SERVIDOR HTTP SEGURO EN UN SISTEMA EMBEBIDO

Oscar Alvarado Nava

Universidad Autónoma Metropolitana, Unidad Azcapotzalco

oan@correo.azc.uam.mx

Eduardo Rodríguez Martínez

Universidad Autónoma Metropolitana, Unidad Azcapotzalco

erm@correo.azc.uam.mx

Hilda María Chablé Martínez

Universidad Autónoma Metropolitana, Unidad Azcapotzalco

hmcm@correo.azc.uam.mx

Resumen

El presente documento describe la implementación de un servidor HTTP con acceso seguro, como un sistema embebido en un FPGA. El acceso al servidor light http es a través de un canal basado en *Secure Sockets Layer* (SSL), logrando que el servidor sea capaz de enviar y recibir información cifrada. El servidor almacena y recupera la información de una base de datos relacional implementada en el mismo sistema embebido y administrada por SQLite. El servidor y la base de datos son administrados a través del sistema operativo Linux, el cual fue compilado para el procesador PowerPC incrustado en el FPGA. El sistema muestra que es posible crear un servidor WEB embebido en un FPGA con soporte de bases de datos relacionales y acceso por canales seguros, logrando con ello un mayor nivel de seguridad en el almacenamiento y transmisión de información.

Palabra(s) Clave(s): FPGA, HTTPS, Linux, network, OpenSSL, SQLite, security.

1. Introducción

EL volumen de información intercambiada a través de los servicios WEB sigue aumentado de manera continua en los últimos años, siendo el comercio

electrónico uno de los principales usos. Los servicios relacionados con el comercio electrónico han generado la necesidad de proteger la integridad, disponibilidad y confiabilidad de la información durante su transmisión y recepción, siendo esta última la parte más vulnerable a amenazas de ataque como *man-in-the-middle* y *eavesdropping*. Estos ataques pueden permitir el acceso a un sitio WEB con información confidencial o interceptar la información enviada por el servidor a un cliente. Una medida para abatir estas amenazas es la creación de canales seguros entre el cliente y el servidor HTTP al cifrar la información. Por otro lado, el desarrollo de aplicaciones de cómputo sobre dispositivos embebidos tiene gran interés para la industria de dispositivos de bajo consumo de energía, ya que permiten crear sistemas de cómputo orientados a una tarea específica sin la necesidad de tener un sistema de cómputo completo. Un dispositivo autónomo capaz de ejecutar un servidor WEB a través de un canal seguro, puede ser de gran utilidad para aumentar la seguridad de los datos, así como minimizar el consumo de energía.

El esquema en capas mostrado en la Figura 1, permite comprender de manera práctica y a la vez concisa, los distintos niveles de abstracción para el diseño y desarrollo de sistemas embebidos.

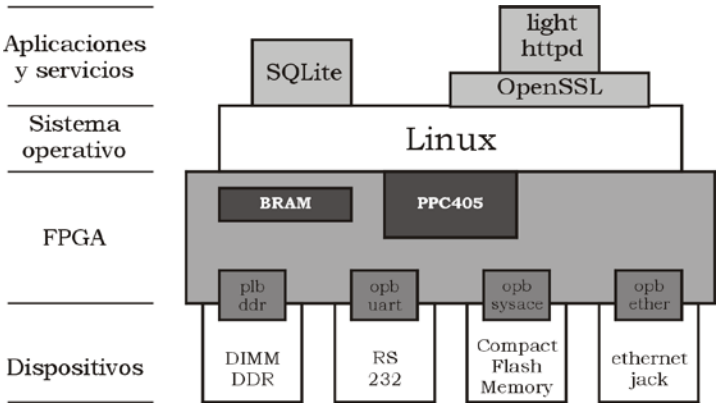

Figura 1 Sistema Embebido en Capas.

En la parte central del esquema se ubica el FPGA, el cual tiene incrustado un procesador de 32 bits (PPC405) y bloques de memoria DRAM (BRAM). A través de los CLBs (*Configurable Logic Blocks*) y los IOBs (*Input-Output Blocks*) del

FPGA, se programan tanto los dispositivos para la entrada y salida de datos como el sistema de buses para la interconexión de los mismos con el procesador. Por medio el procesador incrustado y los bloques de memoria (dentro y fuera del FPGA), ejecutar sistema operativo Linux, el cual es compilado para la arquitectura del procesador y contiene los controladores para el árbol de dispositivos creados. Las bibliotecas, servicios y aplicaciones que se ejecutarán sobre el sistema operativo, se compilan, se cargan y se configuran para generar la funcionalidad que ofrecerá el sistema embebido.

El presente trabajo se centra en la descripción del desarrollo de un sistema embebido basado en un FPGA, en el cual se ejecutan aplicaciones y servicios a través del sistema operativo Linux. Al sistema Linux se integró el servidor lighttpd [1], el manejador de base de datos ligero SQLite [2] y la biblioteca OpenSSL [3] para el cifrado de datos.

