Bringing TCP/IP Networking to Resource Limited Embedded Systems

Roman Glistvain, and Mokhtar Aboelaze, *Senior Member, IEEE*

Abstract— The TCP/IP network interface is considered a *luxury* feature in many embedded devices because it is expensive to implement. It requires additional hardware and complex software to enable the device to be accessible on the local area network or over the Internet. This paper describes a simple way of integrating a simplified TCP/IP stack with the Romantiki operating system. Our approach allows the user to use standard multithreaded approach to write application code, and at the same time use a small memory footprint similar to super loop based applications. This approach paves the way to have complex devices running standard TCP/IP applications on a very inexpensive and energy efficient microcontrollers with a small RAM and Flash memories.

I. INTRODUCTION

Up to a few years ago computers were the only electronic devices connected to a network in an average household. TVs, DVDs and VCRs were devices with dedicated functionality and without the need to be connected to a network. Today with the expansion of Wi-Fi routers, Netflix streaming, smart appliances – many devices have TCP/IP connectivity options and need to be controlled, monitored and troubleshot via TCP/IP. Currently the approach to these devices is to use high end microprocessors and complex software architecture, running operating systems such as Linux, Windows or VxWorks. These operating systems provide a standard multithreaded paradigm of writing applications and include an integrated TCP/IP stack with a standard "Socket API". This approach allows integrating existing off-the shelf open-source or commercial components into embedded devices at the expense of the increased complexity, increased price, slow boot-up time and slow operation of such devices. The result of this approach is a slow adoption of networking in certain devices. For example – it will be hard to sell a TV set which takes 1 min to boot-up. However, the TV set is a good example of a device which could benefit from TCP/IP networking. A TV can stream live video from On-Line TV stations, YouTube, Netflix and other services.

Low end embedded networking microcontrollers are starting to gain popularity and replace complex high end microprocessors in embedded devices requiring simple networking functionality. These microcontrollers are characterized by a simple processor with a limited processing power, limited resources, and low power consumption. They also have a low-to-moderate size memory without a Memory management Unit (MMU). These processors require special attention in developing application programs for them. For example STM32F105R8T6 [19] microcontroller with an embedded Ethernet core has 64K byte of programmable flash and 20K byte of SRAM.

We proposed Romantiki [9] as a simple operating system that can run on embedded devices with very limited memory resources, and at the same time can bring TCP/IP functionality to these resource-limited microcontrollers. Romantiki offers the feature-set of a standard cooperative multitasking operating system but it requires a very small amount of memory compared to a standard operating system such as Linux or Windows.

Since Romantiki is directed at small microcontrollers with limited memory resources, our design requirement for Romantiki is to satisfy the following important criteria:

First, we want our proposed operating system to have a fast startup time.

Second, a small footprint in both RAM and Flash, since small microcontrollers are characterized by a small on-chip memory.

Third we would like our proposed operating system to work with applications that require soft real time constraints. This means we have to have prioritization in scheduling.

Fourth we would like to present a multitasking OS model to the user where a single stack is used for all tasks [10]. We also require that applications developed for Romantiki can be easily ported to other operating systems with a minimum of modifications (zero modifications in most of the cases).

Finally, we would like to present to the user a "socket-like" API to support networking applications. Another objective here is to create a common networking abstraction layer common to Romantiki and traditional operating system such that applications written for Romantiki can be recompiled to run on other operating systems without any changes in the source code.

The remainder of the paper is organized as follows: Section II describes the motivation behind our work and discusses related work. Section III presents an overall view of Romantiki operating system with emphasis on the TCP/IP components. Section IV describes the simplified TCP/IP stack in Romantiki OS and compares it with the traditional full strength TCP/IP stack. Section V provides performance

R. Glistvain is currently with Turck USA in Minneapolis, MN USA. (email : romangl@email.com).

M. Aboelaze is with the department of Computer Science and Engineering, York University Toronto, Ontario. Canada. (email: aboelaze@cse.yorku.ca)

and footprint comparison between a similar networking application running on FreeRTOS and Romantiki OS. Section VI describes future work and conclusion.

