Operating System Support for Dynamic Code Loading in Sensor Networks[*]

Stefan Beyer
Instituto Tecnológico de Informática
Universidad Politécnica de Valencia
Spain
stefan@iti.upv.es

Robert Taylor and Ken Mayes
Department of Computer Science
University of Manchester
United Kingdom
{taylorr, ken}@cs.man.ac.uk

Abstract

Sensor network operating systems have to operate with limited hardware resources. Constraints on power consumption greatly reduce the resources available to such an operating system. Therefore, current systems restrict application flexibility and tend to impose certain programming paradigms, such as event-driven programming.

The Arena operating system is a special-purpose embedded operating system that has the potential to overcome these problems. The operating system is based on a strict separation of mechanism and policy. All operating system policies are placed in user-level libraries. These libraries can be loaded and replaced using a dynamic code loader, in order to configure the operating system dynamically. The dynamic code loader can also be used to load and replace arbitrary application components.

1 Introduction

Sensor networks are special-purpose systems. They are often designed to perform very specialised tasks. The software running on such systems could benefit greatly from adapting dynamically to changing environment conditions.

However, existing operating systems for sensor networks are very static and often enforce a particular programming model (for example event-driven) on the application programmer.

Arena is a small embedded operating system which has the potential to change the static nature of sensor network systems. The system does not impose any policies on the application programmer. Applications can be programmed purely event-driven or using a multi-threaded paradigm with configurable scheduling policies. Arena is configured by linking to different application libraries. This can be done statically or dynamically using a dynamic code loader. The dynamic code loader can also be used to load and unload arbitrary application components.

This paper introduces the Arena operating system and its dynamic configuration facility for embedded systems in its current state. A network protocol case study is introduced to show the flexibility of the system. Finally, plans to adapt Arena to network sensor nodes are outlined.

2 Related Work

2.1 Sensor Network Operating Systems

TinyOS [8] is an example of a statically configured operating system. TinyOS avoids the complex thread management and scheduling support that many larger operating systems provide, by enforcing a purely event-driven programming model.

This approach has its drawbacks, however; TinyOS is unable to make dynamic optimisations or dynamically change components at run-time. This means that new components cannot be downloaded in the field and then dynamically linked into the running program. Another drawback is that the event-driven programming model can make it difficult for application programmers to develop complex applications, as it might be difficult to express certain applications as state machines.

There are several techniques that have been developed to enable dynamic configuration in TinyOS. The first such technique facilitates the dynamic linking of components [12]. This approach does not allow system components, for example device drivers, and policies, such as thread scheduling, to be swapped at run-time.

Maté [14] is a Virtual Machine (VM) for TinyOS. Maté allows multiple scripts to run concurrently in a sandbox environment, which has a number of advantages. Firstly, a network can be dynamically changed over time in response to the environment. Secondly, multiple applications can simultaneously operate within a network. Maté introduces an

[*]This work has been partially supported by the Spanish MCYT grant TIC2003-09420- C02-01.

interpretation overhead which limits the type of computations that can be performed.

The SOS [7] operating system takes a different approach. SOS uses a a minimal kernel with a lot of the functionality implemented in user-controlled components. In SOS, the components under user control can be dynamically linked at run-time, although the facilities required to dynamically link components introduce a number of problems. The main problem is that the ability to dynamically configure an application only extends to user-controlled components. Kernel components cannot be replaced; for example, new operating system policies may be needed but the system does not allow this. Another problem is that message passing to other components in SOS introduces an software overhead.

Contiki [6] combines an event-driven kernel with an application-level preemptive multi-threading library. Although this gives the application programmer the choice of either programming model, the execution order of various processes is still dependent on events occurring in a certain order. A dynamic loader allows application-level code to be loaded or replaced at run-time. The loader allows devices drivers to be loaded, but operating system policies cannot be configured through the dynamic loader.

Applications are likely to change over time, so an operating system that allows itself to be reconfigured is an option that should be considered. Operating systems that can be dynamically configured, such as SOS, only allow application specific modules to be updated. Policy decisions and device drivers cannot be changed and are hidden inside the kernel. This prevents an application from using policies which best suit the task at hand. The structure of the Arena operating system described in Section 3 is such that it has been found that moving all operating system policy out of the kernel and under the control of the application does not reduce efficiency [1]. Therefore, the Arena operating system seems ideally suited for sensor networks because application components, as well as operating system policy, can be incrementally updated over time.

2.2 Dynamic Code Loading Systems

Devices running TinyOS can be re-programmed using Deluge [10]. Deluge allows new binary images to be installed over a network onto a device. Once loaded, a device can reboot to use this new image.