El documento está organizado de la siguiente forma: la Sección 2 presenta un estudio de los trabajos relacionados. La Sección 3 describe el sistema y las herramientas para el desarrollo del *hardware* y *software* del sistema embebido.

Los dispositivos que conformarán el sistema embebido y que serán programados dentro del chip del FPGA, se describen en la Sección 4. La Sección 5 describe el *software* del sistema operativo y su configuración. En la Sección 6 se explica la forma de obtener, configurar y compilar los servicios que ofrecerá el sistema embebido. Finalmente, en las Secciones 7 y 8 se describen los resultados y conclusiones respectivamente.

2. Trabajos Relacionados

En los sistemas de automatización es común utilizar los servidores WEB embebidos basados en FPGA para mejorar las funcionalidades de la Interfaz Humano–Computadora. Un protocolo seguro que es capaz de operar con recursos limitados en un sistema embebido es el HTTPS.

En el trabajo propuesto en [4] se describe el desarrollo de un sistema de control y monitoreo en tiempo real utilizando un servidor WEB implementado en el procesador programado (*soft processor core*) MicroBlaze. El sistema es capaz de

brindar acceso remoto a los clientes que utilicen un navegador WEB para recibir los enlaces descendentes, enviar comandos por enlace ascendente o actualizar el sistema.

En el trabajo propuesto en [5] se describe un servidor WEB orientado a sistemas operativos ligeros que puedan operar correctamente en una aplicación embebida. El servidor en tiempo real es utilizado para control y acceso de datos remoto. Se analizan dos elementos básicos al implementar el servidor: el desempeño del sistema operativo ligero y el principio de comunicación HTTP. Para satisfacer los requerimientos prácticos de la aplicación, el servidor es diseñado para obtener el mayor desempeño posible con los recursos limitados del sistema.

En el trabajo presentado en [6] se propone una arquitectura de un servidor WEB basado en FPGA eficiente y de bajo costo. Se utiliza el microprocesador 8051 para la ejecución del servidor WEB y los protocolos TCP/IP. Las páginas WEB se trasmiten por peticiones HTTP en un módulo de transmisión implementado en Verilog.

Otra métrica de medición importante en las aplicaciones WEB embebidas es el consumo energético. En el trabajo presentado en [7] se presenta un análisis de medición de consumo energético para aplicaciones embebidas web en el área de Internet de las Cosas. Particularmente centra su análisis en el consumo energético utilizando el sistema operativo embebido Contiki para servidores web HTTP.

3. Plataforma de Desarrollo

El proceso de diseño de un sistema embebido considera el desarrollo de *software* y el desarrollo de *hardware*, los cuales pueden darse de manera simultánea o de manera independiente, integrándose en la fase de implementación o programación del dispositivo. Una característica importante de las herramientas de desarrollo actuales, es que es posible desarrollar una aplicación completa sin considerar el dispositivo objetivo, siendo únicamente necesario cambiar algunos parámetros para la implementación del proyecto en un dispositivo con una tecnología específica.

La plataforma de desarrollo fue conformada principalmente por el crosstool-ng [8] y las herramientas de Xilinx ISE (*Integrated Software Environment)* 10.1 y el EDK (*Embed-ded Development Kit*) 10.1, los cuales se instalaron sobre una computadora de escritorio con la distribución Debian GNU/Linux 7.2. El ISE es utilizado para el análisis y síntesis de proyectos desarrollados con lenguajes de descripción de hardware (HDL), el cual incluye simuladores funcionales y analizadores de desempeño en el tiempo. A través del Xilinx EDK se desarrolla tanto el *software* del proyecto como la configuración del sistema cómputo embebido, seleccionado por ejemplo ya sea un *hard processor core* y/o un *soft processor core*; así como la cantidad de bloques de SRAM y los IP (*Intellectual Property*) *cores* que funcionan como periféricos del sistema de cómputo embebido.
Una *toolchain* es un conjunto de herramientas de desarrollo de *software* que son ligadas o encadenadas por niveles. Los elementos mínimos son un cargador, un ensamblador, un compilador y bibliotecas de desarrollo. Opcionalmente una *toolchain* puede contener un depurador o un compilador para un lenguaje de programación específico. Generalmente la *toolchain* utilizada para el desarrollo de sistemas embebidos es una *toolchain* cruzada, ya que su compilador soporta compilación cruzada. Al proceso de compilar un programa para una plataforma distinta a la que se lleva a cabo la compilación, se le denomina compilación cruzada.
Para construir la *toolchain* se utilizó buildroot [9] la cual es una herramienta para compilar y generar sistemas de archivos, *kernels* e incluso *bootloaders* con una configuración relativamente sencilla. Buildroot hace uso de busybox [10] que es un conjunto de herramientas populares y acertadamente adaptadas para el desarrollo de sistemas embebidos. Busybox es un binario independiente que proporciona soporte para muchas utilidades comunes de línea de comandos de Linux. La obtención, configuración y compilación se muestra en la figura 2.
Instalar la plataforma de desarrollo sobre un sistema estable, libre y fácilmente replicable como Debian GNU/Linux, es de gran ayuda para obtener las herramientas de *software* básicas y su documentación sin que esto represente un gasto económico extra. Dichas herramientas de *software* van desde un manejador

