Take A Hike

A Trek Through the Contiki Operating System

Hossam Ashtawy, Troy Brown, Xiaojun Wang, Yuan Zhang
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824
{ashtawy, tbrown36, wangxi44, zhangy72}@msu.edu

Abstract—**This paper examines the Contiki Operating System and provides in-depth coverage of the core concepts required to understand how it behaves. This understanding can be used to increase an application developer's knowledge of the underlying system. The paper can also be used to help a kernel developer port the operating system to another platform or to develop additional device drivers for an existing platform. No matter how the end user expects to utilize Contiki, the tutorial provides a base level of understanding, which can be leveraged to enhance the development experience.**

1 Overview

The Contiki Operating System falls within the category of Event-Driven systems. This means that the applications executing on the OS are triggered by events. An application consists of a process, which is also considered an event handler. Any time an event is destined for the process, the process is scheduled and the event is delivered to the process in the form of activating its event handler. The event handler parameters allow for the event, along with any data, to be passed to the handler so that it can be examined and acted upon.

Unlike traditional personal computer operating systems, the kernel of Contiki is not protected from applications. This means that an application access and/or corrupt global data, even data belonging to the underlying operating system. There are a number of reasons for this type of configuration. The most prominent reason is the fact that Contiki is suitable for deeply embedded systems, where memory management hardware, such as an MMU, is not available on the platform. This is common for micro-controllers as well as low-end processors.

A further reason for the kernel and applications sharing the same memory space is due to memory and processing constraints. Allowing for a more tightly coupled operating system and application allows for more efficient use of memory. It is not unusual for a typical embedded system to have less than 1K of RAM available to the system. In order for an operating system to be useful on these types of systems, it must not consume all of the resources of the platform. Therefore, one of the driving forces behind the Contiki operating system is to minimize the amount of resources consumed.

In addition to an application and kernel sharing the same memory space, and therefore access to each other's resources, an application also exists in the same memory space as another application. Therefore, applications can also corrupt each other in addition to the kernel.

Even though applications and the kernel share a memory space and have access to each other's global data, this is not always considered a bad design. As has already been indicated, resources are scarce on these systems; however RAM is not the only resource that is scarce. Normally, the processors (or micro-controllers) that are in these systems have just enough processing power to perform the task at hand. The processors are normally selected based on a price/performance tradeoff. The more that can be done with less processing power, the less the end product will ultimately cost. The fact that the applications and kernel share memory space makes it very efficient for them to share information through global variables. Although this causes the operating system and applications to be tightly integrated, it allows for more efficient usage of the available memory. This is the approach taken by the designers of Contiki; increase the coupling to allow for more efficient usage of system resources. Due to this efficient usage of resources, typical configurations of the Contiki operating system can fit within 2K of RAM and 40K of ROM.

2 History

The origins of Contiki lay within the day dreaming of its creator, Adam Dunkels[1] who works for the Swedish Institute of Computer Science[2]. During his morning commute on the train in 2002, he had the idea of an embedded web browser. Since he had previously developed an embedded TCP/IP networking stack and web server, the thought of a web browser was not that far fetched. Using a Commodore 64 as a means to display the browsed web using a primitive text based graphical interface (i.e., Contiki Toolkit) and an Ethernet card, also developed by Dunkels[3], the web browser was born.

This successful development led Dunkels to further exploit the seemingly vast amounts of remaining resources in the machine by developing additional applications. Ultimately,

[1] http://www.sics.se/~adam/
[2] http://www.sics.se/
[3] http://dunkels.com/adam/tfe/

this led to the desire to run multiple applications on the CPU at the same time, thus Contiki was born as an attempt to fulfill this desire, allowing for multiple applications to execute concurrently within the event-based computing paradigm.

3 APPLICATIONS

Ever since its original release Contiki has been supported on multiple platforms ranging from sensor network nodes to out-dated home computers from the 1980's (such as the Commodore 64, Atari, Apple II, etc.) to resource constrained embedded systems.

In addition to multiple platforms, Contiki has also been deployed within a number of different industries for both commercial and non-commercial usage. Applications vary widely and include automobiles, airplanes, satellites, freighter ships, oil drilling equipment, container tracking and television production equipment, to name a few.