II. Motivation and Previous Work

A. Motivation

In his paper we focus on resource-constrained embedded devices which have network based user interface (UI). These devices include network routers, managed switches, automation equipment, wireless sensor network devices, as well as military surveillance and communication devices. We will show that it is possible to write software for such devices following standard OS like architecture and create real-time software components which can be shared between complex devices and resource limited devices.

The objective of our proposed operating system is reducing cost by using simple inexpensive microcontrollers, reducing the startup time by using a simple OS with a much faster startup time compared to existing ones, and finally reducing the power consumption of the system since both the application and the OS can fit in the memory of small microcontrollers without having an off-chip memory.

The uniqueness of the Romantiki operating system is that it integrates a specialized TCP/IP stack which allows developing multitasking networking application code that is easy to maintain and extend. This code can easily be shared between traditional operating systems and Romantiki OS through the use of a common OS abstraction layer.

Networking programs written for Romantiki enjoy a significant performance increase over the corresponding programs written for operating systems with traditional networking abstractions. In this paper we compare the performance of a simple TCP based application running on Romantiki OS and FreeRTOS with lwIP TCP/IP stack. It will be shown that Romantiki based application provides 2.7x TCP performance increase compared to the same application running on FreeRTOS with lwIP

B. Related Work

Traditional embedded networking devices are not power efficient due to the use of large number of on-board components to provide simple networking functionality. This inefficiency is in conflict with the new trend to reduce the amount of power consumption in devices and allow battery operated applications. This trend produced a number of microcontrollers with memory, networking and other peripherals integrated into the microcontroller chip [2], [13]. The amount of memory in these microcontrollers is very limited and is not sufficient to run traditional networking operating systems.

An example of a small and efficient operating system is Nut/O/S [16] which provides a large set of services and is targeted to limited memory embedded devices. It is very practical for programmers who migrate from superloop based paradigm towards an operating system based approach and who want to avoid mistakes when using synchronization primitives and accessing shared memory. However, this operating system does not provide footprint reduction compared to preemptive operating systems since tasks in this system use separate stacks. Salvo [18] is another cooperative OS for small microcontrollers. Its main drawback is its inability to perform blocking calls within nested functions called from a thread.

A small real-time operating system was proposed in [11]. Their OS was designed for monitoring flight parameters and responding to any risk to the aircraft. Their main design criterion is fault tolerance. They designed and built the hardware for the system and the fault tolerance was introduced at both the hardware and software level. Although the OS is small it was not meant to support networking applications.

In [20], the author presents design challenges of the design of an operating system for wireless embedded systems powered by energy harvesters. Although their OS is designed to run on limited resources microcontrollers like our system, the main design criterion for their OS is energy efficiency, since it is using energy harvesting techniques for power instead of batteries.

Some embedded TCP/IP stacks are extremely small and provide an extensive TCP/IP protocol handling but they do not provide the BSD style networking API. Examples of such systems are uIP TCP/IP stack [4], Contiki OS [6] and microchip [17] TCP/IP stack. Those stacks are programmed in an event-driven style with very limited blocking socket capabilities.

Other TCP/IP stacks such as lwIP [5], uC/TCP-IP [22] and InterNiche TCP/IP stack [15] provide networking abstractions compatible with traditional BSD socket layer. These typically run under an RTOS and require a significantly larger amount of resources in RAM and Flash than event driven TCP/IP stacks.

Romantiki implements a TCP/IP stack targeted at server only embedded systems located on a local area network. This stack stands in the middle between BSD Socket compatible TCP/IP stacks and event driven TCP/IP stacks. It provides networking abstractions similar to traditional BSD socket API while fitting the footprint of an event–driven TCP/IP stack. The Romantiki TCP/IP stack offers a "socket-like" networking API which makes it possible to write single threaded blocking socket applications and multisession non-blocking socket applications. This functionality is typical of an embedded system which handles multi-session servers using non-blocking sockets to avoid multiple tasks handling sessions with large stack allocations per task. The socket-like networking API employed in Romantiki makes it possible to share the same application codebase between large projects running on complex preemptive operating systems and projects running on resource limited embedded devices. This is especially useful for providing limited functionality networking applications on resource limited devices while providing the same applications with full functionality running on traditional operating system sharing the same application code.