When new code is loaded in Maté, a soft-reboot occurs. This halts all threads and clears all script state. A Maté virtual machine then behaves as if it had just been reset.

Arena does not need to reboot or clear state to perform dynamic updates as its dynamic configuration facility is based on dynamic code loading. Various existing dynamic code loaders have been investigated to decide on a suitable mechanism.

Dyninst [5] is an example of a low-level approach to dynamic code loading. However, it lacks flexibility, as it cannot link in arbitrary code.

Probably the most suitable approaches for arbitrary dynamic code loading are based on dynamic linking and shared library systems. ELF systems [18] typically provide an API to the dynamic linker that can be used by the programmer to implicitly load executables. Apart from relying heavily on a UNIX environment, these systems use ELF shared objects, which are used for shared libraries. These shared libraries are loaded through the memory management subsystem on UNIX systems and rely heavily on a memory management unit (MMU), which small embedded systems, and network sensor nodes in particular, do not generally provide.

DLD [9] enhances a.out-based systems with dynamic loading and unloading of modules. DLD is a library package providing the ability to load relocatable object files, normally used as input files for static linkers, into a running application. The unlinking process relies on a garbage collector. DLD is the closest of all existing systems surveyed to the Arena loader However, DLD was designed for UNIX systems and certain aspects of it, in particular the use of a garbage collector, make it less useful for small systems with memory restrictions and real-time constraints.

Both SOS and Contiki provide code loaders similar to the Arena code loader, but the structure of both operating systems does not allow these loaders to be used to configure all aspects of operating system policies.

3 The Arena Architecture

3.1 Overview

Arena is an application-oriented operating system [15] [16] intended for both distributed and real-time applications [11] [2].

Arena introduces a separation between mechanism and policy. Low-level mechanisms are provided by a hardware-specific nano-kernel, the hardware object (HWO). The HWO provides mechanisms, such as the ability to save and restore the contents of the registers, but does not impose any policy on the context in which these registers are saved and restored. This architecture keeps the HWO free of operating system policy, but provides an architecture-independent hardware abstraction with opaque data types for low-level entities, such as register contexts.

All operating system policies are implemented in user-level libraries. These libraries are linked to the application, allowing the application programmer to choose the policies required, by linking to different version of these libraries.

The tight coupling between application and operating system policies leads to operating system managers

(OSMs), such as the process manager, residing at user-level. The OSMs interact with the HWO through the HWO provided downcall interface. Furthermore, OSMs provide an upcall interface to enable the HWO to cause OSM code to be executed.

On the occurrence of a hardware event, the HWO can make an upcall to some user-level resource manager. The upcall mechanism enables deferred processing of the event via an application-specific event handler thread.

3.2 The Dynamic Object Loader

OSMs are implemented as user-level resource managers in libraries in Arena. These can be linked statically or dynamically to the application to achieve re-configuration.

Dynamic linking is achieved by a dynamic object loader (DOL) which allows the dynamic loading and replacement of user-level modules which can be either regular code modules or OSMs. In contrast to other dynamic configuration systems, the system introduced here is optimised for systems with strong resource constraints, especially systems without an MMU. In order to reduce the run-time overhead, code is patched directly, in the same way as is done by a static linker.

In the Arena HWO nano-kernel, the Arena loader protocol (ALP), a very lightweight transfer protocol, resides at the top of the network protocol stack. ALP is similar to TFTP [17], but is implemented directly on IP. It provides the DOL with a simple send and receive interface, which allows the transfer of modules from a remote module server. The remote system contains an *application server*, which answers requests for whole applications and a *module server*, which is responsible for the transfer of modules. ALP packet types allow requests for either whole applications, whole modules, or individual symbol or string tables. This ALP interface is used by the DOL, which is linked into the application at user-level, to load the modules and link them into the application. Loadable modules are contained in ELF relocatable object files.

The application is loaded by the HWO using ALP. Once the application has been loaded in memory, it may require the loading of further modules. That is, subsequently, as required, modules can be loaded by the DOL. The DOL interacts with the remote module server to pull over the required modules.

The DOL can be used by the application either directly or through special reconfiguration layers. Providing reconfiguration layers for OSM loading hides details about resolving symbols and saving and restoring OSM state from the application programmer. As an example of this, a reconfiguration layer, that loads in different thread managers given only a scheduling policy, has been implemented [3] [4]. The network protocol loader introduced in section 4 is implemented as a reconfiguration layer that does not need any application interaction at all.

4 Network Protocol Loading

4.1 Overview

This section introduces a case study, which is aimed at demonstrating the use of the DOL for "on demand" loading of OSMs to save memory. The DOL has been used to implement a network protocol loader. The loader allows the dynamic loading of protocols onto the system. That is, the embedded software can listen to communication ports without the bulk of the associated protocol being loaded. The protocol is loaded when a message needs to be sent or interpreted.