de versiones distribuido, hasta el uso aplicaciones y herramientas más complejas como un compilador cruzado, depuradores, bases de datos relacionales, etc.

```
user@host:~/toolchain$wget http://git.buildroot.net/\
   buildroot/buildroot-2013.02.tar.bz2

user@host:~/toolchain$tar -jxvf buildroot-2013.02.tar.bz2

user@host:~/toolchain$cd buildroot-2013.02

user@host:~/toolchain/buildroot-2013.02$make menuconfig

user@host:~/toolchain/buildroot-2013.02$make
```

Figura 2 Obtención y configuración de buildroot.

4. *Hardware* del Sistema

Tarjeta XUPV2P

Se utilizó la tarjeta desarrollo XUPV2P [11] distribuida por la empresa Digilent [12], la cual cuenta con un FPGA Virtex-II Pro (XC2VP30) y dispositivos para el desarrollo de aplicaciones, destacando 10/100Mbps Ethernet PHY, una ranura para una tarjeta Compact Flash, dos ranuras para DIMMs de memoria DDR-SRAM, y un puerto RS-232, entre muchos otros.

Para generar el árbol de dispositivos a través del *Board Support Package* (BSP) del EDK, se utiliza el *Device Tree Generator* el cual se puede obtener a través de la herramienta git en el sitio Github de Xilinx [13]. Además es necesario los archivos de configuración de la tarjeta de desarrollo. En la figura 3 se muestra las líneas de comando para obtener las herramientas y los archivos de configuración del *hardware* de la tarjeta de desarrollo. Cabe mencionar que es necesario obtener una licencia de uso para el dispositivo OPB_ETHERNETLITE, esta solicitud se hace en el sitio de Xilinx con registro previo.

```
user@host:~/xup2vp$git clone git://github.com/xilinx/\
   device-tree.git

user@host:~/xup2vp$wget http://www.digilentinc.com/\
   Data/Products/XUPV2P/EDKXUP-V2ProPack.zip
```

Figura 3 Herramientas y archivos para la creación del árbol de dispositivos.

Sistema en un chip

El FPGA XC2VP30 además de contar con una gran densidad de CLBs para la implementación de circuitos digitales, tiene incrustados dos *hard core* PowerPC-

405 y varios bloques de SRAM que en conjunto suman 2 MBytes. Para que este sistema CPU-RAM sea capaz de ejecutar programas más grandes y complejos, como un sistema operativo moderno, es necesario incluir módulos DDR-SRAM en las ranuras de la tarjeta XUPV2P.

La elección del tipo procesador (*hard core* o *soft core*) es relevante tanto para el desempeño y como para la utilización de recursos del FPGA. Por ejemplo, al utilizar un *hard core* permite la inclusión de más componente de *hardware* o periféricos, proporcionando una mayor funcionalidad al proyecto. Además de la elección del procesador, es necesario generar el árbol de dispositivos dentro del proyecto de *hardware*.

En el FPGA XC2VP30 se creó el sistema de cómputo embebido mostrado en la figura 4, el cual está compuesto del *hard core* PowerPC 405 a 400Mhz (ppc405), todos los bloques de memoria SRAM *On-Chip* de 128KBytes (bram_block) con su respectivo módulo de acceso al bus (plb_ram), un sistema de buses jerárquico compuesto del bus local del procesador (plb), del bus de periféricos en chip (opb) junto con sus respectivos árbitros y un puente entre buses (plb2opb). Para el almacenamiento de la información fue necesario agregar el mdulo (opb_sysace) para el acceso a un sistema de archivos ext2 sobre una *Compact Flash Memory* de 1 GByte. Así mismo, para poder ejecutar el sistema operativo Linux se agregó al espacio de direcciones del procesador un módulo de memoria externo DIMM DDR SDRAM de 256 MBytes (plb_ddr). Se incluyó un transmisor-receptor serial para la entrada y salida estándar del sistema (opb_uart). El *hardware* anterior y sus parámetros de configuración se listan en el archivo xilinx.dts, generado por el EDK.

5. Software del Sistema

Linux Kernel

El sistema operativo Linux deberá ser compilado para la arquitectura objetivo, ya sea para el PowerPC o para el *soft core* MicroBlaze [14], así mismo deberá contener los controladores para el árbol de periféricos configurado.

