Lorien: A pure dynamic component-based Operating System for Wireless Sensor Networks[*]

Barry Porter
Computing Department
Lancaster University
Lancaster, England
barry.porter@comp.lancs.ac.uk

Geoff Coulson
Computing Department
Lancaster University
Lancaster, England
geoff@comp.lancs.ac.uk

ABSTRACT

In this paper we examine the current state of the art in WSN operating systems in terms of their general programming models and runtime reprogramming features. While all OSs admit the need – and provide some capability – for runtime reprogramming, we find that no existing OS employs a unified approach at the dynamic end of the spectrum. In this paper we present such a unified solution with a new OS design called *Lorien*. Lorien is a dynamic component-oriented OS aimed at permitting component-based changes to itself, including architectural change, throughout *every* aspect of the system at runtime including its kernel. A Lorien system typically permits 43KB – 90% – of program memory on the TelosB platform to be fully reprogrammable within a unified programming model, supporting rich future middleware and systems research in the WSN field.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design;
C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems

Keywords

Software components, dynamic, WSN, operating system

1. INTRODUCTION

The contribution of this paper is an exploration of possibility in the spectrum of operating system dynamics for WSN hardware. We identify 3 aspects of current OS design: the programming model employed in the general-case; the programming model with which runtime reprogramming is achieved; and the extent to which the OS is reprogrammable.

We find that while all OSs have some method of achieving runtime reprogramming – a requirement driven by the long-term remote deployment of many WSN systems – none of them do so using a unified programming model at the dynamic end of the spectrum.

Our aim in this work was thus to explore a unified-model approach supporting full system dynamics throughout the entire OS and wider system. To this end we employ a dynamic component model which provides a general *primitive of change* through the ability to load, interconnect and unload components. To the best of our knowledge this is the first example of such a dynamic unified-model OS in WSNs.

In this paper we (i) introduce the spectrum of current WSN OSs in terms of their programming models and runtime reprogramming capabilities, (ii) outline our aim to fulful an unfilled point in this spectrum at the extreme-dynamic / unified-model end, and (iii) present the key design points enabling this with a new OS called Lorien.

This work is part of a longer-term project and serves to promote & support rich future research in middleware and general software systems for WSNs.

2. RELATED WORK

In this section we survey existing WSN operating systems and develop a categorisation of them fistly with respect to their general programming models and secondly with respect to their runtime *re*programming models. We constrain our survey here purely to operating systems, excluding middleware and other higher-layer work. We are additionally here only interested in system dynamics; OS concerns such as concurrency models and network stacks are orthogonal and beyond our scope.

We include in our survey TinyOS [7], SOS [6], Contiki [5], Mantis [2] and LiteOS [3]. We contrast each with our target OS which we call *Lorien*.

We first examine the general programming models of these OSs, summarised in Figure 1. By "Image / Flat" in this table we mean any system developed / compiled monolithically. Here we see that three existing operating systems – TinyOS, LiteOS and SOS – have something other than a flat general programming model. TinyOS uses component-based design via the nesC language, while both LiteOS and SOS use module-based approaches in GNU C[1]. We put our Lorien aims in the 'Components' category to gain the strong architectural reasoning promoted by TinyOS; in contrast to TinyOS however Lorien maintains its component separation

[*]This work has been partially supported by the ICT Programme of the European Union under contract number ICT-2008-224460 (WISEBED).

[1]There are various definitions of a 'module'; here we refer to loadable units that employ an ad-hoc architectural composition approach rather than the strong provided/required interface-based architecture of a component model.

