Type-Safe Updating for Modular WSN Software

Barry Porter Utz Roedig Geoff Coulson
School of Computing and Communications, Lancaster University, UK
Email: {b.porter, u.roedig, g.coulson}@lancaster.ac.uk

***Abstract*—Modular software, in which strongly-separated units of functionality can be independently added to and removed from a node's running software, offers a promising approach to effective dynamic software updating in Wireless Sensor Networks (WSNs). Modular software updating approaches offer high efficiency, in terms of both network costs and update installation at nodes, as well as low disruption, allowing existing software to continue to operate during updates. Existing approaches however critically lack *safety*, relying on weakly-typed event-based programming abstractions for inter-module interaction. This precludes compile-time or composition-time verification of interoperability between dynamically loaded modules and therefore presents major risks for future large-scale production-class deployments. In this paper we present Lorien: a component-based modular operating environment that employs *interface-based* inter-component interaction to support completely type-safe software composition, while still supporting high update efficiency and low disruption. Our approach also has very wide scope, allowing almost 90% of software to be remotely updated on common sensor platforms such as the TelosB. We compare Lorien against existing modular designs, finding that the safety properties of Lorien are offered with near equal efficiency.**

I. Introduction

The software deployed in Wireless Sensor Networks is seeing an increasing need to be updated post-deployment. Motivations for software updating range from remote correction of errors; to fixing inefficiencies in a particular deployment environment; to facilitating adaptation to observed environmental changes; or even to the inexpensive repurposing of generalised hardware as the needs and priorities of a deployment's stakeholders evolve. This is particularly relevant in large-scale environments such as the emerging 'smart cities' infrastructures being developed around the world (e.g. [1]).

Early work on software updating based on monolithic images (e.g. [2], [3]) tended to incur high update costs as well as service disruptions by requiring updated nodes to be rebooted. More recent *modular approaches* (e.g. [4], [5]), in which nodes are updated incrementally by adding or removing strongly-separated units of functionality, offer a more promising approach to achieving the required levels of software agility with low update costs and low disruption. The state of the art, however, achieves modularity at the cost of *type-safety* in inter-module interaction. In SOS [5] for example modules exchange 'events' in the form of opaque memory blocks with assumed internal structure to which it is not possible to apply either compile-time or composition-time verification of modular interoperability as successive updates are applied. Uncoordinated changes to event type identifiers or the internal structure of events can then leave sensor nodes open to critical software failures caused by ill-informed developer assumptions or simple mistakes. While acceptable for the small experimental deployments of today this fails the needs of large-scale production-class deployments that demand guarantees of correct functioning over an evolving software fabric. In addition the majority of current modular solutions lack in the scope of software that can be remotely updated.

In more detail we characterise WSN software update solutions using the following four criteria:

1) **Efficiency:** How much data is needed for an update and how costly is its application at a node? This impacts on the energy expended to disseminate and apply updates.
2) **Disruption:** How much does the enactment of an update disrupt running services? The lower the disruption the less data loss and service loss incurred.
3) **Safety:** How confident can we be that an update will function correctly at its deployment site? High safety is strongly required in production-class deployments.
4) **Scope:** What proportion of a node's software is updatable? This impacts on how much of deployed software can be operationally enhanced without a site visit.

Generally, classic image-based WSN work has addressed criteria (3) and (4); while more recent modular software has addressed criteria (1) and (2).

As an example, image-based updates in TinyOS-Deluge [2] offer *low efficiency* since entire images must be distributed and stored, and *high disruption* because the enactment of an update causes all of program memory to be re-written followed by a node restart. But it offers *high safety* because new images are subject to compile-time verification before deployment, and *wide scope* since almost the entirety of a node's software image can be changed. In contrast, Contiki [4] exhibits *high efficiency* because only a single application module is sent as an update, and *low disruption* because existing software on a node continues to function after addition of a new application. But it affords *low safety* because application/kernel linkage is weakly typed, and *narrow scope* because only the application-level is susceptible to updating. Finally, SOS [5] also offers *high efficiency* and *low disruption* due to its modular updates, but still suffers from *low safety* by employing a weakly-typed event-based composition and interaction model. It offers somewhat *wider scope* than Contiki due to its more generalised modular design but this falls well short of TinyOS-Deluge.

The key contribution of this paper is an approach that satisfies **all four** of the above criteria in the context of strongly-typed modular software updating in our *Lorien* system. As well as *high efficiency* and *low disruption* Lorien offers *high safety* because software modules (or 'components') interact

This work was partially supported by the ICT Programme of the European Union under contract number ICT-2008-224460 (WISEBED).

exclusively via strongly-typed formal interfaces. This allows both compile-time type checking and composition-time verification such that Lorien nodes will refuse to allow modules to interact if their interface types are incompatible. Lorien also offers very *wide scope* by employing a common component model across both application- and OS-level functionality.

In the remainder of this paper we first in Section II provide a more detailed discussion of related work, then in Section III present our type-safe modular updating approach. In Section IV we provide an evaluation of Lorien against contemporary modular updating systems according to the above four criteria and we offer our conclusions in Section V.

II. Related work

Research on software updating in WSNs is an ongoing topic, the work of which broadly fits into three categories: i) approaches to update static image-based software; ii) approaches to modularising software into dynamically loadable units; and iii) virtual machine approaches supporting the dynamic addition of interpreted code. A survey is provided in [6]; here we characterise this work in efficiency, disruption, safety and scope, and also provide an update on more recent work.