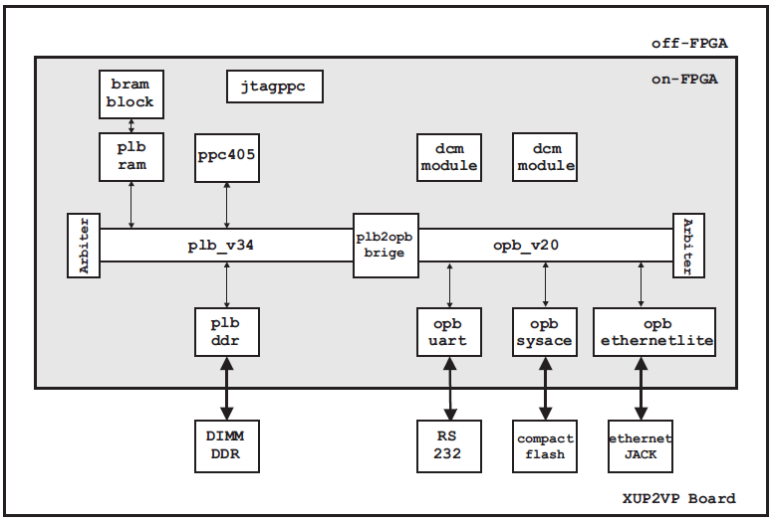

Figura 4 Hardware del sistema embebido basado en un FPGA.

Las fuentes del *kernel* de Linux se pueden obtener del sitio oficial [15], sin embargo, para este proyecto utilizamos la versión linux_xlnx, la cual es una versión del *kernel* 3.2 modificada para sistemas embebidos y soporta el *hardware* mostrado en figura 4. Para que el *kernel* pueda ser cargado, es necesario un *ramdisk*, el cual es un sistema de archivos en memoria. Ambos elementos también están disponible en el Github y el repositorio de binarios de Xilinx.

Las líneas 1 y 4 de la figura 5 muestran los comandos para obtener las fuentes del *kernel* y el *ramdisk*. La línea 10 de la misma figura 5 indica cómo se lleva a cabo la relación del *hardware* y las fuentes del sistema operativo, solamente es necesario copiar el archivo que contiene la configuración del árbol de dispositivos con el prefijo virtex405- y el *ramdisk*, al directorio de arranque.

```
1  user@host:~/kernel$git clone git://github.com/xilinx/\
2     linux-xlnx.git
3
4  user@host:~/kernel$wget http://xilinx.wdfiles.com/\
5     local-files/powerpclinux/ramdisk.image.gz
6
7  user@host:~/kernel$cp ~/xup2vp/xilinx.dts\
8     linux-xlnx/arch/powerpc/boot/dts/virtex405-https.dts
9
10 user@host:~/kernel$cp ramdisk.image.gz\
11    linux-xlnx/arch/powerpc/boot
```

Figura 5 Kernel Linux y sistema de archivos en RAM y relación con hardware del sistema.

El *kernel* de Linux tendrá los subsistemas principales, como el *Scheduler*, el *Virtual File System*, el *Memory Manager* y el *Inter Process Comunication*, así como los controladores necesarios para el acceso a los dispositivos. Ya que los controladores pueden ocupar una gran parte de la imagen, es importante configurar adecuadamente los parámetros de compilación para soportar sólo *hardware* seleccionado y las funcionalidades del *kernel* necesarias en el proyecto.
Para la compilación cruzada del *kernel* de Linux, es necesario crear variables de ambiente para que la *toolchain* acceda a las herramientas y configuraciones. En la figura 6 se muestra un fragmento del contenido del archivo ambiente.sh el cual crea el ambiente para la compilación cruzada.

```
unset CC CXX CPP CFLAGS INCLUDES CXXFLAGS LD_LIBRARY_PATH
    LIBRARY_PATH CPATH
export CC CXX CPP CFLAGS INCLUDES CXXFLAGS LD_LIBRARY_PATH
    LIBRARY_PATH CPATH
unset ARCH
export ARCH=powerpc
unset CROSS_COMPILE
export CROSS_COMPILE=powerpc-linux-gnuexport
PATH=/path/to/toolchain/bin:$PATH
```

Figura 6 Archivo *ambiente.sh.*

La figura 7 muestra los comandos para la configuración y compilación cruzada del *kernel* de Linux.

```
user@host:~/linux-xlnx$source ~/toolchain/ambiente.sh

user@host:~/linux-xlnx$source ~/Xilinx/settings.sh

user@host:~/linux-xlnx$source ~/EDK/settings.sh

user@host:~/linux-xlnx$make menuconfig

user@host:~/linux-xlnx$make simpleImage.initrd.virtex405-
    https

user@host:~/linux-xlnx$cp arch/powerpc/boot/\
   simpleImage.initrd.virtex405-https.elf ~/xupv2p
```

Figura 7 Comandos para la compilación cruzada del *kernel* de Linux.