The Contiki development team also consists of a broad range of international companies who are committed to advancing the Contiki operating system and the platforms on which it runs. Cisco and Atmel are just two of the contributors to the continued development of Contiki, reflecting the types of areas in which Contiki has been designed to operate. Cisco is a major networking company and Contiki has one of the smallest fully functional network stacks in existence, including an IPv6 network stack. Atmel is a global corporation with a significant foothold in the micro-controller market, a key segment for which Contiki was designed to operate. One of the main applications for Contiki is for use on nodes within a sensor network. The networking aspect of these systems as well as the small computing platforms tie directly into the product domains of these two companies where little evidence is needed in order to see why they have decided to become involved in the development of the operating system.

4 EVENT DEFINITION

Since events constitute the heart of the Contiki Operating System they play a very large role in the proper functioning and interaction between the kernel, applications and drivers within the system. Each application essentially is one process with a main entry point corresponding to an event handler. All interactions are based on either shared global data or event communication.

4.1 Event Types

Events can be classified in two distinct ways. The first way is to determine the destination of the event. Events can have either a single process recipient (i.e., single receiver event) or all processes (i.e., broadcast event). The actual receiver class associated with an event is determined during run-time when the sender, using one of Contiki's Event APIs, injects the event into the system. An event is initiated either by the kernel or by a process. Device drivers executing on an interrupt context (i.e., caused by an interrupt) use a different method of communication, poll requests, discussed later.

Normally, the kernel will only send events to a process when it is created (INIT event), stopped (EXIT and EXITED events) or polled (POLL event). Processes on the other hand, can send events whenever they desire, depending upon the implemented functionality corresponding to the process. It should be noted that the kernel itself cannot receive an event. A process event-handler is the only place where events can be received.

The second method by which an event is classified is determined based on whether the event is a synchronous event or an asynchronous event, thus distinguishing the kind of event being sent. Asynchronous events are queued in the kernel's event queue and delivered when the OS schedules that event to be delivered. Since the event queue is a FIFO, events will be scheduled based on the order in which they were added to the queue. There are Event APIs provided by Contiki to send an asynchronous event. These types of events can be considered a deferred procedure calls since the event is not immediately delivered. When it is delivered, the receiver process (or processes) will have their event handler routine (i.e., the main entry point for the process) invoked with an argument specifying the event being sent.

The kernel, on the other hand, handles a synchronous event immediately, causing the receiving process (or processes) to be immediately scheduled and their event handler to be invoked. After the synchronous event is delivered, the original process will resume its execution. This type of event is analogous to an inter-process procedure call.

Contiki employs run-to-completion semantics, which is common in event-based operating systems. This means that no preemption is employed by the operating system. Instead, the processes are expected to quickly process received events, register to receive new events and finally, return from the event handler so that another process can be scheduled. This requires that applications cooperate with each other or the system may come to a halt where one process is consuming all of the processor's time, preventing rescheduling of other processes.

This cooperative environment is not safe from a security standpoint, but since the system is very small and tightly coupled, it is not expected that internal security will be a large concern. Typically, applications of this size are small enough that one or two people will implement the entire system and thus will have full control over how the processes behave and interact with each other.

4.2 Event Structure

For asynchronous events, the kernel must keep enough information so that it can later schedule a process to receive the event. Therefore, for each event in the event queue, three pieces of information are maintained, as specified within the kernel's source[4] event_data structure, and illustrated in TABLE 1.

The "ev" field contains the numerical value associated with an event. Each event type, such as INIT, EXIT, EXITED, etc. has a unique value that is used to differentiate between event types. This event number will be delivered to the process

[4] See /contiki-2.4/core/sys/process.c in the kernel source code available at http://www.sics.se/contiki/download.html.

specified in the "p" field when the event is scheduled, by invoking the process event handler passing the event number as a parameter. In addition, data can be associated with the event. The "data" field contains a pointer to the data that is to be delivered to the receiving process. Note that since the kernel and all of the applications share the same memory space, they can easily share data with each other simply by passing a pointer from one place to another. This is very efficient, as no data has to be copied from one process address space to another.

The only distinction between a single receiver and a broadcast event is what is specified in the "p" field. For a single receiver, this will be a pointer to the process structure of the receiving process. For a broadcast event, this will contain the special constant: PROCESS_BROADCAST (which is defined as NULL).