1) Image-based solutions: Work in this category relates primarily to TinyOS [7], with Deluge, differential patching, and FlexCup being the main works. A deployed TinyOS image is a static, monolithic binary with no internal structure, designed to maximise runtime efficiency in space and energy.

The Deluge [2] approach offers the ability to replace an entire TinyOS image by sending a complete new image over the network to nodes, storing it in external flash memory, and then erasing program memory and re-writing it with the image stored in external flash. As mentioned, Deluge has poor efficiency and high disruption, but high safety and broad scope.

Because use of the radio is one of the highest costs in WSN energy expenditure, binary differential patching (e.g. [8], [3]) proposes a more efficient solution which retains high safety and broad scope. While this approach does exhibit higher efficiency than Deluge the patch procedure at nodes remains relatively expensive in energy, particularly in [3], and still exhibits high disruption needing a node restart.

2) Modular solutions: Modular solutions load and unload strongly separated units of machine code to/from the running software, aiming to decrease disruption by removing the need to reboot a node to apply updates. The majority of work in this area can be characterised as 'kernel/application' systems, of which Contiki [4] is representative; limited other work—e.g. SOS [5]—points towards more generalised modular systems.

In more detail, approaches such as Contiki (e.g. [9], [10], [11], [12]) use a large fixed kernel in which the majority of node functionality resides, including network protocols, scheduling and sensor drivers, atop which applications can be added post-deployment and linked against any kernel functions or symbols that their machine code references. These approaches offer high efficiency of software updates, only needing the application code to be transferred to nodes, and low disruption, allowing a node's software to continue to function during software updates. They have low safety, however, due to weak typing of dynamic application-kernel linkage which is based on a mixture of opaque event objects and string-name symbol equivalences. Furthermore they offer narrow update scope—precluding e.g. the replacement of a routing protocol independently of the application(s) using it—and are further unable to correct errors, inefficiencies or functionality deficits that their large fixed kernels may be causing in a given deployment. In addition, they have no support for inter-module composition—i.e. dynamically loaded code cannot use the functionality of other dynamically loaded code.

SOS [5] is a truer modular system, supporting inter-module composition and interaction, though still using a relatively large fixed kernel. SOS uses a uniformly-applied asynchronous event model for both module-kernel and inter-module interaction such that each module is essentially designed as a switch-case statement operating on event types. Modules emit events into an event bus and they are passed to any registered subscribers of those events. As with all modular updating systems SOS offers high efficiency and low disruption, and SOS offers somewhat higher scope than the above. However, SOS and related approaches (e.g. [13]), suffer from low safety due to the weak-typing of asynchronous events exchanged between dynamically loaded modules which are essentially opaque memory blocks with assumed internal structure.

Generally, all of the above modular approaches can be seen as a step back from image-based solutions in their level of compile-time verification, reducing the certainty that two elements of software that are expected to be able to interact will correctly function when deployed. This critically reduces confidence in the use of such modular systems in serious production-class deployments despite their potential for high-efficiency and low-disruption software updates.

3) Virtual machine solutions: Finally, VM research for WSNs focuses on easing the programming model by raising the abstraction level, in some cases as high as Java, with the added benefit that interpreted code can in principle be sent to and loaded at sensor nodes with high efficiency and low disruption. Representative examples are Maté [14] and Darjeeling [15]. Both argue that the cost of interpretation is acceptable given the resulting lower cost of software updates. Both, however, focus almost entirely on the efficiency of the VM implementation itself. Disruption is given little consideration, and in terms of scope, work to date is comparable to the kernel/application modular systems discussed above, with the VM acting as the large fixed area of software and the interpreted code units representing the loadable applications.

III. Lorien: A Type-Safe Modular OS

In this section we first in Sec. III-A present our approach to modularisation; in Sec. III-B we then discuss our approach to the safe composition of modules; and finally in Sec. III-C we discuss the safe evolution of a node's software composition based on a set of six software update instructions.

A. Modularisation

Our approach in Lorien is to provide a way to independently *add* and *remove* any part of a node's software while the

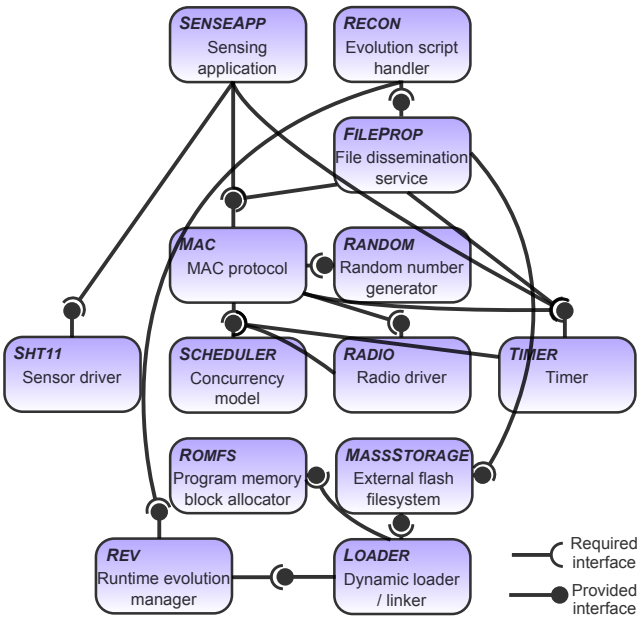

Fig. 1. Architecture of a typical Lorien system. Some inter-component connections are omitted to avoid clutter. Each shaded box can be independently added/removed, and the entire architecture incrementally changed, at runtime.

remaining software stays online (*replacing* a module is then just a removal followed by an addition). This promotes high update efficiency as only updated modules need to be sent to sensor nodes; and low disruption as software can be changed without disturbing surrounding software. Our approach also affords very broad scope as we uniformly apply it to low-level code that is usually wrapped into a monolithic non-updatable kernel. This broad-scope approach is illustrated in Fig. 1 which shows a complete OS configuration composed of strongly-separated units of functionality, from application concerns down to hardware drivers.