El resultado de la compilación es la imagen vmlinux y los archivos System.map y simpleImage.initrd.virtex405-nombre.elf, el cual deberá ser copiado el proyecto creado en el EDK y así pueda ser programado al FPGA a través del sistema SysACE o XDM.

Root file system

Un sistema de archivos jerárquico o *Root File System* (RFS), es la clasificación de los archivos de las aplicaciones por su tipo, función y acceso de acuerdo a un estándar. El RFS es una parte muy importante en el desarrollo de un sistema embebido basado en el *kernel* de Linux, ya que a diferencia de otros sistemas operativos embebidos, Linux requiere de un sistema de archivos para funcionar.

Ya que el espacio de almacenamiento también es limitado en un sistema embebido, el crear un sistema de archivos pequeño, pero suficiente para almacenar los archivos necesarios ha sido una gran preocupación para los desarrolladores. El *Filesystem Hierarchy Standard* (FHS) [16] establece el sistema de archivos mínimo con el que debe contar un sistema Linux para poder funcionar, el cual es mostrado en la tabla 1.

Tabla 1 Sistema de Archivos Mínimo.

Directorio	Descripción
/bin	Comandos comunes, compartidos por el administrador y los usuarios del sistema.
/dev	Referencias a todos los periféricos de hardware que son representados como archivos especiales
/etc	Principales archivos de configuración tanto para aplicaciones del sistema como de servicios
/lib	Archivos bibliotecas de aplicaciones de sistema
/sbin	Aplicaciones para el sistema y el administrador
/tmp	Espacio para archivos temporales
/var	Archivos de sistema y de usuarios de tamaño variable como bitácoras, spoolers, etc.

El *kernel* podría ser compilado para contener una versión comprimida del RFS y de esta forma no necesitar de un dispositivo externo para almacenar los archivos, de manera similar a cómo funcionan los sistemas operativos *live*. A pesar de que con un RFS cargado en memoria se podría obtener una mejora en el rendimiento y en el almacenamiento, no siempre es recomendable ya que, si no se hace un análisis minucioso de los archivos requeridos, la imagen podría alcanzar un

tamaño considerable y consecuentemente no podría cargarse en la limitada RAM del sistema embebido.

La solución comúnmente utilizada para poder contar con un sistema de archivos completo, es crear inicialmente un sistema de archivos pequeño en un dispositivo de arranque (como una memoria flash) y posteriormente montar un sistema de archivo más grande almacenado en otro dispositivo de mayor capacidad, o incluso montarlo desde un servidor NFS a través de una red de datos. El RFS se puede montar desde el proceso de arranque del *kernel* al indicarle por medio del *Kernel Command Line* (KCL) el dispositivo y en el cual está el RFS.

En el presente proyecto se creó un sistema de archivos mínimo almacenado en una partición ext2 de 750MB sobre una Compact Flash de 1 GByte. Las principales aplicaciones utilizadas en el proyecto se listan en la tabla 2.

Tabla 2 Aplicaciones utilizadas en el proyecto.

Tamaño	**Empleo**
OpenSSL	Conjunto de bibliotecas de criptografía.
OpenSSH	Conjunto de herramientas de para la comunicación segura en redes.
lighttpd	Servidor web ligero.
tcl	Lenguaje de script de sintaxis sencilla
sqlite3	Sistema ligero de gestión de para bases de datos relacionales
dhclient	Cliente DHCP
iptables	Programa de usuario y módulo del kernel para la administración de firewalls

Handering

Una vez programado el FPGA y lanzado el sistema operativo, se realizan configuraciones propias del sistema, las cuales van desde la seguridad del mismo hasta el inicio ordenado de los servicios. Las configuraciones se realizan por medio de *scripts* de *shell* y se encargan de probar e iniciar dispositivos, acceder a archivos de configuración, lanzar las aplicaciones e iniciar bitácoras.

Las primeras configuraciones consisten en la fortificación o disminución de la vulnerabilidad del propio sistema (*Hardening*), el cual consiste por ejemplo, en

deshabilitar o remover servicios innecesarios e iniciar en una secuencia adecuada los servicios que se requieren para que el sistema realice su función. Una vez iniciado el servicio de red, se activa un *firewall* a través *iptables*. Las reglas en *iptables* tienen la siguiente finalidad: filtrar o desechar todo el tráfico que no esté asociado a algún servicio activo en el sistema, el reenvío (*forwarding*) de paquetes queda deshabilitado y todo el tráfico saliente del sistema es permitido. Adicionalmente se activa en el *kernel* del sistema el *Reverse Path Filtering* (rp_filter) para evitar ataques de suplantación (*spoofing*) dentro de la red en la que se encuentra el sistema embebido. Así, solo se permite el tráfico de *loopback* y de los servicios SSH y el servicio HTTPS creado. El servicio SSH permite tanto la administración del sistema de manera remota como el envío y recepción segura de archivos.