III. Romantiki OS

Romantiki OS [9] is a cooperative multitasking operating system with task priorities where all tasks share a common stack. The tasks are run-to-completion programs written using local-continuation programming style to provide blocking I/O functionality [10]

Romantiki achieves real-time cooperation by using yielding. Special yield statements are placed in the code that allows the program to yield if there is a higher priority program waiting, and return to the appropriate place in the function after the higher priority task is completed [10].

Romantiki OS uses traditional intertask communication primitives such as Event Objects, Timers, and Semaphores which are implemented using blocking system calls. Romantiki allows making blocking I/O calls within nested functions. The use of traditional intertask communication abstractions allows creating a common OS abstraction API which enables the sharing of application code between other embedded operating systems such as FreeRTOS, ThreadX and Romantiki OS. A block diagram of the Romantiki operating system is shown in Figure 1.

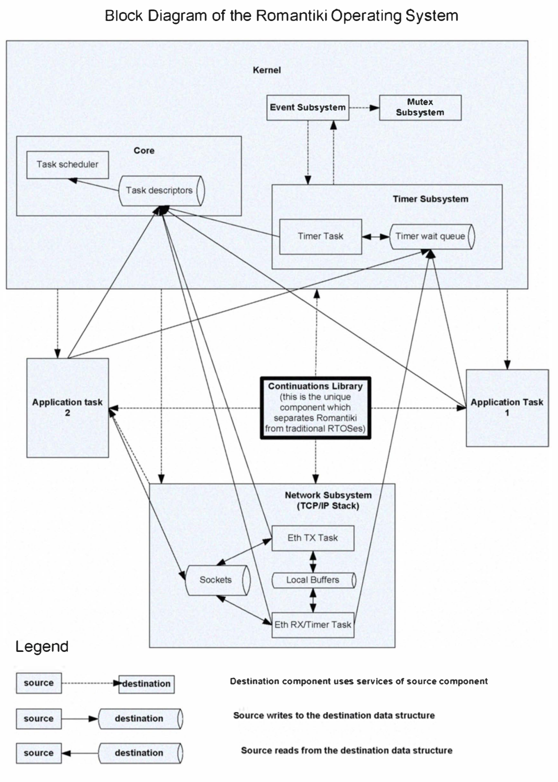
Block Diagram of the Romantiki Operating System
Kernel
Event Subsystem
Mutex Subsystem
Core
Task scheduler
Task descriptors
Timer Subsystem
Timer Task
Timer wait queue
Application task 2
Continuations Library (this is the unique component which separates Romantiki from traditional RTOSes)
Application Task 1
Network Subsystem (TCP/IP Stack)
Eth TX Task
Sockets
Local Buffers
Eth RX/Timer Task
Legend
source
destination
Destination component uses services of source component
Source writes to the destination data structure
Source reads from the destination data structure

Fig 1. Block diagram of Romantiki OS

Romantiki OS features an integrated TCP/IP stack which enabled the creation of TCP/IP server applications in memory limited devices.

IV. TCP/IP Stack in Romantiki

A. *Overview*

The TCP/IP Stack in Romantiki is optimized for “server-only” embedded applications. The device running the Romantiki cannot initiate TCP or UDP connections but it can accept TCP connections and respond to TCP/UDP messages. The TCP/IP stack is written with the intention to keep the code as simple as possible while closely following the “socket API” interface to be able to port existing applications to Romantiki OS.

The following sections describe various algorithms implemented in the TCP/IP stack in Romantiki.