Lorien modules are written in a minimal component-oriented extension of the C programming language which is converted to plain C for compilation. This language models strongly-separated units of functionality as *components* that interact exclusively through *formal interfaces*, providing strong typing for software composition. Compared to nesC [16], as used in TinyOS, our component language firstly does not use 'split-phase' (which tends to complicate systems programming [17]) while still generally using an event model, and secondly uses explicit constructors and destructors for dynamic component instantiation.

In more detail Lorien components are realised as *types* and *instances*. Component types are sent to nodes in relocatable object files (cf. [18]) that are stored in the node's external flash memory chip. The machine code of a component type can then be loaded into program memory when needed and instantiated any number of times, each instance with its own local state and able to be connected into the surrounding system in its own way. Memory (RAM) for component instances, their interfaces and state is dynamically allocated on demand.

The memory layout of a Lorien system is shown in Fig. 2. The small kernel contains only a tiny bootloader and dynamic interrupt table. The machine code of each loaded component

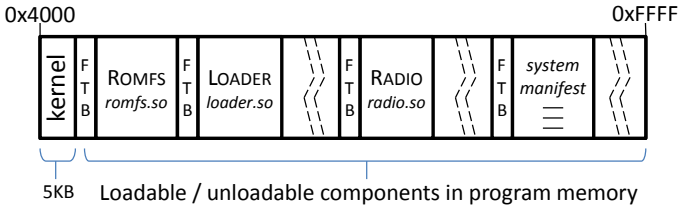

Fig. 2. Memory layout of Lorien on a 48KB TelosB node, with a small fixed kernel and the rest of memory incrementally modifiable as strongly-separated components. Each loaded component type sits in a program memory block allocated by the ROMFS component and can be unloaded by freeing that block (block descriptors are organised in a linked list, marked FTB in the above).

type is placed in its own allocated block of program memory and can be 'unloaded' by freeing that block. In Lorien even the program memory block allocator and dynamic loader/linker are components that can be added/removed with generality. Details of how the highly-modular image in Fig. 2 is created for initial flashing to a sensor node are available in [19].

The source code of an example component type is shown in Fig. 3. The component declares itself to *provide* a collection of interfaces, and to *require* a collection of interfaces, as well as specifying per-instance state. Component instances access their state via a reference passed in to each function call (hidden by our language layer in Fig. 3). A 'runtime evolution manager' (REV in Fig. 1) maintains a list of currently loaded component types, instances, instances' required and provided interfaces, and the current interconnections between the latter. As shown in Fig. 3, a component's constructor is responsible for dynamically registering its interfaces and state with REV.

Note finally that components do not themselves attempt to connect their required interfaces to other components, instead leaving this task to REV.

B. Composition: Safety and architecture

In this section we discuss firstly how safety is achieved and secondly how component interconnections are specified.

1) Safety with strongly-typed interfaces: Safety is upheld in two ways. Firstly, we use formal interfaces for interaction to allow *compile-time* checking of type compliance in the use of those interfaces by each individually compiled component type. And secondly, we use hash-codes generated at compile-time to allow *composition-time* verification of type equivalence of the required/provided interfaces of different components.

In detail, a *provided interface* is essentially a list of function pointers; its type is identified by both a string name and a hash-code, where the hash code is based on the entire interface type including the parameter types and return values of all of the interface's function prototypes. Each function pointer in a provided interface is pointed at the component type's implementation of that function. A *required interface* is then a string name and hash code, identifying its type, plus a null list of function pointers able to be pointed at those of a type-matched provided interface. This forms a 'connection' such that the provider's functions can be called by the owner of the required interface. Example interface types are shown in Fig. 3 (i.e. *ITimer*, *ILife*).

Components can declare multiple same-type provided or required interfaces using a *type:variant* notation, e.g. a required