Programming model / OS	Image / Flat	Modules	Components
TinyOS			x
SOS		x	
Contiki	x		
Mantis	x		
LiteOS		x	
Lorien			x

Figure 1: WSN OS general programming models

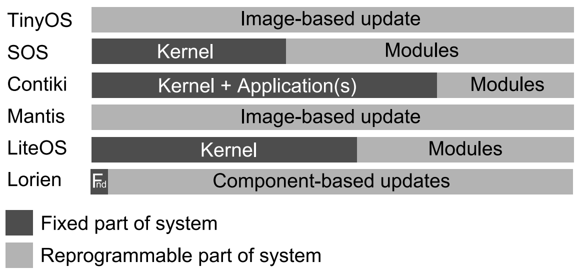

Figure 2: WSN OS reprogramming support

after compilation.

Next we look at the runtime reprogramming models of the same OSs, summarised in Figure 2. This property represents the model available to the developer wanting to perform runtime updates of the deployed code in the WSN, for example when the WSN is deployed in a remote or dangerous location which is difficult to physically access. Figure 2 shows both the model with which that reprogramming takes place and a guide to the extent of the system which is reprogrammable.

Of the OSs surveyed, two enable whole-system reprogramming: TinyOS and Mantis. Because all TinyOS components are compiled into a monolithic system image under the nesC static component model, the system architecture post-deployment cannot be ascertained and so the entire image must be reprogrammed. In the case of Mantis the general programming model is simply a monolithic one and the reprogramming model follows this. Both such approaches require a system restart in order to apply an update.

The remaining OSs surveyed are essentially designed as an augmentable fixed image, where the size of the fixed part varies between different designs. Contiki's is typically the largest as applications are usually developed as part of the kernel image rather than as modules. These systems do not require a system restart to apply an update but clearly are limited in the extent of the system that can be updated in this fashion. We also note that the augmentable parts of these systems employ a different programming model to the fixed part.

With Lorien our aim was simply to enable whole-system reprogramming using a unified dynamic approach – an approach which does not require the system to be restarted to apply an update. It is important that the entire system – not just the non-kernel part – is updatable since we consider it equally likely for the kernel to require an update as for the rest of the system.

By using a dynamic component model to achieve this, we gain support for both the runtime loading and unloading of individual components *and* for runtime architectural change of the component graph. This in turn enables us to leverage components as a general *primitive of change*: not only supporting the basic need for remote reprogramming throughout the system, but also naturally supporting system augmentation or even autonomous system adaptation[2].

In reality Lorien does have a very small fixed part of the system, as discussed shortly; this is however a simple bootloader rather than any form of kernel.

[2]By contrast, FlexCup [8] for example is an approach which allows reprogramming of TinyOS by (offline) image-based component swapping, but does not allow architectural change such as system augmentation.

3. LORIEN

We now describe our Lorien OS, presenting a unified approach to general-case programming and whole-system reprogramming. We first briefly discuss what a component is in Lorien and what we mean by a dynamic component model. We then discuss how we designed a fully dynamic operating system around this component model, in which *every* part of the system is a dynamically replaceable component, highlighting the key enabling elements of this design.

3.1 A Lorien component

Lorien is built upon the well-established OpenCom component model [4]. A component in Lorien is a strongly separated, self-contained unit of functionality – components are in fact compiled separately and then composed into a system as appropriate. The same compiled component can either be composed into the initial system image or can be sent later over-the-air.

A component is *instantiable* (i.e. multiple times) and advertises both its *provided* and *required* interfaces to a component runtime, allowing third-party entities to reason about its architectural dependencies. Figure 3 shows a typical component, written in plain C, with its constructor, destructor and function implementations. The constructor registers the component's provided and required interfaces, and links its implementations of its provided interface functions to the interface's abstract functions. The destructor performs the reverse, cleaning up this state. Components can also have per-instance state which is not shown in this example, but which would similarly be created and registered in the constructor, and detached and cleaned up in the destructor.

Such a component is compiled into a standalone, loadable binary unit using the common position-independent ELF format (known as a 'shared object' in Linux). Architecturally the component shown in Figure 3 appears as in Figure 4 with one *provided* and one *required* interface of the respective types – a component can have 0-N of both kinds of interface. A *required* interface must be connected to another component's *provided* interface to satisfy the dependency.

