Stateful Mobile Modules for Sensor Networks

Moritz Strübe, Rüdiger Kapitza, Klaus Stengel, Michael Daum, and Falko Dressler

Dept. of Computer Science, Friedrich-Alexander University Erlangen-Nuremberg, Germany
{struebe, rrkapitz, stengel, daum, dressler}@cs.fau.de

Abstract. Most sensor network applications are dominated by the acquisition of sensor values. Due to energy limitations and high energy costs of communication, in-network processing has been proposed as a means to reduce data transfers. As application demands may change over time and nodes run low on energy, get overloaded, or simply face debasing communication capabilities, runtime adaptation is required. In either case, it is useful to be able to migrate computations between neighboring nodes without losing runtime state that might be costly or even impossible to recompute. We propose *stateful mobile modules* as a basic infrastructure building block to improve adaptiveness and robustness of in-network processing applications. Stateful mobile modules are binary modules linked on the node itself. Even more importantly, they can be transparently migrated from one node to another, thereby keeping statically as well as dynamically allocated memory. This is achieved by an optimized binary format, a memory-efficient linking process and an advanced programming support.

1 Introduction

A large fraction of Wireless Sensor Network (WSN) applications target long-term monitoring of environmental conditions. Typical examples are monitoring of trees, volcanoes, glaciers, and buildings [1]. All these applications collect various sensor values and transport them to more powerful gateway nodes at the edge of the sensor network. Energy is commonly the limiting factor of long-term monitoring experiments in the context of WSNs. Therefore, reducing communication, which is one of the most energy-intensive tasks in this domain, is crucial. *In-network processing*, the pre-processing of sensor data inside the network is a powerful technique to significantly reduce the amount of data to be transferred [2, 3]. However, in many scenarios the optimal pre-processing has to be determined at runtime. Furthermore, nodes in this domain can run low on energy, get overloaded, or face debasing network conditions. In all these cases the relocation of pre-processing operators is a basic building block to continue service provisioning. Especially challenging in the context of in-network processing is the demand to keep application state, despite relocation. Otherwise, the result is data loss, which can cause blind spots in monitoring experiments decreasing the overall data quality and in the worst case losing important events thereby rendering them useless. Even if it is possible to replace lost data, this can take a considerable amount of time and resources, e.g., using a *pause-drain-resume* strategy [4].

In order to address the aforementioned demands for adaptability and to minimize data loss, resource-efficient system support has to be provided that enables the dynamic

deployment and migration of applications in a state preserving manner. However, even the most basic task to fulfill the goal of dynamic stateful migration, the software deployment, is cumbersome in sensor networks. Code has to be transferred to target nodes, requiring non-negligible communication and energy efforts. As a consequence, several research activities targeted to provide mechanisms and infrastructures to efficiently deploy software in WSNs at the level of system images [5], modules (pre-linked or linked at runtime) [6–8], and byte code [9, 10]. Only, a limited number of systems thereby preserve the execution state of updated code [7, 11], and they all fall short on migration support. Thus, application developers have to do this manually by providing custom serialization routines [12] or rely on a high level byte code language, which has the drawback of a resource-intensive interpretation and a considerable overhead due to the required runtime environment [13].

Taking these facts into account, we propose the concept of *stateful mobile modules*. It enables dynamic migration of stateful, native modules inside a WSN. This is achieved by combining a set of techniques starting with a size-optimized binary format and a memory-efficient linking process. The latter provides the freedom to deploy the same native code on multiple nodes and migrate code inside a WSN as needed. To enable transparent migration of in-memory module state, we provide a programming model similar to high-level languages, such as Java and C#, for supporting serialization. All relevant statically allocated variables that have to survive a migration are marked in the source code. For dynamically allocated memory and pointer variables therein, additional actions are needed. Here, we use a smart-pointer approach provided by an easy to use API. We implemented *stateful mobile modules* and the associated programming model as a resource-efficient layer on top of the Contiki Operating System [14] and evaluated its benefits in a realistic in-network processing scenario.