B. *“Socket-Like” API*

The socket-like API of Romantiki allows us to handle both TCP and UDP sockets. For TCP it implements socket creation, sending, receiving, accepting and closing a socket. It also implements socket select for non-blocking sockets. For UDP it implements creating a socket, sending, and receiving operations.

Each socket can be configured in a blocking or non-blocking mode of operation. Blocking sockets are used to implement single threaded server application while non-blocking sockets are used to create multisession servers.

The socket-like API closely follows the traditional BSD socket API and that makes it possible to create a common abstraction layer which allows us to compile identical source code implementing variety of networking servers for Romantiki and traditional operating systems using BSD socket compatible TCP/IP stack.

Many socket operations are implemented via Zero-Copy fashion common to many TCP/IP stacks such as Trek TCP/IP Stack [21], embOS/IP [7] and uC/OS-II TCP/IP [12]. This contributes to the fast performance of the Romantiki TCP/IP stack

C. *ARP Handling and IP Routing*

Typically a TCP/IP stack is structured in a layered model. TCP/IP defines 4 protocol layers: Application layer, Transport layer, Internet layer and a Link layer. Each layer performs certain processing of incoming and outgoing frames. At the same time each layer is responsible for certain protocols specific to that layer. For example: ARP protocol is handled entirely within the Link Layer while the TCP protocol handling is part of the Transport layer.

The footprint of the Romantiki TCP/IP stack is very small. This is accomplished by blurring the distinction between the layers of the TCP/IP protocols and making certain assumptions about the device functionality and network infrastructure. The following assumptions in Romantiki affect the ARP handling and IP routing:

1. The Link layer is Ethernet based.
2. The device never originates TCP/UDP/ICMP sessions but always responds to external requests.

3. The device has a single Ethernet port. It can not act as a router between multiple IP networks.
4. Paths are symmetric – the outgoing frame should be transmitted to the same gateway which forwarded the incoming frame.
5. The MAC Address of the connection originator is part of the connection structures in Internet and Transport layers. This construct eliminates a standalone ARP cache and simplifies routing decisions for ICMP, UDP and TCP protocols.

Some of the parameters typically required for proper TCP/IP operation are not necessary in devices built with these assumptions. Based on the above assumptions, the Netmask, ARP Cache, Default Gateway and Routing table are not part of the Romantiki TCP/IP stack because each TCP/UDP application knows the MAC address of the gateway the response needs to be forwarded to. This greatly simplifies the structure of the TCP/IP stack as well as makes the memory footprint small and deterministic. This approach also reduces the number of user configurable parameters to one (IP Address of the device) which allows for easier field maintenance.

D. TCP Delayed ACK

The TCP delayed acknowledgement algorithm in Romantiki is different than the one used in a traditional stack. The acknowledgement for the incoming data is delayed until one of two events occurs:

1- The receiving application task blocks waiting for more data. This means that the socket's RX buffer is empty and more data needs to be received.

2- The application is sending the response. In this case the acknowledgment is piggybacked to the response frame.

The algorithm is targeted towards Industrial automation applications where the complete application frame typically fits a single MSS segment. Therefore, the ACK is typically piggybacked to the response frame.

On one hand, this scheme may not be as bandwidth efficient as the traditional delayed acknowledgement algorithm for protocols that have multiple segments in transit. On the other hand, this algorithm is very efficient for the majority of control network traffic in industrial automation networks and many LAN applications and it doesn't pose the performance problem of the traditional delayed ACK algorithm [3].

E. Additional Features of the TCP/IP Stack

The TCP socket can be configured to "defy" the traditional delayed acknowledgement algorithm by splitting each outgoing frame into two. This causes the remote application to send an ACK frame right away. This is similar to uip_split feature of the uIP stack [23].

The TCP IP stack also provides an internal functionality to respond to "ICMP ECHO" frames providing the "PING" functionality as well as responding to ARP frames.

V. Footprint and Performance Comparison

The Romantiki OS provides a coding style similar to traditional embedded operating systems. It features a TCP/IP stack that provides a "socket-like" interface which is easier to program than uIP based code. The performance and footprint of this stack is much better than BSD compatible TCP/IP stacks as it will be described in the following comparison.