What makes Lorien's underlying component model *dynamic* – as opposed to the *static* model of TinyOS – is that components can be *instantiated* and *destroyed* at runtime, and components can be *connected* and *disconnected* at runtime. Furthermore components can be completely – and independently – *unloaded* from the system image at runtime, or other components *loaded* and integrated into the running system image. There are no constraints over how much the system architecture – and system image – can change in this fashion while running. A component runtime keeps track of what the current architecture is, and all archi-

```
typedef struct recplist{
   IRadio *radio;
   } RecpList;
#define RECPS ((RecpList*) comp − > recpList)

int send(Component *comp, Component *binding,
         unsigned char *msg, size_t len, TID to) {
   /* ...  encryption ...  */
   return CALL(RECPS − > radio − > send, msg, len, to);
   }
/* ...  other functions ...  */

int construct(Component *comp) {
   int err = OPENCOM_OK;
   IRadio *ir;
   /* register provided interfaces */
   if ((err = regInterfaces(comp, 1, "IRadio", &ir, sizeof(IRadio)))
         != OPENCOM_OK)
      return err;

   ir − > send = send;
   /* ...  link other functions ...  */

   /* register required interfaces */
   if ((err = regRecpList(comp, sizeof(RecpList), 1, "IRadio",
         sizeof(IRadio))) != OPENCOM_OK) {
      delInterfaces(comp, 1, "IRadio");
      return err;
      }
   return err;
   }

int destruct(Component *comp) {
   delInterfaces(comp, 1, "IRadio"); delRecpList(comp);
   return OPENCOM_OK;
   }
```

Figure 3: An example 'secure radio' component

tectural changes are made via this runtime to help enforce (re)configuration safety.

3.2 Lorien: A pure dynamic component OS

Lorien is constructed from a collection of components like that in Section 3.1. Just before we describe its design a quick primer on WSN hardware is necessary for completeness. Those very familiar with WSN hardware can safely skip this paragraph. The main point we need to make is that current WSN hardware platforms are organised as a 'CPU' (actually termed an *MCU*) with on-board *program memory*. This program memory holds the entire program executed by the WSN node, and generally takes the form of solid-state flash memory that is re-writeable during program execution. The procedure to initially upload a program to a WSN node is to plug that node into a PC (e.g. via USB) and use a special program which takes a compiled image and injects it directly into the MCU's on-board flash memory. The WSN MCU is then rebooted and jumps to a specific address in its program memory at which it expects the first program instruction to reside (this address is hardwired). For the sake of this discussion this first instruction is essentially the entry-point to the `main()` method in a C program. Most WSN platforms additionally have a secondary flash memory chip, external to the MCU, which is typically of a much larger capacity than the MCU's on-board memory. The MCU's memory is executable while external flash memory is not. MCU program memory is furthermore automatically readable from standard C code as if it was a byte

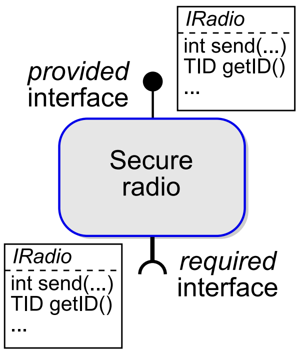

Figure 4: Illustration of the component in figure 3

array without the need for any driver support.

The basic premise behind Lorien's design is that the currently configured system – the collection of components currently in use – resides in the MCU's on-board flash memory. The external flash memory chip is used as a component pool for all other components that can be used in the system if desired. Components can be downloaded over the node's radio (or any other interface) and stored in external flash memory; components can be individually *loaded* from external flash memory into MCU program memory. Components in MCU program memory can be instantiated, interconnected with other components, and destroyed – and can be individually *unloaded* (i.e. deleted from that memory area). Components stored in external flash memory can additionally be deleted if not required for loading at any point in the future. All of this can be done without stopping / restarting the system.