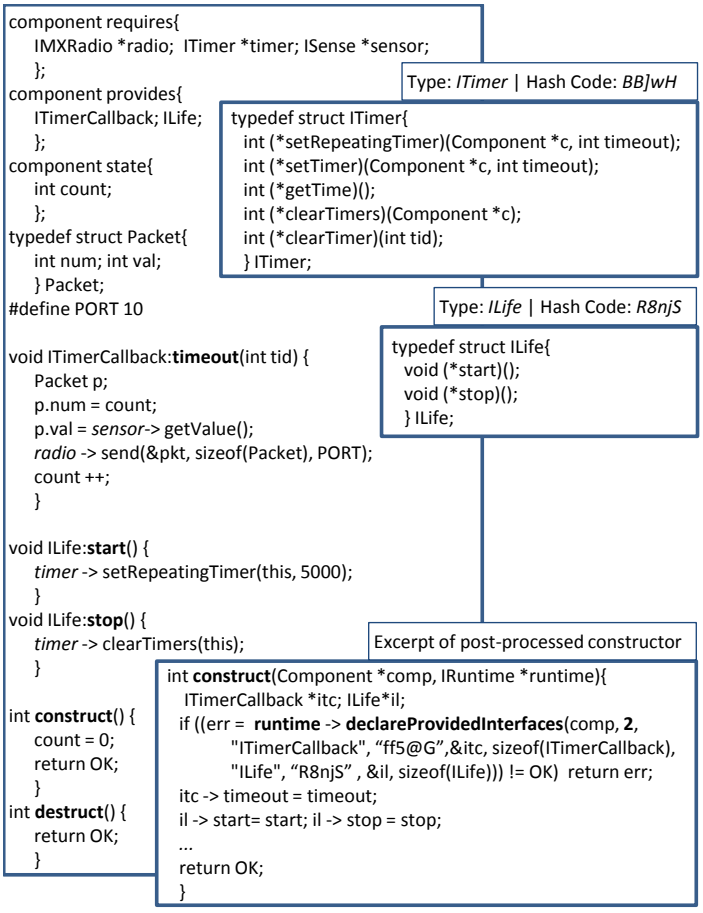

Fig. 3. An example Lorien component type which sends a message every 5 seconds. This is converted to a plain C representation by our preprocessor. Note the synchronous call model for example with *sensor -> getValue()*, despite an underlying task scheduler that fires *timeout()* as a task. Two example interface definitions (*ITimer* and *ILife*) are also shown. The *start* and *stop* functions of the *ILife* interface are part of our integrity model, discussed in Sec. III-C. Finally an excerpt of the post-processed plain-C constructor is shown that registers the instance's provided interfaces with the runtime evolution manager. This is derived from the component's specified interfaces and the *construct()* operation in the component language specification.

interface *IMXRadio:Primary* compatible for connection to any provided interface of type *IMXRadio*; or a generic required interface *ISense* compatible for connection to a provided interface *ISense:Temperature* on an SHT11 driver.

The use of function pointers in interfaces gives complete positional independence between components, allowing every component to be agnostic as to the location of the code (component types) that it calls. Additionally, because components register with REV the addresses associated with their required interfaces, REV can externally change component connectivity across the system at runtime. We characterise this interaction style as *type-safe* because at compile-time each component's use of an interface can be checked by the compiler for type compliance; and at composition time the required and provided interfaces of two components can only be interconnected if their type name *and* compile-time-generated hash codes match.

2) Architecture specification: The overall software configuration, or architecture, of a Lorien node is represented as a collection of per-component-instance *configuration fragments* that are held in a *system manifest* in program memory. Soft-

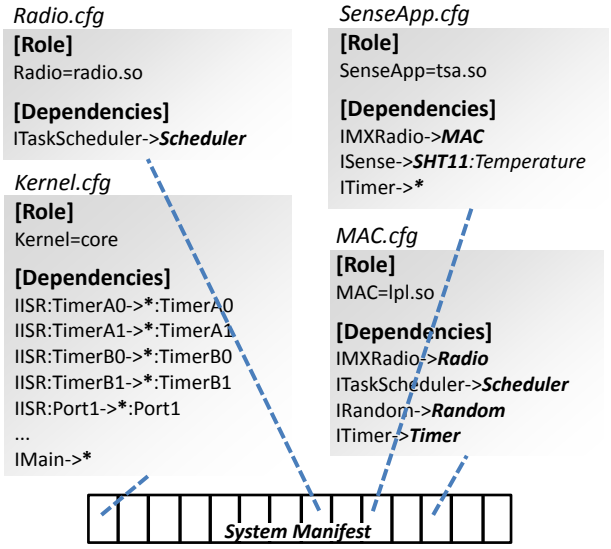

Fig. 4. Example configuration fragments linked from the system manifest. The textual form shown is converted into a compact binary format for tin-time use, though the string names are preserved. The *SenseApp* configuration fragment matches the component type shown in Fig. 3.

ware updates are effected by manipulating the manifest to add and remove fragments (discussed shortly in Sec. III-C).

Configuration fragments are small files that specify a fragment (component instance) of the overall software architecture graph (such as the graph shown in Fig. 1). Example configuration fragments are shown in Fig. 4: they give a string name, called a 'role', to the associated component instance, along with the component type (object file) from which the instance should be created, and they define how the instance's dependencies (required interfaces) should be connected to other component instances in the architecture.

As an example, the configuration fragment 'SenseApp.cfg' in Fig. 4, defines the role SENSEAPP, which is instantiated from the component type `tsa.so`. It has dependencies on provided interfaces of type *IMXRadio* and *ISense* on the abstract roles MAC and SHT11 respectively, selecting the variant 'Temperature' from SHT11. It also has a dependency on the provided interface of type *ITimer* on any role on the manifest that provides this type (in practice the most recent such role on the manifest is chosen).

Note that *callbacks*, an important feature of systems programming that allows components for example to register for timeout events or notifications of data arrivals from the network, are not described by configuration fragments but instead are created on-demand. This is because callbacks tend to be predicated on arbitrary behavioural context such as 'timeout in 5 seconds'. A component requests a callback by passing in a self-reference to its own component instance ('this') as one of the parameters to a function such as *setTimer()*. A component offering callbacks (e.g. TIMER) then declares required interface variants such as *ITimerCallback:1* and directly requests REV to connect them to the compatible provided interfaces of components who pass in self-references.