In order to provide a realistic comparison of performance of different operating systems, it is necessary to compare their performance on a similar application running on an identical hardware. The TCP echo server application, shown below, was used as a basis for comparison since it is a very simple application and yet it exercises a large number of operations in the operating system such as multitasking, intertask communication and networking subsystem. It provides a benchmark on the performance of networking applications running on top of an operating system.

```
unsigned char echo_data[256];
// define listening socket
DEF_LST_SOCKET(echo_listen_sock,1);
// define connected socket
DEF_STRM_SOCKET(echo_real_sock);

BFD EchoServerTask(void* prm)
{
  AUTO unsigned int txed_len, rxed_len;
  SOCK_BUF(sock_buf,256);
  AUTO int res;


  BF_BEGIN
  // initialize listening socket
  LST_SOCKET(echo_listen_sock,
        SOCK_MODE(SOCK_BLOCKING));
  // initialize connected socket
  STRM_SOCKET(echo_real_sock,
        SOCK_MODE(SOCK_BLOCKING),
        (unsigned char*)sock_buf,
        sizeof(sock_buf));
  socket_listen(&echo_listen_sock,5200,0);
  for (;;)
  {
    socket_accept(&echo_listen_sock,
            &echo_real_sock,&res);
    if (res==EOK)
    {
      while (1)
      {
        socket_recv(&echo_real_sock,
              (unsigned char*)echo_data,
              256,&rxed_len,&res,NULL);
        socket_send(&echo_real_sock,
              (unsigned char*)echo_data,rxed_len,
              &txed_len,&res,NULL);
        if ((rxed_len != txed_len) || (rxed_len == 0))
        {
          break;
        }
      }
    }
  }
  BF_END
}
```

The following library makes it possible to run the above code on a TCP/IP stack based on traditional BSD sockets:

```
#define AUTO
#define SOCK_BUF(name,x)
#define BFD void
#define BF_BEGIN
#define BF_END
#define TRUE 1
#define FALSE 0
#define EOK TRUE

#define DEF_LST_SOCKET(x,y) SOCKET x
#define DEF_STRM_SOCKET(x) SOCKET x
enum
{
   SOCK_BLOCKING
};
#define SOCK_MODE(x) x
#define LST_SOCKET(x,y) init_sock_socket(&x,y)
void init_sock_socket(SOCKET* sock,int y)
{
   *sock = socket(AF_INET, SOCK_STREAM, 0);
}
#define STRM_SOCKET(x,y,z,t) init_sock_socket(&x,y)
void socket_listen(SOCKET* x,int port,int tmp)
{
   struct sockaddr_in serv_addr;
   memset((char *) &serv_addr,0, sizeof(serv_addr));
   serv_addr.sin_family = AF_INET;
   serv_addr.sin_addr.s_addr = INADDR_ANY;
   serv_addr.sin_port = htons(port);
   bind(*x, (struct sockaddr *) &serv_addr,
       sizeof(serv_addr));
   listen(*x,5);
}

void socket_accept(SOCKET* listen, SOCKET* client, int* res)
{
   int clilen;
   struct sockaddr_in cli_addr;
   clilen = sizeof(cli_addr);
   *client = accept(*listen,
              (struct sockaddr *) &cli_addr,
              &clilen);
   *res=EOK;
}
#define socket_send(c,b,l,t,d1,d2) *t=send(*c,(char*)b,l,0)
#define socket_recv(c,b,l,r,d1,d2) *r=recv(*c,(char*)b,l,0)
```

The code in the previous section follows the structure of a traditional socket server. Therefore it is possible to adapt a common abstraction layer which allows using the above code in an operating system supporting standard socket abstractions

The AT91SAM7X-EK [1] evaluation board was used to evaluate the performance and the footprint of the Romantiki operating system. The Romantiki OS is compared against FreeRTOS[8] operating system with two different TCP/IP stacks: uIP [23] and lwIP [14]. The FreeRTOS operating system was chosen due to the ease of porting it to the AT91SAM7X-EK [1] evaluation board as well as the ability to limit the number of used features so that the executable image has a similar feature set as the executable image of Romantiki OS.