The key challenge in Lorien's design is how to jump from a flat C program into a fully dynamic component-oriented system, considering our aim for whole-system dynamics under a unified model – meaning that the loader mechanism itself needed to be a component, able to be loaded and unloaded just like any other component, but which clearly could not initially load itself. This kind of circular dependency is indicative of the core challenge that we needed to solve.

Our solution bought us to a design involving two notional parts: the *system core* and *the rest of the system*. Both are constructed using the same kinds of components and can be modified in exactly the same ways; the only difference is that the *system core* is specially composed by our Lorien toolchain and requires some (fully automated) code generation. Our solution therefore requires that the *system core*'s source-code is available to our toolchain, whereas *the rest of the system* is expected to be a collection of precompiled components. Once Lorien is actually running on a WSN node, system core components can be sent individually over-the-air and replaced – or indeed the architecture of the system core entirely changed – in the running system just as any other part of the system.

The main area of interest is therefore the system core, the components of which we show in Figure 5.

The roles of these components are as follows: The *system core configurator* is a special component which helps to bootstrap the system. The *component runtime* holds and affects the current system architecture as already discussed, and the *dynamic loader* is responsible for loading a component from external flash memory into MCU program memory,

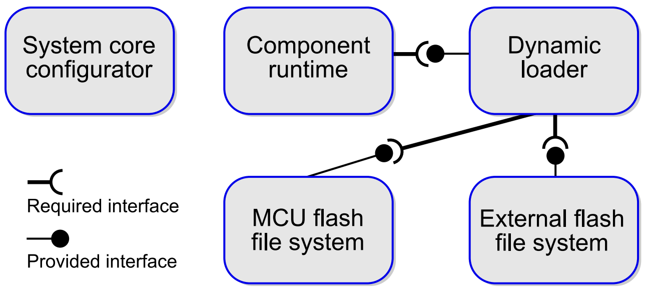

Figure 5: The Lorien System Core Components

or for unloading a component from MCU program memory. The *MCU file system* component is required to maintain a filesystem within MCU program memory in support of actually loading and unloading components – a loaded component is thus a file in the MCU filesystem. The *external flash file system* similarly maintains a filesystem in external flash memory, allowing components (or any other files) to be stored, read and deleted from external flash memory.

In terms of dependencies, all components in the entire system inherently have access to the component runtime; the component runtime itself uses the dynamic loader when any component requests that another component be loaded or unloaded. The dynamic loader in turn uses the MCU file system and external flash file system to perform the loading and linking of a component (i.e. shared object) from external flash memory, or else the unloading of a component. Both filesystem components are additionally available for general use by any other components in the entire system.

The job of our Lorien toolchain is to create an initial system image, for upload to a directly-connected sensor node, which results in the instantiation of this fully reconfigurable system architecture when it boots.

The crux of the solution lies in i) the offline generation of the *initial MCU filesystem state* (i.e. its files and file table) and ii) the generation of a *boot configuration file* describing how the initial architecture should look and where the key offsets into MCU program memory are to help build this architecture. The initial MCU filesystem state generated by our toolchain appears as in Figure 6 – as can be seen it essentially *pre-loads* the key components, creating a kind of snapshot in time reflecting a state as though these system core components had been loaded by the system core itself.

The *fundamentals section* shown in Figure 6 contains the fixed `main()` method of the entire system – the hard-wired address to which the MCU jumps when it starts. This section also contains a number of additional things which we return to later. The boot procedure on a sensor node thus works as follows:

3.2.1 Starting the configurator component

The fixed `main()` method reads from a special fixed address in MCU program memory (an address generated in the `main()` method's code by our toolchain). The value found at this address is itself an address in MCU program memory of the system core configurator's special `boot()` function – a non-component entry function, which the `main()` function now invokes having acquired its address. Unbeknown to the `main()` method, the special fixed address from which it read is actually within a file in the MCU filesystem; the contents of this file can be modified later to change the jump address if the system core configurator component is changed. The *location* of this file of course cannot be changed but it has no reason to as it resides at the very start of MCU program memory, labelled as [A] in Figure 6.