Finally, with a uniformly-applied architecture approach, Lorien supports the addition and removal of even interrupt-facing components with complete generality and type-safety. To enable this, Lorien's small kernel is modelled as 'just

NewSenseApp.evo

[Actions]
RPL(*SenseApp, NewSenseApp.cfg****)***

[Dependencies]
NewSenseApp.cfg
tsa2.so

[Nodes]
ALL

Fig. 5. Example evolution script to replace the SENSEAPP component with a new version. The [Actions] section is a sequentially-executed list of Lorien's update instructions (see Sec. III-C2); the [Dependencies] section lists additional files on which this script's execution depends, including new component types and configuration fragments; and the [Nodes] section lists the IDs of nodes at which this script should be applied, or 'ALL' for all nodes.

another' component instance, with its own 'Kernel' configuration fragment as shown in Fig. 4. Each interrupt is then given its own formal required interface of type *IISR* varianted with the interrupt's name. When an interrupt occurs the kernel checks whether the associated *IISR* required interface variant is connected to anything and, if so, invokes its *isr()* function. The * notation in the Kernel fragment again indicates role-name agnosticism so that these required interfaces connect to any role that has a compatible provided interface.

C. Evolution: Controlling runtime software updates

Software updates on Lorien nodes are controlled by issuing instructions to REV. In a network of nodes this is achieved using *evolution scripts* that specify a list of update instructions to carry out. Scripts are sent to sensor nodes using a data dissemination protocol such as Deluge; an example script is shown in Fig. 5. The RECON component in Fig. 1 is notified of a script's arrival and issues the enclosed instructions to REV.

In this section we bring together the preceding theory to show how entire Lorien software architectures can incrementally evolve. We first present Lorien's runtime integrity model for safe addition/removal of components; we then define the above-mentioned software update instructions; and we conclude by providing examples of the use of these instructions.

1) Runtime integrity model: The ability to add or remove any component presents the potential for chaos in an executing system, wherein the chief concern is *what happens if component C calls a function via a required interface on component D* while D is being removed/replaced. Existing work solves this problem trivially either by avoiding multi-module composition entirely (e.g. Contiki), severely restricting update scope, or else by having all inter-module interaction proceed asynchronously in split-phase via an event bus which can simply detect a lack of subscribers for a published event and garbage-collect that event (e.g. SOS). In Lorien, by allowing components to directly interact using synchronous calls over type-safe interfaces, we gain high compositional safety and simplify the programming model. This comes however at the cost of additional complexity in disruption control.

Our solution is an *integrity model* that governs *when* components may be interconnected, along with a standard way to *start* and *stop* the functionality of any component.

Our integrity model is defined with respect to the three kinds of connection that can be associated with a given component C: *dependencies* of C, specified in configuration fragments; *callbacks* to C, created on-demand; and *dependents* of C (components with C as a dependency).

Our model imposes the following five integrity rules:

I) Before any dependent is connected to C, C must have all of its own dependencies satisfied.

II) Before any dependency of C is disconnected, all of its dependents must first be disconnected from C.

These rules ensure that no component can *call* a function on C (potentially causing C to make an inter-component call of its own) unless C itself is fully connected to its own dependencies, guaranteeing any such call's success. Further:

III) At the point that C becomes fully connected to its dependencies the *ILife:start()* function is called on C, allowing it to start up its functionality including the registration of callbacks such as timers.

IV) At the point that one of C's dependencies is to be disconnected, breaking C's full connectivity, C's *ILife:stop()* function is called before that disconnection is performed allowing it to gracefully shut down as if it were about to depart the system (including cancelling any callbacks).

These two rules manage the life cycle of C such that a task cannot be executed in C when its dependencies are not complete – which could potentially cause C to erroneously make a call to a missing dependency. While stopped the software architecture around C can be safely updated. Finally:

V) Before a component instance is destroyed, all of its dependents and dependencies must be disconnected from it while adhering to Integrity Rule II.

2) Software update instructions: We now present Lorien's set of six software update instructions. Implemented in REV, these instructions are based on our configuration fragment architecture approach and transparently embody the runtime integrity model discussed in the previous section. The update instructions are as follows:

- **INS(fragment)** Loads a configuration fragment from external flash memory into program memory and inserts it into the system manifest. This in turn causes the associated component type to be loaded into program memory (if not already loaded), and an instance of that type created and mapped to the fragment's role name.
- **CNL(roleName)** (read: 'connect lower') Attempts to connect all dependencies of the given role. For each dependency a connection is only made if the specified role/interface exists *and* Integrity Rule I is satisfied. If it was possible to connect all dependencies of this role then `start()` is called on the component instance as per Integrity Rule III. `CNL` is typically called after `INS`.
- **CNU(roleName)** (read: 'connect upper') Attempts to (recursively) connect all dependents of the given role in adherence to Integrity Rule I. For each role whose dependencies become fully satisfied as a result of this process the `start()` function is called on them as per Integrity Rule III. `CNU` is usually called after `INS` and `CNL`. Pseudocode for `CNU` is shown in Fig. 6.

```
CNU(Role R)
if (fullyConnected(R))
  for each role X in manifest dependent on R:
    for each X->R dependency D:
      connect(D);
    CNU(X);

DCU(Role R)
for each role X in manifest dependent on R:
    DCU(X);
    for each X->R dependency D:
      disconnect(D);
```

Fig. 6. Integrity-model-compliant pseudocode of CNU and DCU.