6. Servicios y Aplicaciones

Las servicios y aplicaciones que se ejecutarán en el sistema embebido, deberán ser compilados (de manera cruzada) y agregados en el sistema de archivos del sistema embebido. Esta compilación se puede hacer cuando se está creando el sistema de archivos del sistema, como se describió en la Sección 5-B, o bien se puede compilar por separado y copiar los binarios al sistema de archivos.

Debe ser claro que para cada servicio, se deberán de resolver las dependencias que requieran, esto es, se deberá agregar el código dependiente para su compilación en el orden necesario y su posterior enlazado. En las secciones siguientes se comentará los detalles para la compilación de los principales servicios.

OpenSSL

OpenSSL es esencialmente un conjunto de bibliotecas para el cifrado de datos y herramientas con las cuales es posible la implementación SSL. Las bibliotecas de cifrado nos ofrece los algoritmos más utilizados para cifrado simétrico y de llave pública, algoritmos *hash* y para resumen de mensajes. Incluye también un

generador de números pseudoaleatorios y soporte para la manipulación de formatos comunes de certificados y material para el manejo de claves.

El código fuente de OpenSSL se puede obtener del sitio oficial o de alguna distribución de Linux. Por ejemplo, se puede extraer el código fuente de los paquetes de Linux Debian para PowerPC ejecutando los comandos que se muestran en la figura 8. Es necesario considerar las dependencias del código, en este caso, libssl, libssl-dev y xlib1g. Para la compilación de c´odigo y enlazado de módulos de la biblioteca OpenSSL, se carga el ambiente de la *toolchain* cruzada para su ejecución. El comando mostrado línea 3 de figura 9, compila el código del servidor HTTPS y lo enlaza con la biblioteca de OpenSSL.

```
1 user@host:~/ssl$wget http://ftp.us.debian.org/debian/\
2    pool/main/o/openssl/\
3    libssl0.9.8_0.9.8o-4squeeze14_powerpc.deb
4
5 user@host:~/ssl$dpkg -x
6    libssl0.9.8_0.9.8o-4squeeze14_powerpc.deb ./
```

Figura 8 Comandos para obtener el código fuente de OpenSSL para PowerPC.

```
1 user@host:~/ssl$source ~/toolchain/ambiente.sh
2
3 user@host:~/ssl$powerpc-linux-gnu-gcc -Wall\
4    common.c server.c wserver.c -o httpsServer\
5    -pthread -lssl -lcrypto
```

Figura 9 Compilación cruzada y enlazado de biblioteca de OpenSSL.

Finalmente, los binarios y sus archivos de configuración, son copiados al sistema de archivos del sistema embebido como se muestra en la figura 10.

```
1 user@host:~/ssl$mount /dev/'deviceCompactFlash' /mnt
2
3 user@host:~/ssl$mkdir /mnt/root/servicio
4
5 user@host:~/ssl$cp httpsServer\
6    dhparam.pem root.pem server.pem\
7    /mnt/root/servicio/
```

Figura 10 Compilación cruzada y enlazado de biblioteca de OpenSSL.

SQLite

SQLite es un motor de bases de datos relacionales para sistemas de bajos recursos. Lo que hace diferente a SQLite es que un solo proceso realiza lecturas y escrituras directamente sobre los archivos que contiene la base de datos. Esto reduce el tiempo de acceso a la base de datos debido a que no hay comunicación entre procesos. Además, debido a que es un conjunto de funciones, es ideal para sistemas de bajos recursos, como los sistemas embebidos.

Se creó una base de datos que contiene información que podemos considerar confidencial que será accedida por el servidor HTTPS con sentencias SQL y serán enviadas al cliente a través de funciones OpenSSL. De forma similar a OpenSSL, el código de SQLite se obtiene, se compila y se copia al sistema de archivos del sistema embebido.

Servidor HTTPS

Un servidor HTTPS es una versión segura de un servidor HTTP, ya que implementa un canal de comunicación basado en SSL entre el navegador del cliente y el servidor HTTP. Este canal requiere que antes del envío de datos, el servidor y el cliente realizan siguiente negociación:

- El cliente envía al servidor las opciones de cifrado, compresión y versión de SSL junto con algunos bytes aleatorios llamados *Challenge* de Cliente.
- El servidor selecciona las opciones de cifrado, compresión y versión de SSL entre las que ha ofertado el cliente y le envía su decisión y su certificado.
- Servidor y cliente negocian la clave secreta llamada *master secret*.
- Utilizando la clave *Challenge* de Cliente y las opciones pactadas, se envía la información cifrada.