The `boot()` function of the system core configurator component has one task: Boot the component runtime, then use the runtime to instantiate the system core configurator component (i.e. itself) proper – the remainder of its duties are then undertaken in its constructor which is within component space.

3.2.2 Starting the component runtime

To achieve this the configurator component first reads from a *boot configuration file* which details various parts of the boot procedure, including the address of the component runtime's own special boot function. This boot configuration file is initially generated and placed in MCU program memory by our toolchain, and is found by the configurator at runtime by reading from one more special fixed address in program memory ([B] in Figure 6). This address is again generated in the configurator component's source code by the toolchain and its contents can later be modified as a file should the configuration file's position change.

By invoking the component runtime's own boot function, the configurator acquires a reference to the component runtime's core interface through which all other tasks can be performed (such as instantiating and interconnecting other components). The configurator's immediate task is to use this interface to request that an instance of itself be created, causing its own constructor to be invoked, in which the remainder of its tasks are carried out[3].

3.2.3 Building the system core

The configurator component's remaining tasks are to load, instantiate and interconnect the other components in the system core. The boot configuration file instructs it on how to perform these duties and the configurator simply follows this list of instructions using a simple generic parser. The configurator additionally performs relevant state injections into selected components as instructed by the configuration file, for example to inform the dynamic loader component of the components that are already loaded and their file handles for use with the MCU file system component.

We now have a system core which is fully instantiated, interconnected, but more importantly is *unloadable* and *architecturally reconfigurable* in general. Any changes to the system core components are notified via the system core configurator which updates the boot configuration file as appropriate – and if necessary its address-of file [B] – using the MCU file system component. All system core components have state transfer interfaces to allow staged hand-off to alternative components. For example, a new dynamic loader can be loaded by the existing loader, state can be transferred into that new loader from the current one, and the new one

[3]In the interest of simplicity we have missed a step here in which the configurator uses the component runtime's interface to inject state informing the runtime of the list of components that are pre-loaded – by doing this the component runtime avoids trying to actually load components (with a non-yet-created dynamic loader component) when instantiation requests are made since it can see that they are already loaded into program memory.

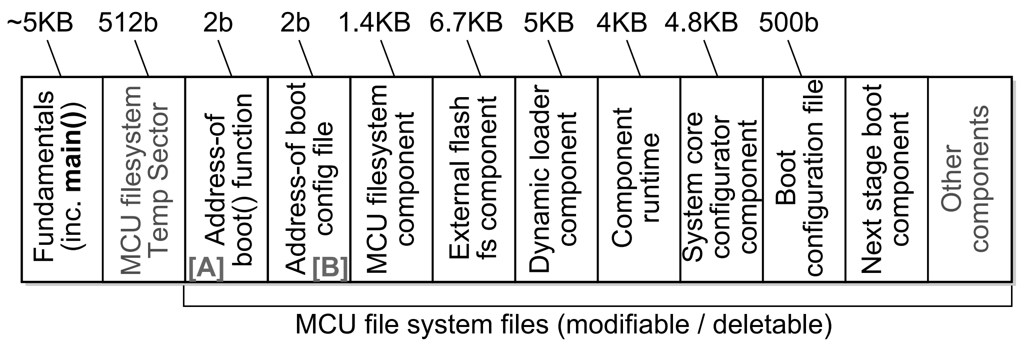
~5KB
512b
2b
2b
1.4KB
6.7KB
5KB
4KB
4.8KB
500b
Fundamentals (inc. main())
MCU filesystem Temp Sector
Address-of boot() function
[A]
Address-of boot config file
[B]
MCU filesystem component
External flash fs component
Dynamic loader component
Component runtime
System core configurator component
Boot configuration file
Next stage boot component
Other components
MCU file system files (modifiable / deletable)

Figure 6: The initial image created by the Lorien toolchain for MCU program memory

can unload the existing loader. Such procedures work because a dynamic loader's list of loaded components includes itself – and similarly the MCU file system's file table includes an entry for its own program code in MCU program memory.