TABLE 1. CONTIKI EVENT QUEUE STRUCTURE

Field	Description
ev	Event Number
data	Pointer to data associated with event.
p	Pointer to process structure of receiver.

5 PROCESS DEFINITION

The kernel process module (i.e., found in process.c) is the heart of the OS. It contains a number of APIs for manipulating processes, events and scheduling. There are two main data objects that exist here, the Process List and the FIFO Asynchronous Event Queue.

The Process List is implemented as a singly linked list of process structures. This list only contains processes structures for processes that have been started in the system. These could either be processes that were started when the OS bootstrapped or were dynamically started during the run-time of the system. The length of the Process List can grow and shrink at run-time as new processes are started and/or older processes are stopped.

The FIFO Asynchronous Event Queue is implemented as a ring buffer whose length is fixed at compile time, through a configurable constant. The asynchronous events in this queue can be either of the single receiver or broadcast type. Note that synchronous events are never put into this queue as they are dispatched immediately. The queue holds events that have not yet been dispatched. They will be scheduled at a later time when the scheduler is invoked.

5.1 Process Structure

For all active processes in the system, Contiki must keep track of them so that they can be scheduled when events arrive, restore context when they are blocked and know what state they are in at any time. For each process in the process list, six pieces of information are kept, as specified within the kernel's process structure, and illustrated in TABLE 2.

The "next" field contains a pointer to the next process structure in the process list, or NULL if it is the last structure. The "name" field contains a pointer to the name associated with the process. The "thread" field contains a pointer to the event-handler that is associated with the process. Each process has an event-handler associated with it, as events are the main way that processes communicate with each other. The "pt" field contains the protothread state. Protothreads are further discussed in section 5.2. The "state" field contains the current state of the process. This is dependent upon whether the process has been started, stopped or is currently scheduled and is further detailed in section 6. Finally, the "needspoll" flag is used to signal a process that needs to check for data. This is usually associated with device drivers and is an indirect way to send an event to a process. This will also be discussed in section 6.

TABLE 2. CONTIKI PROCESS STRUCTURE

Field	Description
next	Pointer to next process in process list.
name	Name of the process.
thread	Pointer to protothread event handler.
pt	Protothread corresponding to process.
state	Process state (None, Running, Called).
needspoll	Flag requesting that process check for data.

5.2 Protothreads

A protothread is a stackless thread. This makes it great for use in small footprint systems, as memory is not wasted on multiple stacks that are usually only partially used. This means that protothreads are also well suited for the types of applications that Contiki is targeting. The main benefit that protothreads provide is conditional blocking inside a process event-handler. Traditionally, event-handlers cannot block as they implement run-to-completion semantics. If an event-handler did not return, it would prevent the scheduling of other processes. However, protothreads allow blocking operations within an event handler.

Through the use of special protothread blocking operations, some state machines that are normally found within an event-based application can be eliminated. The types of state machines that can be eliminated through the use of protothreads are those that exist strictly to allow the application to work in the presence of a non-blocking event handler. This is illustrated quite well in [1] which provides a side-by-side comparison of a traditional event-based application and also one implemented using protothreads.

As can be seen in the side-by-side comparison, the actual size is significantly decreased through the use of protothreads. The framework associated with maintaining state through multiple invocations of the event handler has been removed. The protothread example shows that normal sequential blocking operations can be implemented in a much simpler manner. The result is less application code, clearer implementations, blocking operations and event infinite loops.

5.3 Protothread Implementation

A protothread consists of two main elements, an event-handler and a local continuation. The event handler is the one associated with the process. A location continuation is used to save and restore the context when a blocking protothread API is invoked. It is a snapshot of the current state of the process. The difference between a local continuation and a more general continuation is that with a local continuation, the call history and values of local variables are not preserved.

In Contiki, protothread APIs are implemented as macros. These macros, when expanded, actually expose the implementation of the protothread, especially the inner workings of the local continuation. Reference [1] provides an example of an application both with the protothread macros, as would be seen in a normal application, as well as the expanded version of those macros. This demonstrates that in one particular implementation of a local continuation, a switch statement in combination with the associated cases, actually save and restores the context of the thread. Once this expansion is examined, it becomes clear that the explicit state in a traditional state machine has now become an implicit state in the protothread. When the macro is expanded, the numbers associated with the case statements actually refer to the line number in the original source where the blocking protothread call existed, guaranteeing unique values. As can be seen from this expansion, blocking as implemented by a protothread is all an illusion. Hidden "return" statements combined with a switch statement is used to provide an illusion of blocking at the application level. However, under the hood these mechanisms are used to return from the event-handler and also restore the context upon reentry.