Un certificado de clave pública es un documento que certifica que el interlocutor es quien realmente dice ser, esto se hace para evitar que un atacante pueda hacerse pasar por el servidor y recibir la comunicación segura en su lugar. Estos certificados pueden generarse con herramientas de OpenSSL y para una mayor seguridad pueden ser firmados por una autoridad certificadora, por ejemplo:

VeriSign, *Thawte*, *GoDaddy* o *GeoTrust*. Los certificados también pueden ser auto-firmado, pero el cliente no tendrá total seguridad que la información está siendo enviada al servidor correcto y recibirá un llamativo aviso en su navegador de que el certificado no es de confianza.

En la figura 11 se muestra a manera de ejemplo, los comandos para crear el certificado del servidor y firmarlo con Autoridad Certificadora (CA) del servidor utilizando las herramientas de OpenSSL.

```
user@host:~/xup2vp/certificados$openssl req -newkey\
   rsa:1024 -sha1-keyout serverkey.pem -out serverreq.pem

user@host:~/xup2vp/certificados$openssl x509 -days 3650\
   -req -in serverreq.pem -sha1 -extfile\
   /etc/ssl/openssl.cnf -extensions\
   usr_cert-CA serverCA.pem -CAkey serverCA.pem\
   -CAcreateserial -out servercert.pem

user@host:~/xup2vp/certificados$cat servercert.pem\
   serverkey.pem serverCAcert.pem\
   rootcert.pem >> server.pem

user@host:~/xup2vp/certificados$openssl x509 -subject\
   -issuer -noout -in server.pem
```

Figura 11 Creación de Certificados con OpenSSL.

La implementación del servidor HTTPS se logró al agregar al código del servidor lighttp una capa de seguridad implementada con las funciones de OpenSSL. A continuación, se hace una breve descripción de la creación de la capa de seguridad. Inicialmente se crea un contexto de seguridad (SSL_CTX) y se le cargan las bibliotecas (SSL_load_error_strings()) necesarias. Después se hace la autenticación con los certificados (SSL_CTX_use_certificate_chain_file()), las llaves (SSL_CTX_use_PrivateKey_file()) y los *passwords* (SSL_CTX_set_default_passwd_cb()).

7. Resultados

Una vez programado el hardware en el FPGA, se lanza el *kernel* de Linux y crea los servicios básicos a través de sus archivos de configuración. El servicio

HTTPS se puede lanzar de manera manual o de manera automática por medio de un script.

Una vez generada la solicitud de un cliente y antes de realizar la transferencia de información se establece un *handshake* en el cuál se negocian y se establecen los parámetros e intercambio de claves que se utilizarán para el transporte de nuestra información y creando así, para nuestro caso, un canal de comunicación seguro bajo OpenSSL.

Una vez que ha terminado la negociación, se recibe y analiza la solicitud dando paso a la consulta en nuestra base de datos por medio de una sentencia SQL de donde es extraída la información solicitada en caso de existir. Realizada nuestra consulta se procede a dar respuesta a la solicitud, enviando la información requerida al cliente para posteriormente cerrar el canal de comunicación. Este proceso se lleva a cabo por cada petición de información al servidor, el cuál fue diseñado para atender solicitudes concurrentemente. En la figura 12 se muestra un fragmento de la bitácora del servidor.

```
[root@xupv2p servidor]#./https

**************************************
Escuchando por el puerto 4433...
GET /index.html HTTP/1.1
Host:192.168.1.9:4433
User-Agent:Mozilla/5.0 (X11; Linux i686; rv:24.0) Geko
    /20100101 Firefox/24.0
Accept: text/html, aplication/xhtml+xml,application/xml;q
    =0.9,*/*;q=0.8
Accept-Languaje: es-MX,es-ES;q=0.8,es-AR;q=0.7,es;q=0.5,en-
    US;q=0.3,en;q=0.2
Accept-Encoding: gzip, deflate
Connection: keep-alive
If-Modified-Since: Wed 19 Mar 2014 01:34:12 GMT
Cache-Control: max-age=0
Ejecutando SQL (select from )...
```

Figura 12 Bitácora del HTTPS.

8. Análisis de Resultados

La implementación del servidor HTTPs presentada, está totalmente apegada al RFC5246 (The Transport Layer Security (TLS) Protocol v1.2) el cual utiliza un intercambio de llaves con el algoritmo Delffie-Hellman y cifrado asimétrico con RSA. Ambos algoritmos requieren aritmética entera, por lo que la carencia de un

FPU (Unidad de número de Punto Flotante) en el procesador incrustado PPC405 no implican una diferencia importante en cuanto al desempeño del sistema, sin embargo, hay una serie de aspectos técnicos que si influyen en el rendimiento del sistema desarrollado en comparación con un sistema de cómputo convencional. Uno de los más evidentes es la diferencia entre las frecuencias de operación entre los distintos procesadores, el procesador embebido PPC405 tiene una frecuencia de operación de entre 100MHz y 500MHz máximo, siendo la frecuencia de operación recomendada de 200Mhz, mientras que por ejmplo un procesador Intel Atom x86 de gama baja, tiene una frecuencia de operación mínima de 1.6GHz.