3.3 Additional details

In this section we address some outstanding questions about how some parts the mechanisms described in Section 3.2 work – which we omitted in order to avoid distracting from the general discussion.

3.3.1 The rest of the system

When the system core configurator finishes its own boot procedure it reads from the boot configuration file the source name of a component to use as the 'next stage boot' component. This next stage boot component must provide the *IBoot* interface, and the configurator instantiates the component, acquires a reference to this interface and invokes its `boot()` function, passing a reference to itself as a parameter through which other system core components can be accessed.

The next stage boot component performs any of its own boot duties in this function and calls a special function `setMain()`, passing a reference to its own component instance, and a reference to its own (component) `main()` function, as parameters. When the next stage boot component's `boot` function terminates, the system core configurator's boot function can also finally return, and the original `main()` method of the entire system invokes the 'main' function last registered with `setMain()`, passing in the corresponding component instance reference. If a component `main()` function ever returns then the original system `main()` function simply again invokes the last registered 'main' function, allowing a hand-off of next stage boot components. It is within the component `main()` function of a next stage boot component that the developer's code resides – this function could for example contain a thread scheduling or message handling loop. This procedure is illustrated in figure 7.

We note that all other components can also of course provide state-transfer interfaces to enable them to perform similar staged hand-offs to those employed by the system core.

3.3.2 The fundamentals section

The `setMain()` function is an example of a *globally linked symbol*, and its implementation exists within the 'fundamentals' section. The dynamic loader component has a list

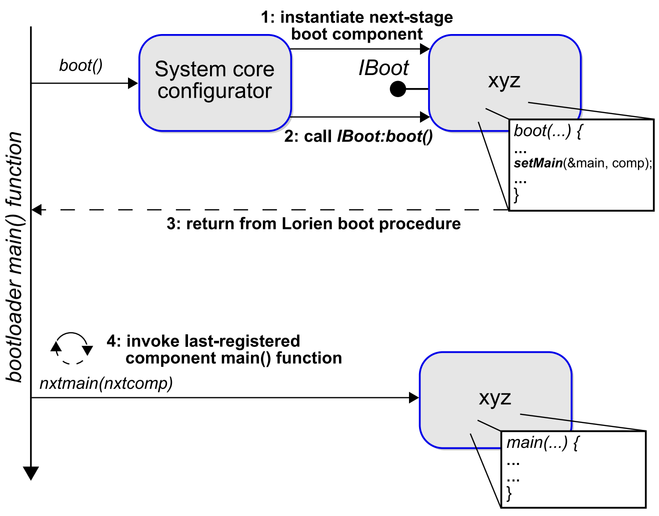
1: instantiate next-stage boot component
boot()
System core configurator
IBoot
xyz
boot(...) {
...
setMain(&main, comp);
...
}
2: call IBoot:boot()
3: return from Lorien boot procedure
bootloader main() function
4: invoke last-registered component main() function
nxtmain(nxtcomp)
xyz
main(...) {
...
...
}

Figure 7: The 'rest of the system' handoff

of such globally linked symbol names – generated by the toolchain depending on what the developer wanted to have in the fundamentals section – and whenever it loads a component it links all of these symbols with their corresponding addresses. The other globally linked symbols of note are `setISR()` and `clearISR()` which allow a component to register interrupt handlers. The developer can additionally include as many other functions as they would like in the fundamentals section of their system (such a list is provided to the Lorien toolchain when it is building the initial system image). The number of functions included determines the size of the fundamentals section, and so the size of the non-replaceable part of the system since this program code exists outside of component space.

3.3.3 Statistics

While we do not provide a detailed evaluation of Lorien in this paper, we highlight some of the main statistics of interest. The default Lorien system core (of course its components can be selected as desired to trade off size, capabilities and performance) takes 28KB of program memory on the TelosB platform, including the fundamentals section, leaving 20KB for the rest of the system.