Local continuations can be implemented in many different ways, such as through the use of special hardware, special compiler support or through the use of a switch statement as detailed in [2]. The benefit of using a switch statement is that no special hardware or compiler is required. It can be implemented using an ANSI C compiler.

There are a number of limitations associated with local continuations. One of the major limitations is that local variable state is not preserved across a blocking protothread API. Looking at the implementation, it is clear why this is not guaranteed. The execution actually returns out of the event handler, so the value of a local variable on the stack is not saved. Another limitation is that since switch statements are utilized under the covers, the application cannot utilize switch statements within the same routine as the protothread APIs. If a switch statement were to be used in the same event handler routine, the case statements could clash causing unwanted behavior. Additionally, since the protothread blocking APIs expand into individual cases of the switch statement, an application cannot block in a different routine than where the main PT_BEGIN/PT_END switch statement resides. This is necessary in order to preserve the cases within the switch statement.

6 PROCESS CONTROL

Figure 1 identifies the major users of the process control APIs. The figure only identifies the most common APIs that are used for process and event handling as well as scheduling. The Operating System (i.e., Contiki) is used to start processes during the boot sequence. Processes related to I/O and/or kernel services as well as general application processes can be identified to the kernel as needed to be automatically started when the operating system is booted. These types of services could be such things as networking stacks and event timers. The actual services to start will be specific to the system being designed. If a service is not needed, it is usually not included in order to conserve resources. The kernel is also responsible for running processes as part of the scheduler.

Interrupt Handlers are typically associated with device drivers to service I/O requests of the supporting hardware. These normally result in the request to have a process poll. The actual polling activity will also be dependent on the type of device. One example maybe associated with processing data bytes from a serial port. Polling will be further discussed in section 6.2.

Processes can start and stop other processes (as well as stopping themselves) and also post events. Since events are a normal part of inter-process communication on Contiki, these APIs are used frequently in general applications.

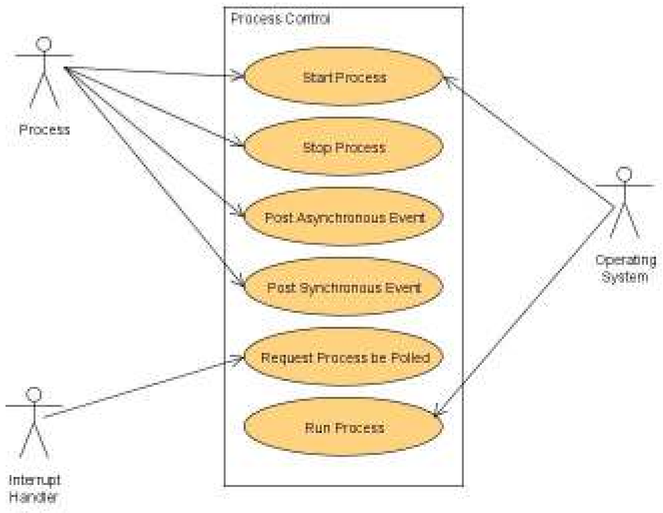

Figure 1. Common Process Controls

6.1 *Process State*

Figure 2 illustrates the different states that a process can be in at any point and time. This is maintained in the "state" field within the process structure. As soon as the process is started, it is added to the kernel's process list, transitions to the RUNNING state and the synchronous INIT event is sent to it, allowing any type of initialization that may be required, to occur. When a process is scheduled, it will transition to the CALLED state while the associated event-handler is executing. When a process is stopped, synchronous events will be sent to all processes. EXITED events are posted to all processes except the exiting process, to allow for reclamation of resources associated with the exiting process. An EXIT event will be posted to the exiting process to allow it to clean up after itself.