- **DCU(roleName)** Disconnects all dependents of the given role (recursively). For each component whose dependencies lose full connectivity as a result the `stop()` function is called on them as per Integrity Rule IV. `DCU` is usually called in preparation to remove a role from the system. Pseudocode is also shown in Fig. 6.
- **DCL(roleName)** Disconnects all dependencies of the given role. If this role had all dependencies satisfied then `stop()` is called on it prior to the first disconnection as per Integrity Rule IV. Integrity Rule II requires a call of `DCU` to precede a call of `DCL` for a given role.
- **DEL(roleName)** Destroys the component instance associated with this role name, unloads its component type if no other instances of that type exist, and removes the configuration fragment from program memory, deleting it from the system manifest. According to Integrity Rule V, `DCU` and `DCL` must precede `DEL` for a given role.

Finally, if there are two same-named roles on the manifest we satisfy new dependency connections against the latest one, and conversely apply `DCU`, `DCL` and `DEL` to the earliest one.

3) Examples: We conclude this section by providing usage examples of the above instructions to evolve node software.

a) Generic addition, removal and replacement: Any component can be added to the system using the sequence `ADD` $= \{$`INS`; `CNL`; `CNU`$\}$; conversely any component can be removed using the sequence `REM` $= \{$`DCU`; `DCL`; `DEL`$\}$. Replacement then generally works simply as `RPL` $= \{$`REM`; `ADD`$\}$, assuming the configuration fragment added describes a component with the *same role name* as that just removed. When replacing components involved in the dynamic loading process (i.e. the dynamic loader itself, filesystems, and REV) the replacement must necessarily be added before removing the original, and can be achieved using the sequence `RPLA` $= \{$`INS`; `CNL`; `DCU`; `CNU`; `DCL`; `DEL`$\}$.

b) Boot sequence: At boot time all roles on the manifest are instantiated and those with zero dependencies have `CNU` called on them to initiate a recursive closure that connects their direct and indirect dependents. Consider the boot sequence for the system in Fig. 1: the roles with no dependencies are SHT11, MASSSTORAGE, ROMFS, SCHEDULER, and RANDOM. The order in which `CNU` is called on these roles does not matter, but assuming it is in the order stated, `CNU(SHT11)` connects SENSEAPP to SHT11; `CNU(MassStorage)` connects LOADER and FILEPROP to MASSSTORAGE; `CNU(ROMFS)` connects LOADER to ROMFS and then REV to LOADER and RECON to REV; `CNU(Scheduler)` connects TIMER, RADIO and MAC to SCHEDULER and then FILEPROP and SENSEAPP to TIMER; and finally `CNU(Random)` connects MAC to RANDOM and then FILEPROP and SENSEAPP to MAC, and finally RECON to FILEPROP, thereby completing the graph with all components started as soon as they became fully connected to their dependencies.

Component	Machine code size (bytes)	Object file size for transmission & storage (bytes)	INS time (ms)	INS energy (mJ)	DEL time (ms)	DEL energy (mJ)
REV	7840	11271	1714	22.52	80	1.53
LOADER	1808	2331	669	10.13	80	1.53
MASSSTORAGE	5046	6437	1195	16.35	80	1.53
ROMFS	1880	3005	813	11.00	80	1.53
FILEPROP	3506	4340	972	10.42	80	1.53
RECON	1714	2174	637	7.49	80	1.53
RADIO	3820	4969	972	10.73	80	1.53
MAC	2454	3212	782	8.67	80	1.53
TIMER	1556	2116	622	7.40	80	1.53
SHT11	788	1269	558	6.41	80	1.53
SCHEDULER	1108	1694	542	6.62	80	1.53
RANDOM	334	608	447	5.44	80	1.53
SENSEAPP	396	673	462	5.53	80	1.53
Average	2480	3392	799	9.90	80	1.53

Fig. 7. Update costs, including object file sizes and time and energy cost of adding / removing (and therefore replacing) each component in the configuration of Fig. 1. Energy is calculated based on observed current draw throughout the INS / DEL procedures with a supply voltage of 3V.

IV. EVALUATION

In this section we evaluate our TelosB implementation of Lorien. We begin by reporting empirical data on the 'efficiency' criterion. This is divided into update efficiency (subsection IV-A) and associated memory overheads (subsection IV-B). We then discuss disruption, safety and scope analytically in subsection IV-C. We base our evaluation on the configuration of components shown in Figure 1, which includes everything needed to disseminate and install updates in a WSN including a file dissemination protocol (FILEPROP) and a full-featured external flash filesystem (MASSSTORAGE).

As comparison points we use Contiki and SOS as representative of kernel/application and more generalised (but weakly-typed) modular approaches respectively. Lorien configures the TelosB's MSP430 MCU to operate at 4Mhz by default.

One aspect of overhead that we do not evaluate here (due to lack of space) is the cost of indirection in inter-component interaction based on function pointers. However, we note that existing work such as SOS [5] has compellingly demonstrated that these costs are not significant compared to general operating costs. Our own results concur with this.

A. Update efficiency

The efficiency of WSN software updates is a function of i) how much data needs to be transferred to and stored at nodes, and ii) how expensive it is to install the update at the sensor node once it arrives.