The performance of the different operating systems is measured using the setup described in Figure 2.

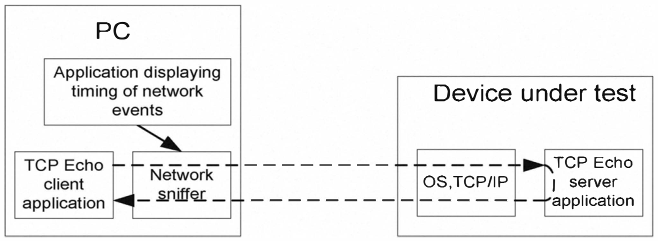
PC
Application displaying timing of network events
TCP Echo client application
Network sniffer
Device under test
OS,TCP/IP
TCP Echo server application

Fig 2. Experiment Setup

The absolute timing of network frames was captured using the network sniffer and results were averaged over 5 samples for each platform.

Table 1 provides the comparison between the performance of the echo server application on Romantiki OS and FreeRTOS (using both lwIP and uIP).

TABLE 1 COMPARISON BETWEEN ECHO SERVER PERFORMANCE ON ROMANTIKI OS AND FREERTOS

Operating System and TCP/IP stack	Combined size of all data segments in bytes	Size of the Code segment in bytes	Cycle time in microseconds
Romantiki OS based application	8064	14396	345
FreeRTOS and uIP based application	9320	13112	238
FreeRTOS and lwIP based application	16554	33176	934

As we can see from the footprint comparisons, the Romantiki based application has a similar footprint as the uIP application running on FreeRTOS operating system (both the code size and the RAM size). At the same time, the coding style of the Romantiki based socket applications is very similar to a lwIP based application while occupying a far smaller memory and code size footprint.

The Romantiki OS positions itself as a general purpose OS which enables code sharing between other operating systems running TCP/IP stacks similar to BSD stack. This functionality is similar to the lwIP stack running on top of a FreeRTOS operating system.

The differences between the application running on Romantiki OS and a similar application based on FreeRTOS with lwIP are the following:
The Flash footprint of the Romantiki application is 2.3 times smaller than the same application implemented in FreeRTOS with lwIP stack.

The RAM footprint of the Romantiki application is 2 times smaller than the same application implemented in FreeRTOS with lwIP stack.
The TCP performance of the Romantiki application is 2.7 times faster than the same application implemented in FreeRTOS with lwIP stack.

The performance of the uIP application is the best among the three platforms. This is due to the requirement that TCP/IP applications in uIP need to be structured as event handlers invoked by Ethernet processing task. This reduces the amount of intertask communications to process TCP/IP frames at the expense of abandoning traditional "socket" based program structure. The event handling approach for TCP/IP applications is fast and simple for request/response applications where the response data can be directly computed from the request. However, the code becomes quite complex when the data needs to be read or written using blocking I/O. The performance of such an application could suffer if the blocking I/O is implemented in a polled fashion. An application such as a web server where the web pages are stored on SD card could be slow and complex when implemented using FreeRTOS with uIP stack. Moreover, the requirement of using event driven coding style makes it hard to adapt existing TCP/IP code based on blocking sockets to uIP stack.

The memory footprint of the Romantiki OS based application is quite similar to that of uIP based application, however, the performance of the uIP based application is 1.4 times better. The performance lead of uIP is due to the event handling structure of the uIP stack.

As it was shown in this section, Romantiki OS provides a big boost in performance while also providing a far smaller footprint compared to FreeRTOS with traditional TCP/IP stack lwIP. Even though the Romantiki TCP/IP stack does not use the standard socket interface, its API is very close to traditional sockets and makes it possible to share the same codebase between projects running on large microprocessors and resource limited microcontrollers.