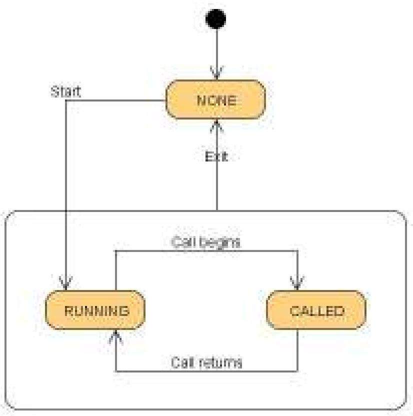

Figure 2. Process State Transitions

6.2 *Process Polling*

Polling is a way to signal a process to do something. It is normally associated with interrupt handlers. Contiki is designed without interrupt locking, so that if desired, it may be run atop a real-time executive. Due to this fact, it is possible that attempting to send an event from the interrupt context would cause race conditions with the foreground kernel scheduler. In order to sidestep these race conditions, a polling flag is set in the process structure of an associated I/O process (using the "Request Process be Polled" API) during interrupt context. When the foreground kernel scheduler runs, it will notice this flag is set and trigger a POLL event be posted to the associated process. That process will then perform whatever data operations might be associated with a "poll" operation. This means that there are normally two portions to an I/O driver; the portion associated with the interrupt that causes the polling flag to be set and the portion associated with the process that performs some kind of I/O specific operation on the data.

7 SCHEDULING

The Contiki OS employs a lightweight scheduler that dispatches processes when there are events posted against them or when the polling flag is set. Once a process is scheduled and the event handler starts executing, the kernel does not preempt it until completion, as described in [3]. When the system boots, as well as when a process blocks for an event, the scheduler selects a new process to run. From Contiki's scheduler perspective, the highest priority is given to processes that needs to poll. The scheduler knows when there exists one or more polled processes by checking a global system variable "poll_requested". See Figure 3 for a flow chart demonstrating this selection.

If polls are requested, then as the flowchart in Figure 4 shows, the scheduler walks through the system's process list and dispatches all the processes that have their polling flags set. When no polling processes remain, the scheduler services an event from the event queue and in turn schedules the corresponding process. The event can be destined to only one process or broadcast to all the running processes in the system. In the case of a single process event, the corresponding process is dispatched and the scheduler checks for polled processes again. If it is a broadcast event, then after each event has been processed, the scheduler dispatches all polled processes if there are any flagged.

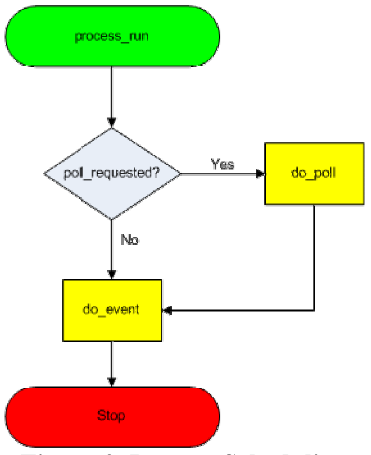

Figure 3. Process Scheduling

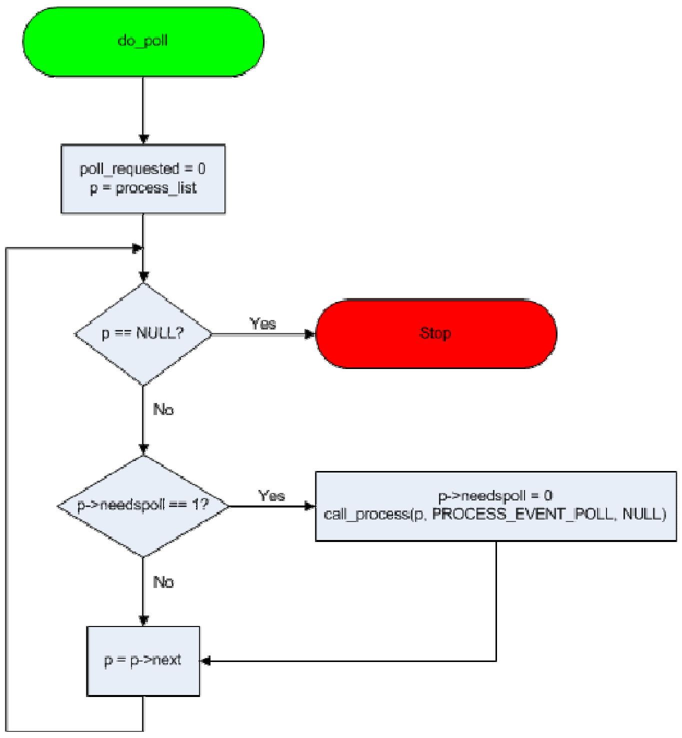

Figure 4. Scheduling Polling Processes

For preemptive multithreading, the Contiki OS provides a library that is built atop the kernel and provides an API to applications to run preemptively in a multithreaded mode. These routines are called explicitly by applications to run, yield and schedule threads.