1) Transmission and storage: These costs arise from the three kinds of files that need to be sent to a Lorien node to update its software: evolution scripts, configuration fragments, and component type object files. The latter two are only needed if they have not previously been sent in earlier update cycles – an application component that is already in a node's external flash memory can be added/removed from the running system simply by sending an evolution script.

Evolution scripts and configuration fragments have almost insignificant transmission and storage costs: based on the experimental scripts we have written, evolution scripts are on average 50-100bytes (a 10-byte header plus ∼10bytes per instruction); and a configuration fragment for our example configuration is on average 134bytes (see subsection IV-B).

Component type object files therefore represent most of the cost of transmission and storage. The sizes of the raw machine code and object file for each component type used in our example configuration are given in Fig. 7. The overhead of Lorien's relocatable object files over their raw machine code size, which averages 1.3, is comparable to that reported for Contiki [18]. Overall, Lorien's object file sizes are of the same order as those reported in SOS [5] and Contiki [18]; Lorien's are very slightly larger (averaging 100bytes in our example system) due to the need for component constructors which declare and configure type-safe interfaces.

2) Installation of updates: Once the necessary files have arrived in a node's external flash memory, the software update instructions embedded in an evolution script are passed to REV. The execution of `INS` and `DEL` incur the most overhead here—especially if they need to load or unload a component type. The remaining instructions relate only to interconnection changes and are of negligible cost. The cost of `INS` and `DEL` applied to each component in our example configuration is shown in Fig. 7 (rightmost 4 columns); the cost of replacement is simple the two combined. The cost includes loading (or unloading) a component type, loading (or unloading) a configuration fragment, and updating the manifest. We measure energy cost in a way similar to both [18] and [5] to enable comparison. Specifically, we measure the cost of writing one 512byte page of program memory on the MSP430 as 0.51mJ.

The cost of `INS` then involves writing the machine code of the to-be-installed object file (the sizes of which are given in Fig. 7) into the necessary number of sequential program memory pages and processing any relocations specified. On top of this we add a fixed overhead of $0.51 * 8 = 4.08$mJ to cover the costs associated with allocating program memory blocks and writing the configuration fragment and updated manifest to program memory. This breaks down as 2 page writes on average for each program memory block allocation (including maintaining the linked list of memory blocks), of which we need one allocation each for the object file, configuration fragment and updated manifest resulting in 6 page writes. We then need to write the configuration fragment to its allocated program memory block (these are all <1 page in size) and the updated manifest (also always <1 page)[1]. `DEL` has a fixed cost of $0.51*3 = 1.53$mJ as it involves 1 page write each to free the object file and configuration fragment memory blocks, and the write of 1 page to update the manifest.

[1]Note that even if we only modify part of a page, we must write the entire page to preserve the data in the given page around that which we are writing, since a page of flash memory must first be erased before it can be written.

System Element	Total size in ROM (bytes)	Amount of ROM taken by constructor & destructor (bytes)
Lorien Fixed Kernel	5632	-
REV	**7840**	738
LOADER	**1808**	178
MASSSTORAGE	**5046**	276
ROMFS	**1880**	172
FILEPROP	**3506**	314
RECON	**1714**	292
RADIO	3820	376
MAC	2454	504
TIMER	1556	288
SHT11	788	88
SCHEDULER	1108	174
RANDOM	334	98
SENSEAPP	396	200
Configuration fragments + manifest	1748	-
Total	39630	-

Fig. 8. Makeup of the program memory image of the system in Fig. 1.

The average cost of `INS` in Lorien is of the same order as a dynamic module addition in Contiki and SOS. Adding a simple application in the low-safety Contiki approach for example is reported as costing 2.9mJ [18] compared with 5.5mJ for SENSEAPP in Lorien, the additional overhead being due to Lorien's use of constructors and the need to update configuration fragments and the system manifest. To put this into perspective, [18] reports that receiving 1000bytes of data by radio costs 4.8mJ of energy, making Lorien's update cost very low compared with normal operating costs. The cost of `DEL`, and its equivalents, is negligible in all three approaches.

B. Memory overheads

We measure memory costs in terms of both program memory ('ROM') and RAM. The TelosB platform incorporates 48KB of the former and 10KB of the latter.

The ROM occupancy of Lorien has a fixed cost and variable costs, both of which are shown in Fig. 8. We define the fixed cost as 21,794bytes—i.e. the ROM overhead of those components that support update dissemination and installation. These elements are shown in bold in Fig. 8. On top of this, there are variable costs which scale with the number of loaded component types and instances. These variable costs are incurred i) by constructors/destructors, and ii) by configuration fragments and the system manifest. In our example configuration, the overhead of the former is 3,698bytes, at an average of 284bytes per component type; and the overhead of the latter is 1,748bytes, at an average cost of 134bytes per instance. The total variable cost is therefore 5,260bytes.

In terms of RAM all overhead is variable: our example configuration uses 3,500bytes of RAM, of which 2,346bytes are costs of our modular design (the remainder being taken by radio buffers etc.). These costs include per-component instance meta-data such as interfaces and connections, and the list of loaded types, and average 170bytes per component instance.

By comparison, the fixed ROM cost of SOS's software update support is 20,464bytes [5] on the Mica2 platform, whereas that of Contiki is 25,506bytes on the TelosB platform. Lorien's fixed software update cost at 21,794bytes therefore compares well; we further note that in Lorien all of this functionality can be updated like any other, in contrast to SOS and Contiki. The RAM cost of Lorien is less significant given its small size, taking just 34% of the TelosB's available RAM.