Although the protocols chosen for the experiments might not be suitable for sensor networks, the techniques are applicable to all types of protocols and show how memory can be saved with minimal run-time overhead. For example, ethernet and TCP/IP were used during the experiments. Future sensor network specific implementations of Arena will use wireless communication and light-weight protocol stacks with shorter packet lengths.

The loading of protocols is supported at the transport layer and at the application layer.

Dynamic unloading of protocols can be performed explicitely, as well as implicitely.

4.2 Transport Layer Implementation

The Arena network protocol architecture [2] places transport layer protocols in OSMs at user-level. This means that transport protocols can now be accessed by the DOL. To demonstrate this the dynamic loading of TCP has been implemented.

The TCP code has been split into two parts. A statically linked part, containing the code for monitoring a TCP port, is linked statically to the application. This part provides the equivalent operations to the UNIX system calls *bind* and *listen* and also contains stubs for all the API calls provided by the dynamic part. A dynamically loaded part, containing the main TCP state machine, is located in a module at the remote module server. The TCP state machine is the part of code that performs the actual data transfer, flow control and error handling.

The Arena TCP interface functions are not explained in detail in this paper. The functions behave in a similar way to the BSD socket interface [13].

There are two cases to distinguish: The embedded system is used in *server mode*, when the embedded system waits for incoming connections. Here, the dynamic part is only loaded when a remote system tries to establish a connection to a port to which the system listens. *Binding* and *listening* to a port number does not invoke the loading of the dynamic part of TCP. The system is used in *client mode*, when the embedded system tries to establish a connection to some remote system. When a connection is being negotiated between the system and remote host, TCP packets are sent between them, so the dynamic part is needed. Therefore, a *connect* call causes TCP to be dynamically loaded directly. Since establishing a connection is almost always shortly followed by sending of TCP datagrams this implicit eager loading of the dynamic part is sensible.

4.3 Application Layer Implementation

To demonstrate the dynamic loading of application-layer network protocols a simple HTTP server has been implemented. If the embedded application wishes to become a web server it calls a HTTP initialisation stub. The DOL is used to load and link the protocol into the application. The HTTP server is self-contained. Therefore, the only symbol that has to be resolved is that of the initialisation routine itself. Once the location of the real initialisation routine is established, it is invoked. TCP might still not be loaded at this stage, but the loading of TCP is postponed until the first request for a web page is being received. Once the monitoring of the HTTP port is set up, the application can continue execution. Note that at this stage HTTP is loaded, but the dynamic part of TCP does not have to be loaded yet. On receipt of a datagram on the HTTP port, the `tcp_input` stub is called. If TCP has not been loaded yet, this causes the dynamic part of TCP to be loaded. A TCP connection is established and requests are received.

5 Evaluation of Network Protocol Loading

5.1 Experimental Setup

As there no implementation of Arena for any specific sensor network architecture, all experiments were run on an Atmel AT91M40800-based development board (32MHz), with 4MB of external RAM and Cirrus Logic CS8900A 10Mbps ethernet chip. The application server and the module server, from which modules were loaded, ran on an Intel PC running Linux.

5.2 Memory Footprint Analysis

Loading TCP dynamically adds 1.6 KBytes to the system. Considering that the dynamically-loaded part of TCP is 33 KBytes there is a considerable gain in memory, when TCP is not loaded.

Dynamically loading HTTP adds a negligible size to the memory footprint. The type of application-level protocol used on sensor networks are likely to be similar in size as the simple HTTP implementation used in these experiments.

5.3 Reconfiguration Performance

To establish the cost of loading TCP, the time it takes to establish a connection was also measured. In statically-loaded TCP the connection was established at an average time of 17ms. If TCP had to be loaded dynamically during the connecting process it took a total of 1.6s

However, this figure has to be seen in relation to the execution time of the application and to the frequency of this loading taking place. It therefore depends on the target application whether the delay is acceptable. Therefore, dynamically loading TCP does not make sense in all scenarios, but can be extremely useful for some applications. The dynamic loading of HTTP was not measured, but it is likely to be very fast since the code size of the module is very small.

5.4 Run-time Overhead

Data buffers of various sizes were sent from the PC to the embedded system and "bounced" back to the sending system. The times from the sending of the buffers to the receipt of the bounced buffers were measured using both a statically and a dynamically loaded version of TCP.

data size (bytes)	128	256	512
static (μs)	7030	9175	12655
dynamic (μs)	6860	8480	13735

data size (bytes)	1024	2048	4096
static (μs)	18693	32290	60045
dynamic (μs)	19049	31942	60113

Table 1. Network Data "Bounce" Times

Table 1 shows the average times it took to bounce the data buffers of various sizes. It can be seen that overall the times are very similar in the static and dynamic cases.