8 EXAMPLE APPLICATION

A typical Contiki application consists of one or more processes defined in a C module. The skeleton of this module is as follows:

1. Process Declaration – every process in the application should be defined via the "PROCESS" macro before using it.
2. Autostart List – List of processes that needed to be automatically started when the module is booted. This list is managed by the "AUTOSTART_PROCESSES" macro.
3. Event Handler – Each process in the module has a body, which functions as an event handler. The body, which starts with the PROCESS_THREAD macro, contains:
    a. Initialization – Resources are allocated and variables initialized.
    b. Infinite Loop – Waiting and processing of events takes place.
    c. Resource Deallocation – Release resources at the end of the process thread (body).

Listing 1 demonstrates a simple Hello World application that contains a single process, waits for a timer event and when the time expires, the "Hello World" message is displayed.

```
#include "contiki.h"

PROCESS(example_process, "example process");

AUTOSTART_PROCESSES(&example_process);

PROCESS_THREAD(example_process, ev, data)
{
  static struct etimer timer;

  PROCESS_BEGIN();

  etimer_set(&timer, CLOCK_CONF_SECOND);

  while(1)
  {
    PROPROCESS_WAIT_EVENT(ev == PROCESS_EVENT_TIMER);
    printf("Hello, world\n");
  }

  PROCESS_END();
}
```

Listing 1. Hello World Example

9 Boot Sequence

The Contiki boot sequence varies from one platform to another, but the general process is similar. Hardware initialization, services, driver and kernel processes are started at boot-up. Then the system spins in an infinite loop where the schedule is invoked to start scheduling processes. Listing 2 provides an example of a minimal boot sequence.

```
//Kernel/Driver Processes to be started at boot time
PROCINIT(&etimer_process,&tcpip_process,
&serial_line_process);

int main(void) // Kernel Main entry point
{
  process_init();//initialize Kernel process module
  procinit_init();//Start Kernel/Driver Processes

  // Start initial Application Processes
  autostart_start(autostart_processes);

  while(1)
  {
    process_run(); // Schedule process(es) to run
  }
}
```

Listing 2. Minimal Boot Sequence

10 I/O Subsystem

The Contiki kernel does not provide a hardware abstraction layer (HAL), instead it lets applications and drivers communicate directly with the hardware as described in [3]. Even though Contiki supports a wide variety of sensors and devices, it lacks any well-defined and structured driver architecture, further detailed in [4]. The device drivers' interface varies from device to device and from one platform to another.

Usually when a sensor or device changes, a platform specific low-level code handles interrupt firing, and then a platform independent process, SENSORS which starts at boot time, is polled by the interrupt service routine to broadcast events (of SENSORS_EVENT type) to all the running processes in the system. The posted events carry information about which device or sensor has been changed. And finally, one or more processes receiving these asynchronous events may process the data generated by the source sensor/device.

11 Memory Management

12 File System

13 Networking

13.1 uIP and lwIP Stacks

13.1.1 Overview

TCP/IP protocols are widely used in the networking of an operating system. However, these protocols cannot be used directly in embedded systems, as they consume too many resources; they require a large code size to implement and consume a large amount of memory. In [5], Adam Dunkels gives two possible solutions: uIP (micro IP) or lwIP (lightweight IP).

Though uIP and lightweight IP are both simplified stacks implemented for embedded systems, their goals are slightly different. lwIP has many similar functions as TCP/IP, such as IP, ICMP, UDP and TCP, and it can be extended to support additional functionality, while uIP only supports the minimum set of features to keep the network running. Therefore, the resources they need are different. According to [6], uIP takes one fourth of the RAM and ROM, compared to lwIP's 30 kilobytes. Both of them can even run in limited 8-bit systems.