VI. FUTURE WORK AND CONCLUSION

Romantiki was designed as a proof of concept of a multithreaded networking operating system which fits the RAM and Flash footprint of a superloop project. It is very compact and easily portable to different architectures.

This operating system allows constructing complex devices using very small and inexpensive microcontrollers.

There are many different applications of these devices in the areas of industrial automation, home automation, telecommunications and military applications.

The future work will concentrate on adding originator support in the Romantiki TCP/IP stack.

REFERENCES

[1] AT91SAM7X-EK Evaluation board http://www.atmel.com/dyn/products/tools_card.asp?tool_id=3759 [Accessed May 2010]

[2] AT91SAM7X ARM7 Microcontroller [Online] available http://www.atmel.com/dyn/resources/prod/documents/6120s.pdf [Accessed May 2010]

[3] S. Cheshire "TCP Performance problems caused by interaction between Nagle's Algorithm and Delayed ACK" http://www.stuartcheshire.org/papers/nagleDelayed Ack [accessed May. 2010]

[4] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali" Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems" Proceedings of the 4th International Conference on Embedded Networked Sensor Systems" Boulder, CO 2006.

[5] A. Dunkels "Full TCP/IP for 8-bit architecture" Proceedings of the 1st international conference on Mobile systems, applications and services. pp 85-98San Francisco, CA 2003.

[6] A. Dunkels, B. Grönvall, and T. Voigt "Contiki – "A Lightweight and Flexible Operating System for Tiny Networked Sensors". Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks pp 455-462 2004

[7] Emb/OS Real Time Operating System http://www.segger.com/embosip_general.html

[8] FreeRTOS Operating System http://www.freertos.org/ [Accessed May 2010]

[9] R. Glistvain, M. Aboelaze "Romantiki – A Single Stack Operating System for Resource Limited Embedded Devices". Proceeding of the 7th International Conference on Informatics and System. Cairo, Egypt March 2010.

[10] R. Glistvain "Romantiki O/S – networking operating system for limited memory embedded devices". A M.Sc. Thesis, Dept. of Computer Science and Engineering, York University. Toronto, Canada April 2010.

[11] T. Kaegi-Trachsel, and J. Gutknecht "*Minos: The design and implementation of an embedded real-time operating system with a perspective on fault tolerance*". Proceedings of the International multiconference on Computer Science and Information technology pp 649-656 Oct. 2008.

[12] J. Labrosse "MicroC/OS-II: The Real-Time Kernel:" 2/ED CMP Books 2002.

[13] LM3S8962 ARM-Cortex M3 Microcontroller [Online] available http://www.luminarymicro.com/products/LM3S8962.html [Accessed May 2010]

[14] lwIP TCP/IP stack http://en.wikipedia.org/wiki/LwIP [Accessed May 2010]

[15] NicheStack IPv4 http://www.iniche.com/nichestack.php [Accessed May 2010]

[16] Nut/O/S Cooperative RTOS Available http://www.ethernut.de/en/firmware/nutos.html [Accessed May 2010]

[17] N. Rajbhart. "Microchip TCP/IP stack" – AN833 Microchip Technology Inc. August 2008.

[18] RTOS http://www.pumpkininc.com/ [Accessed May 2010]

[19] STM32F105R8 Microcontroller with embedded Ethernet core http://www.st.com/mcu/devicedocs-STM32F105R8-110.html [Accessed May 2010]

[20] A. Strba "Operating System design challenges for wireless embedded systems powered by energy harvesters" Proc. Of the International Symposium on Applied machine Intelligence and Informatics. Pp 35-4- Jan. 2009.

[21] TREK TCP/IP stack http://www.treck.com/pdf/TCP.pdf [Accessed May 2010]

[22] uC/TCP-IP stack for uC/OS-II http://www.arm.com/community/display_product/rw/ProductId/2442/ [Accessed May 2010]

[23] UIP-Split module for uIP TCP/IP stack http://www.sics.se/~adam/uip/uip-1.0-refman/a00201.html [Accessed may 2010]