)version	19(T)/ 19(R)	4.91	8.47	3.35	16.73
(T/R)attach	24(T)/ 20(R)	5.42	6.77	3.07	15.26
(T/R)walk	17(T)/ 35(R)	50.22	7.77	4.41	62.40
(T/R)open	12(T)/ 24(R)	3.5	8.37	3.08	14.95
(T/R)write (8 bytes)	33(T)/ 11(R)	657.35	5.22	1.65	664.22
(T/R)write (255) bytes	281(T) /11(R)	7315.4	7.34	1.19	7323.9
(T/R)clunk	11(T)/ 11(R)	2.93	4.30	2.16	9.39

Table 3 Performance of Styx Software Component (time in μs)

Message Type	Length (bytes)	Decoding Time	Encoding Time	Misc. Time	Total Time
(T/R)version	19(T)/ 19(R)	0.84	0.84	0.08	1.76
(T/R)attach	24(T)/ 20(R)	1.04	1.04	0.08	2.16
(T/R)walk	17(T)/ 35(R)	1.00	0.96	0.08	2.04
(T/R)open	12(T)/ 24(R)	0.56	1.20	0.08	1.90
(T/R)write (8 bytes)	33(T)/ 11(R)	1.40	0.52	0.08	1.90
(T/R)write (255) bytes	281(T) /11(R)	11.32	0.52	0.08	11.92
(T/R)clunk	11(T)/ 11(R)	0.52	0.45	0.08	1.05

Table 4 Performance of Styx Hardware Component (time in μs)

Demonstration

The demonstration mainly consists of three parts:

1. a Styx message processing unit (mounted on a robot);
2. a group of standalone (and static) Styx-aware ubiquitous devices;
3. a PC-based Styx GUI user application.

The PC (with GUI) has a WiFi connection, which enables communication to the Styx message processing unit. Users can use the GUI or an embedded Styx console to access the resources on the Styx message processing unit.

The Styx message processing unit is mounted on a robot chassis together with a digital camera module and three different wireless communication modules (WiFi, Bluetooth and ZigBee). The robot controller is mapped on the Styx namespace as a local resource, so that it user can control the robot by writing command to the robot controller file. Similarly, the digital camera module is also mapped to the name space as local resource, and the camera functions, such as clicking, resetting, can be achieved by written into camera control files. The image buffer of the camera is also within the namespace space to be read and displayed by the (remote) user GUI.

The group of standalone ubiquitous devices consist of static low-resource audio / visual units (i.e. each has audio output / input; LCD output). One standalone device has Bluetooth communication capability, the other has ZigBee.

Architecturally, the message processing unit (robot) acts as an intermediary between the PC and the standalone devices (as well as hosting devices of its own – e.g. camera). It contains CPU, Styx client and server IP-cores. The standalone devices contain only Styx server IP-cores and no CPU.

The Styx-aware ubiquitous devices are mounted to the same namespace as the remote resources. Once the Styx message processing unit receives a request of remote resources, it sends the incoming message to the destination (via ZigBee or Bluetooth).

The demonstration is implemented and functional. The Styx message processing unit occupies 41 slices of Xilinx Spartan-3 400 FPGA (6) (on a Opal Kelly XEM3001v2 board (15)), with a maximum clock rate of 79MHz. A 256k byte external SRAM module is used for image buffer of the digital camera. The ubiquitous devices utilise Xilinx Spartan-II 300E FPGAs for text / speech synthesising services. The user GUI software is developed in Java to allow compatibility between different platforms. It is delivered in approximately 4k lines of code.

Summary

The performance results show an increased performance of a hardware Styx IP-core over the software only component. It is also important to note that the software Styx was tested on a 300MHz CPU while the hardware executed at a mere 25MHz speed. Figure 7 plots a graph of the total cycle time values in tables and to assess the performance of Styx software and the Styx IP-Core. It can be seen that compared to the Styx software implementation, significant improvement has been made by the hardware IP-Core in terms of speed. For example, the total cycle time to “walk” to a file is 62.4μs and 2.04μs respectively for the software and hardware versions. Also, it is clear from the performance graph that the Styx IP-Core outperforms the software counterpart by several orders of magnitude. We also note that the Styx IP-core requires only 35K gates (without the on-chip verification module), whilst the software version requires a CPU (typically several orders of magnitude larger in size).

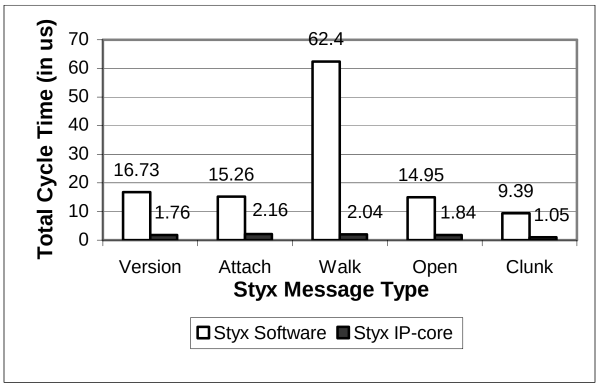

Figure 7 Timing of Software and Hardware Styx Components

CONCLUSIONS

This paper has described the importance of the Styx protocol to solve problems with respect to the interoperability between ubiquitous network devices, providing network and distribution transparency. To provide an efficient low-resource implementation of Styx, we have presented the design and implementation of a hardware Styx IP-core. This Styx IP-core component can act a co-processor module or an independent hardware module that can be transparently plugged into any existing communication hardware or software of a network communication device. The performance results demonstrate that the Styx IP-core component is fast and efficient compared to the software Styx component.

Current work is considering a standalone Styx client/server IP-core that adapts (during synthesis) to the application-specific requirements. The aim is to further improve the resource usage of the component by removing parts not required for a specific application. Also, more complex prototypes, including Styx client/server IP-core for standard bus architectures such as the Wishbone Bus (7), and processor IP-cores such as OpenRISC (8) are being developed.

REFERENCES

[1] Wright, G.R., and Stevens, W.R.,1995, TCP/IP Illustrated, Volume I and II

[2] Dorward, S., Pike, R., Presotto, D.L., Ritchie, D.M., Trickey, H., and Winterbottom, P., 1997 The Inferno Operating System, Bell Labs Technical Journal (1997)

[3] Audsley, N., and Patil, A., 2004, DEMOS-Implementing Operating System Communication Components on FPGA, Proceedings of the Embedded Real-Time Systems Implementation Workshop,

Lisbon, Portugal

[4] Breitstein, S.R 1997 Inferno Namespaces, Online White-Paper, Lucent Technologies

[5] Vita Nuova Online:http://www.vitanuova.co.uk

[6] Xilinx, online:http://www.xilinx.com

[7] Wishbone System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores Online:http://www.opencores.org/projects.cgi/web/wishbone/wishbone

[8] OpenRisc1200 Online:http://www.opencores.org

[9] Sharma, A., 2000, Inferno real-time capabilities Lucent Technologies, Bell Labs Innovations

[10] Tanenbaum, A., 2001, Modern Operating Systems, Prentice-Hall

[11] Silberschatz A. and van Steen V., 2001, Distributed Systems: Principles and Paradigms, John Wiley.

[12] Intel Corporation, 1998, Embedded Pentium Processor Family Developer's Manual

[13] Silberschatz A., Baer P. and Gagne G., 2005, Operating System Concepts, John Wiley

[14] BurchEd Co., online http://www.burched.com.au

[15] Opal Kelly Co., online http://www.opalkelly.com