As can be seen, uIP and lwIP are simplified TCP/IP stacks for generic purpose. Some mechanisms, which are rarely used or less important, are removed. Next we will discuss the mechanism details of these two stacks.

13.1.2 Implementation

The uIP and lwIP stack are implemented as a main control loop. As Figure 5 shows, the main control loop does two things:

1. Checks for packets that arrived from the network.
2. Checks if a periodic timeout has occurred. In the first stage, if new packets arrive, the headers of the packets are parsed and sent to specific applications in the operating system. In the second stage, the network thread checks if there is a new packet that needs to be sent out to its peers.

The advantage of this mechanism is that it doesn't use interrupts and task schedulers. These mechanisms are very popular in other TCP/IP stacks; however to support the limited systems that Contiki targets, they are not used.

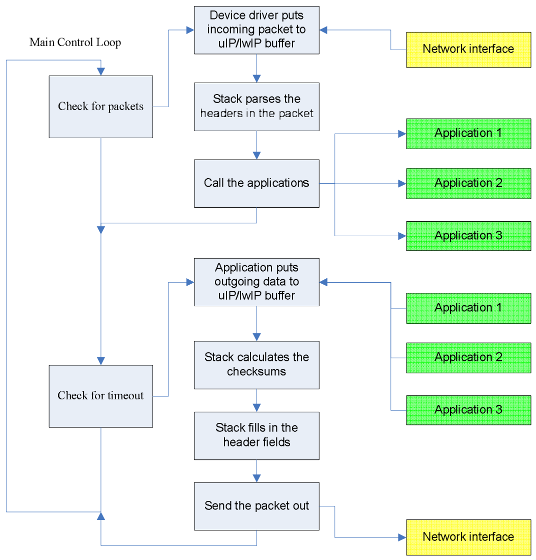

Figure 5. Main Control Loop

13.1.3 RFC Compliance

RFC 1122 is a document that is one of a pair that defines and discusses the requirements for a host system. The requirements can be divided into two categories; those that deal with the host-to-host communication and those that deal with communication between the application and the networking stack.

In the stack, all RFC requirements that affect host-to-host communication have been implemented. However, in order to reduce code size, in Contiki, certain mechanisms in the interface between the application and the stack have been removed.

13.1.4 Memory and Buffer Management

Due to the different design goals of uIP and lwIP, they have different memory management solutions. lwIP uses dynamic buffer allocation mechanisms. The dynamic memory is allocated from a global pool of available memory blocks.

The uIP stack does not use explicit dynamic memory allocation. Instead, it uses a single global buffer for holding packets and has a fixed table for holding connection state.

13.1.5 Retransmission, Flow Control and Congestion Control

lwIP maintains two output queues for retransmission: one holds segments that have not yet been sent, the other holds segments that have been sent but have not yet been acknowledged. However, uIP does not keep track of packet contents after the device driver has sent them.

The purpose of TCP's flow control mechanisms is to allow communication between hosts with wildly varying memory dimensions. In the implementation, the application cannot send more data than the receiving host can buffer.

The congestion control mechanisms limit the number of simultaneous TCP segments in the network. Since uIP only handles one in-flight TCP segment per connection, the amount of simultaneous segments cannot be further limited, thus the congestion control mechanisms are not needed. lwIP has the ability to have multiple in-flight segments and therefore implements all of TCP's congestion control mechanisms.

13.1.6 Performance

The performance overhead is calculated using two major factors:

1. Copying data from network to host memory.
2. Checksum calculation. A small, embedded device does not have the necessary processing power to have multiple protection domains and the power to run a multitasking operating system. Therefore there is no need to copy data between the stack and the application program.

There are other details we haven't talked about. For instance, the uIPv6, which is the first IPv6 stack for memory-constrained devices that passes all Phase-1 IPv6 ready certification tests, is illustrated in [8]. More TCP/IP features implemented by uIP and lwIP can be found in [5].

13.2 Rime Stack

The Rime stack is a layered communication stack that provides a hierarchal set of wireless network protocols. The goal of designing Rime is to simplify the implementation of communication protocols. The layered structure as detailed in [7], is shown in Figure 6. The Internet architecture, in the layers of Rime, is usually thin. Each layer adds its own header to outgoing messages. The Rime stack takes a little more resources but significantly reduces the complexity of sensor network protocol implementations.