C. Disruption, Safety and Scope

We evaluate these remaining three criteria analytically.

In terms of *disruption*, components can in theory independently arrive and depart a Lorien system while surrounding software stays online. In reality however our runtime integrity model causes selected components to be halted while their dependencies (direct or indirect) are being updated. Examining Fig. 1 we can deduce that the average number of halted components per replacement is 2.5. However, the addition and removal of 'application' components (those which have no dependencies) causes no disruption at all. This matches Contiki at the application level (Contiki anyway has no support for changing software below the application level); and is also essentially equivalent to SOS in which modules simply stop receiving events from modules being replaced.

In terms of *safety*, despite its modular design amenable to high-efficiency low-disruption updates, Lorien assures both compile-time and composition-time safety. It supports compiler-verification of the type-compliance of interface usage by each individual component; and further at composition time guarantees that two components may only interact via a common interface if that interface shares the same hash-code derived at compile-time from its complete type. This significantly improves over both Contiki, which uses weak typing for application/kernel symbol linkage, and SOS, which uses weak typing via events for inter-module interaction.

Finally, in terms of *scope*, Lorien permits 42.5KB of the TelosB's 48KB of program memory, or 88.5% of node software, to be updated. This compares with 50% of node software amenable to updates under Contiki and SOS on the same platform (see discussion of fixed memory costs in Sec. IV-B).

V. Conclusion

In this paper we have examined software update solutions for WSNs, demonstrating that existing work either exclusively offers wide scope and high safety (e.g. TinyOS-Deluge), or else high efficiency and low disruption (e.g. SOS). In response we have presented an approach that satisfies all four criteria based on our modular Lorien operating environment. High efficiency and low disruption are achieved by enabling components to be individually added to / removed from a node's running software; high safety by enforcing component interaction by type-safe interfaces checked at both compile-time and composition-time; and wide scope by applying our model uniformly down through low-level code that can be updated in exactly the same way as application-level code.

We have compared our approach to existing modular solutions demonstrating that additional safety and scope is offered at a very low cost that remains well within the capabilities of current generation low-power hardware such as the TelosB. We believe this paves the way to high-efficiency yet safe software updates for future large-scale production-class WSN deployments which will demand strong correctness guarantees in an evolving software landscape.

References

[1] J. Bernat, "SmartSantander: The path towards the smart city vision," in *Proceedings of the 1st ETSI M2M Workshop*, 2010.

[2] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *SenSys '04: Proc. 2nd international conference on Embedded networked sensor systems*. ACM, 2004, pp. 81–94.

[3] 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*, February 2006, pp. 212–227.

[4] 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*. IEEE Computer Society, 2004, pp. 455–462.

[5] 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*, Seattle, Washington, USA, June 2005, pp. 163–176.

[6] C.-C. Han, R. Kumar, R. Shea, and M. Srivastava, "Sensor network software update management: a survey," *Int. Journal of Network Management*, vol. 15, pp. 283–294, July 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*, vol. 34, no. 5, pp. 93–104, 2000.

[8] N. Reijers and K. Langendoen, "Efficient code distribution in wireless sensor networks," in *2nd ACM International Conference on Wireless sensor networks and applications*, 2003, pp. 60–67.

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

[10] K. Klues, C.-J. M. Liang, J. Paek, R. Musăloiu-E, P. Levis, A. Terzis, and R. Govindan, "Tosthreads: thread-safe and non-invasive preemption in tinyos," in *SenSys '09: Proc. 7th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2009, pp. 127–140.

[11] M. Strübe, R. Kapitza, K. Stengel, M. Daum, and F. Dressler, "Stateful Mobile Modules for Sensor Networks," in *6th IEEE/ACM International Conference on Distributed Computing in Sensor Systems (DCOSS 2010)*, vol. LNCS 6131. Santa Barbara, CA: Springer, June 2010, pp. 63–76.

[12] 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*, vol. 10, no. 4, pp. 563–579, 2005.

[13] D. Hughes, K. Thoelen, W. Horré, N. Matthys, J. D. Cid, S. Michiels, C. Huygens, and W. Joosen, "Looci: a loosely-coupled component infrastructure for networked embedded systems," in *Proceedings of the 7th International Conference on Advances in Mobile Computing and Multimedia*, ser. MoMM '09. ACM, 2009, pp. 195–203.

[14] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," *Operating Systems Review*, vol. 36, no. 5, pp. 85–95, 2002.

[15] N. Brouwers, K. Langendoen, and P. Corke, "Darjeeling, a feature-rich vm for the resource poor," in *SenSys '09: Proc. 7th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2009, pp. 169–182.

[16] D. Gay, P. Levis, R. R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesc language: A holistic approach to networked embedded systems," *SIGPLAN Notices*, vol. 38, no. 5, pp. 1–11, 2003.

[17] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: simplifying event-driven programming of memory-constrained embedded systems," in *SenSys '06: Proc. of the 4th international conference on Embedded networked sensor systems*. ACM, 2006, pp. 29–42.

[18] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-time dynamic linking for reprogramming wireless sensor networks," in *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, November 2006, pp. 15–28.

[19] B. Porter and G. Coulson, "Lorien: A pure dynamic component-based operating system for wireless sensor networks," in *MidSens '09: Proc. of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*. ACM, 2009, pp. 7–12.