A typical size for the fundamentals section itself is around 5KB, including the most commonly used global symbols in

C (the mandatory bootloader part of this section is just 1.2KB). On the TelosB platform this leaves 43KB for dynamic components. A breakdown of ROM costs among system core elements is provided in Figure 6.

The default system core additionally uses 3KB of RAM, leaving 7KB for the rest of the system. Given its capabilities we believe that these figures are promising and compare well with other reprogramming approaches (e.g. [8]).

3.4 Summary

The key enabling point to Lorien's design – besides a dynamic component model – is the generation of an initial image containing a snapshot of the system in time such that a number of components are effectively *pre-loaded* in MCU program memory as if the system core had loaded itself. Additionally the fact that program memory is automatically readable as a generic memory area, without the need of the MCU file system component, is leveraged to allow a configurator component to read a file detailing how to boot the system, with a collection of addresses in program memory generated in this file by our toolchain to help do this.

This approach allows *every* component in a Lorien system to be changed at runtime, and indeed any aspect of the system architecture to be fundamentally changed using a single, unified programming model.

4. CONCLUSION

We have presented a survey of existing WSN operating systems in terms of their programming models and runtime reprogramming support, and shown that no existing OS shares a common approach across these concerns at the dynamic end of the spectrum. We have explored how to create such an OS and have described the design of Lorien, a dynamic-to-the-core component OS which leaves on average 43KB – 90% – of TelosB MCU program memory available for the dynamic component system.

Such a system can have any of its components remotely and independently updated, including those in the system core, can download new applications to add to the system, or can autonomously perform adaptations using a pool of alternative components all while the system is still running. All of these capabilities are achieved using a dynamic component model in such a way that the system can reason about a component as a *unit of change*, and can more broadly reason about the system architecture in terms of inter-component dependencies and connections.

We are aware that we have not mentioned concurrency models, scheduling, network stacks or a range of other traditional OS concerns; hopefully however the reader can see that such concerns are beyond the scope of the Lorien system core and therefore fit within the realm of 'the rest of the system'. A concurrency model for example can be encapsulated within a component and treated the same way as any other component in the system – loaded, unloaded, updated, etc.

Any number of additional components can be included in the initial system image, as long as they will fit into MCU program memory; further components can be downloaded later into external flash memory. A common initial image that we use for example is a basic shell component which can be communicated with over the TelosB serial interface, and to which we can send components to be stored in external flash memory and then load and instantiate whichever of these components represents the entry-point component of the system (which will in turn load and configure the rest of the system).

From the point of view of the developer the complexity of Section 3.2 is completely hidden – as far as the average Lorien developer is concerned they write (or select) a next stage boot component and inform the Lorien toolchain of which component this is as well as any other components they would like to be included in the initial system image.

In future work we plan to perform extensive evaluation of Lorien to establish the full cost of the unified dynamics that it provides, and also develop system design processes to help developers easily create dynamic systems with self-configuring properties. Already however we believe Lorien supports rich future research in middleware and general software systems for wireless sensor networks. Lorien is an open-source project available for download from [1].

5. REFERENCES

[1] Lorien on SourceForge. http://opencomc.sourceforge.net/lorien/.

[2] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. *Mobile Network and Applications*, 10(4):563–579, 2005.

[3] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He. The liteos operating system: Towards unix-like abstractions for wireless sensor networks. In *IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks*, pages 233–244. IEEE Computer Society, 2008.

[4] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Transactions on Computer Systems*, 26(1):1–42, February 2008.

[5] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462. IEEE Computer Society, 2004.

[6] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176, Seattle, Washington, USA, June 2005.

[7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGOPS Operating Systems Review*, 34(5):93–104, 2000.

[8] P. J. Marron, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *Third European Workshop on Wireless Sensor Networks*, pages 212–227, February 2006.