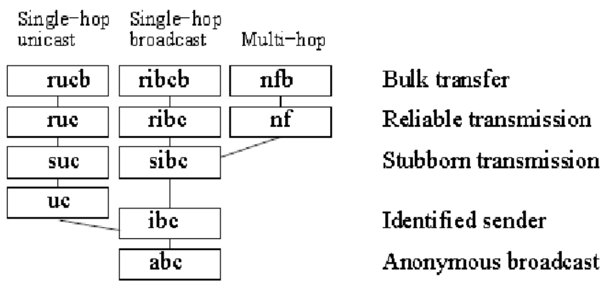

Figure 6. Rime Stack

13.3 Distributed TCP Caching

Traditional TCP is known to exhibit poor performance in wireless environments. Reference [9] focuses on the performance of TCP in the context of wireless sensor networks and introduces the Distributed TCP Caching (DTC) mechanism, which increases TCP performance.

DTC overcomes the problem by caching TCP segments inside the sensor networks and through the use of local retransmission of TCP segments. Further more, DTC shifts the burden of the load from vulnerable nodes close to the base station into the sensor network.

Other techniques that enable the use of TCP/IP for wireless sensor networks are also developed: spatial IP address assignment, shared context header compression and application overlay routing as described in [10]. An advantage of using TCP/IP is to directly communicate with an infrastructure consisting either of a wired IP network or of IP-based wireless technology.

14 SUMMARY

We have attempted to provide a broad overview of the Contiki operating system and as well as supply a detailed examination of some of the most important aspects of the system. However, there are many additional features that could not be detailed in this paper. Additional topics such as dynamic loading and unloading of software, multi-threading within a protothread as well as the Contiki Toolkit GUI are some additional areas that could be explored in a future trek. The additional topics are important aspects of the operating system; however they are not the main focus as they may only see limited use due to their niche qualities. The topics selected for this paper were expected to see more mainstream use and were selected based on that criterion.

Contiki has pioneered a number of innovations in operating system design. It was the first operating system to implement protothreads, IP-based sensor networks, dynamic loading as well as preemptive threads atop of event-based systems. As a further reflection upon the importance of these concepts, the statement that “Imitation is the sincerest form of flattery” can be applied to Contiki, as many of the operating systems that compete in the same arena, have incorporated these innovations into their own systems.

15 REFERENCES

[1] A. Dunkels, O. Schmidt and T. Voigt, “Using Protothreads for Sensor Node Programming,” Proceedings of the REALWSN’05 Workshop on Real-World Wireless Sensor Networks. Stockholm, Sweden, June 2005.

[2] A. Dunkels, O. Schmidt, T. Voigt and M. Ali, “Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems”, Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems. Boulder, Colorado, USA, November 2006.

[3] A. Dunkels, B. Gronvall and T. Voigt, “Contiki – A Lightweight and Flexible Operating System for Tiny Networked Sensors”, Swedish Institute of Computer Science, 2004.

[4] M. Oklobdzija, M. Nikolic, V. Kovacevi, “Serial Port Device for Contiki Operating System”, Telecommunications forum TELFOR 2009. Serbia, Belgrade, November 2009.

[5] A. Dunkels, “Full TCP/IP for 8-Bit Architectures”, ACM/Usenix MobiSys, 2003.

[6] A. Dunkels, “Rime – A Lightweight Layered Communication Stack for Sensor Networks”, EWSN, 2007.

[7] A. Dunkels and J. Vasseur, “IP for Smart Objects”, IPSO Alliance White Paper #1, September 2008.

[8] M. Durvy, J. Abeille, P. Wetterwald, C. O’Flynn, B. Leverett, E. Gnoske, M. Vidales, G. Mulligan, N. Tsiftes, N. Finne and A. Dunkels, “Making Sensor Networks IPv6 Ready”, ACM SenSys, 2008.

[9] A. Dunkels, T. Voigt, J. Alonso and H. Ritter, “Distributed TCP Caching for Wireless Sensor Networks”, MedHocNet, 2004.

[10] A. Dunkels, T. Voigt and J. Alonso, “Making TCP/IP Viable for Wireless Sensor Networks”, EWSN, 2004.