La interfaz de red en un sistema de cómputo convencional es dedicada y en muchas ocasiones está integrada con la tarjeta madre a un bus de alta velocidad. En el sistema embebido presentado, la parte que implementa la interacción con los protocolos de la capa de enlace de datos están implementadas dentro de un módulo instanciado en los CLBs del FPGA, los módulos con mejor rendimiento son propietario y tienen un costo y no estuvieron disponibles para este trabajo.

9. Conclusiones

El presente proyecto demuestra que es posible crear un servicio HTTPS con acceso a una base de datos relacional en un solo chip. Enlazando bibliotecas orientadas al cifrado, fue posible crear un canal de comunicación seguro y siendo la seguridad un tema central en nuestros días, es una alternativa para crear aplicaciones que requieren confidencialidad de datos.

Al estar embebido en un chip el servicio HTTPS adquiere grandes ventajas, entre ellas, un bajo consumo de energía y un nivel de vulnerabilidad mínimo. Además, al contar con un sistema operativo el sistema embebido logra una gran flexibilidad en su configuración y modificación, permitiendo mejorar su funcionamiento en un tiempo de desarrollo corto.

El sistema descrito fue desarrollado en un FPGA ejecutando Linux, sin embargo, con los IDEs de desarrollo actuales es posible migrar fácilmente el proyecto a otras arquitecturas o plataformas más poderosas y populares como MIPS o ARM y ganar con ello un valor agregado en la industria.

Es importante señalar que generalmente el precio a pagar por una gran flexibilidad, es un impacto negativo sobre el rendimiento del sistema. Por ejemplo, el uso de un lenguaje de programación interpretado ayuda en la integración de aplicaciones mejorando el tiempo de desarrollo, pero debido a las numerosas capas de *software* necesarias, aumenta el tiempo de ejecución de la aplicación.

10. Trabajo Futuro

Una posibilidad es instanciar varios procesadores *soft core* en un FPGA con mayor densidad de CLBs, y asignar cada uno de ellos un servicio, utilizando un *kernel* con soporte *Symmetric Multi Processing*. Otra posibilidad es utilizar procesadores incrustados con mayores recursos, como ejecución superescalar y crear servidores multihilo. Aumentar el nivel de seguridad con llaves de mayor longitud, verificadas por coprocesadores programados en el hardware del FPGA.

11. Bibliografía y Referencias

[1] Home – lighttpd – fly light. http://www.lighttpd.net/. Jan. 2014.

[2] Sqlite home page. http://www.sqlite.org/. Jan. 2014.

[3] R. S. Engelschall. Openssl: The open source toolkit for ssl/tls. http://www.openssl.org/. Jan. 2014.

[4] A. Hanafi, M. Karim, "Embedded web server for real-time remote control and monitoring of an FPGA-based on-board computer system". Intelligent Systems and Computer Vision (ISCV). Fez. 2015. Pp. 1-6.

[5] L. L. Wang, P. F. Zeng, "A lightweight operating system-oriented web server realization". Wavelet Active Media Technology and Information Processing (ICCWAMTIP). 10th International Computer Conference on, Chengdu. 2013. Pp. 186-190.

[6] J. Zhang, J. Tian, "Design and implementation of an efficient web server based on FPGA". Computer Science and Network Technology (ICCSNT), 2nd International Conference on, Changchun. 2012. Pp. 172-175.

[7] N. Cherifi, G. Grimaud, T. Vantroys, A. Boe, "Energy Consumption of Networked Embedded Systems". 3rd International Conference on Future Internet of Things and Cloud (FiCloud). 2015. Pp. 639-644.

[8] Crosstool ng. http://www.crosstool-ng.org/. Jan. 2014.

[9] E. Andersen. Buildroot. http://buildroot.uclibc.org/. Jan. 2014.

[10] Busybox. http://busybox.net/. Jan. 2008.

[11] Xupv2p documentation. http://www.xilinx.com/univ/xupv2p.html. Jan.2012.

[12] D. Inc. Digilent Inc. - digital design engineer's source. http://www.digilentinc .com/. Jan. 2014.

[13] K. Sievers. Xilinx github. https://github.com/xilinx. Jan. 2014.

[14] Xilinx. Microblaze soft processor. http://www.xilinx.com/tools/microblaze.htm Jan. 2012.

[15] Linux Kernel Organization. The linux kernel archive. http://www.kernel.org/. Jan. 2014

[16] D. Quinlan. Filesystem hierarchy standard. http://www.pathname.com/fhs. Jan. 2004.