In some instances the times are slightly lower in the dynamic case, in other instances they are slightly higher. These variations are small and are probably due to variation in network load. Overall the results seem to suggest that there is no measurable overhead in using dynamically-loaded TCP.

6 Future work

Future work includes porting Arena onto our custom developed platform. This will allow us to study Arena in the context of sensor networks. This includes investigating the separation of operating system policies from the mechanisms that implement them in sensor networks which has received little attention thus far. Operating Systems such as TinyOS, SOS and Contiki have fixed policies which are hidden from an application. Arena exposes these policy decisions to an application and allows them to be swapped as an application demands. We intend to develop a number of resource managers that implement different policies e.g. scheduling, power management policies etc., and will evaluate which are more effective.

Other avenues of investigation include the investigation of running of multiple applications within the same node. The ability to run multiple applications is an open question in sensor network research. In order to arbitrate resources to all running applications efficient virtualisation policies must be used. The flexibility of Arena can be used to investigate how virtualisation policies effect application performance.

7 Conclusion

We have shown that embedded software can be configured dynamically by loading and linking code into the system at run-time. The Arena operating system provides a platform in which OSMs reside in user-level libraries. A system has been developed which allows the loading of relocatable pieces of code into the running system. To demonstrate the flexibility and performance of the code loading system a network protocol loader has been implemented.

Furthermore, plans to employ the system on sensor networks have been outlined. We believe that Arena can significantly improve the way sensor networks are programmed within the resource and battery power restrictions typical of such a network. The flexibility provided by dynamic code loading coupled with Arena's strict separation of mechanism of policy allows for a large number of different programming paradigms to be used.

References

[1] S. Beyer. *Dynamic Configuration of Embedded Operating Systems*. PhD thesis, University of Manchester, 2004.

[2] S. Beyer, K. Mayes, and B. Warboys. Application-compliant networking on embedded systems. In *Proceedings of the 5th IEEE International Workshop on Networked Appliances*, pages 53–58, October 2002.

[3] S. Beyer, K. Mayes, and B. Warboys. Dynamic configuration of embedded operating systems. In *WIP Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 23–26, December 2003.

[4] S. Beyer, K. Mayes, and B. Warboys. Dynamic loading in an application specific embedded operating system. In *International Workshop on Software Techniques on Embedded and Pervasive Sys tems*, May 2005.

[5] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[6] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proc. of the 29th Annual IEEE Int. Conference on Local Computer Networks*, pages 455–462, 2004.

[7] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *Proc. of the 3rd int. conference on Mobile systems, applications, and services*, pages 163–176, 2005.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. of the ninth international conference on Architectural support for programming languages and operating systems*, pages 93–104, 2000.

[9] W. W. Ho and R. A. Olsson. An approach to genuine dynamic linking. *Software - Practice and Experience*, 21(4):375–390, 1991.

[10] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. of the 2nd int. conference on Embedded networked sensor systems*, pages 81–94, 2004.

[11] S. Kingsbury, K. Mayes, and B. Warboys. Real-time arena: A user-level operating system for co-operating robots. In *Proceedings of The Interantional Conference on Prallel and Distributed Processing Techniques and Applications*, pages 1844–1850, 1998.

[12] S. Kogekar, S. Neema, B. Eames, X. Koutsoukos, A. Ledeczi, and M. Maroti. Constraint-guided dynamic reconfiguration in sensor networks. In *Proc. of the third international symposium on Information processing in sensor networks*, pages 379–387, 2004.

[13] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman. *The Design and Implemetation of the 4.3 BSD UNIX Operating System*. Addison-Wesley, Reading, MA, 1989.

[14] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *Proc. of the 10th int. conference on Architectural support for programming languages and operating systems*, pages 85–95, 2002.

[15] K. Mayes, S. Quick, J. Bridgland, and A.Nisbet. Language- and application-oriented resource management for parallel architectures. In *ACM SIGOPS European Workshop*, pages 172–177, 1994.

[16] R. Morrison, D. Balasubramaniam, M. Greenwood, G. Kirby, K. Mayes, D. Munro, and B. Warboys. A compliant persistent architecture. *Software - Practice & Experience, Special Issue on Persistent Object Systems*, 30(4):363–386, 2000.

[17] K. Sollinsl. *The TFTP Protocol (Revision 2) – RFC 1350*, July 1992.

[18] *Tools Interface Standards. Executable and Linkable Format (ELF), version 1.2*, 1995.