In the remainder of the paper, we first outline an introductory application scenario. Sections 3 and 4 introduce our system support for resource-efficient linking and for runtime migration. Next, we detail evaluation results (Section 5) and briefly summarize related approaches (Section 6). Finally, Section 7 concludes the paper.

2 Overview

In the following, we outline a data stream processing example, which represents a typical use case for stateful mobile modules. Next, we summarizes the derived goals that we took into account for building the proposed system support.

2.1 Environmental Monitoring Stream Processing Example

Fig. 1 depicts a distributed stream processing query targeting the long-term monitoring of microclimate changes on a rock. The query is composed of a set of connected stream operators distributed over seven nodes: one taking the role of a gateway to the sensor network and six additional nodes that build the actual WSN. Besides receiving data from the network, a server connected to the gateway controls the placement and the wiring of the stream processing operators. These operators are structured as stateful mobile modules so they can be dynamically distributed.

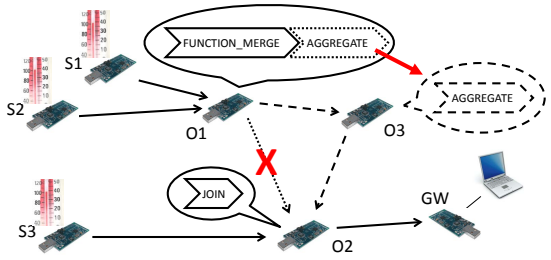

Fig. 1. Migrating of the AGGREGATE operator due to system resource shortage at its current node

In our example scenario, the outer left nodes create three streams each providing temperature data: S1, S2, and S3. The distributed query creates outputs values if the temperature of S3 provided by a sensor placed near the ground is lower than in the area of S1 and S2, both located on top of the rock. A JOIN operator (O1) delivers these items to a sensor node that is connected to the gateway (GW). In our scenario the values of S1 and S2 might be erroneous due to isolation (e.g., only one sensor is exposed to direct sunlight), so we implemented a simple way of sensor data cleaning by using a minimum function provided by FUNCTION_MERGE and an AGGREGATE operator for smoothing outliers (e.g.,cause by clouds). Both are placed on an intermediate node on the stream path (O1).

In the following, the migration of the AGGREGATE operator instance is described. It calculates the mean of a configurable number of samples. For example, due to energy reasons and debasing communication, the central server decides to integrate a neighboring node (O3) into the distributed query by initiating the migration of this operator. This includes transferring the module and its state, and rerouting the data flow. Furthermore, the new node has to receive the results of FUNCTION_MERGE and to deliver the results to the host of the JOIN operator. Other scenarios might include the migration of FUNCTION_MERGE or relocation of the JOIN operator. In all these cases stateful mobile modules enable a code and run-time state migration that is transparent from an operator's point of view.

2.2 Goals

From the described scenario and targeted more complex ones [15], we derived the following goals and requirements for providing stateful mobile modules:

Distribution of modules. Modules generated on a host outside the WSN can be sent to one or more sensor nodes. Furthermore, module code can be shared among nodes by direct exchange. Modules should only require minimal runtime support besides the Operating System (OS) and need to be provided in a space-efficient format. The former avoids overhead during execution, e.g., opposed to a byte-code-based approach, the latter targets low communication costs.

Linking, loading, and running of modules. Consequently, modules should be linked on the node. This allows the use of the same module on nodes with slightly different kernels, e.g., different minor versions or supported hardware. Additionally, the

linking process should be memory-efficient and fast. The former leaves more space for applications the latter enables faster integration and therefore implicitly aims at reducing energy demand.

Migration of modules. It should be possible to migrate a module to a new node with minimal disruption, thereby preserving its execution state. This means that in-memory data, including static variables as well as dynamically allocated memory, is automatically copied to the new node and can be utilized right away. The rationale behind this requirement is not to lose costly computed state information, e.g., gained by long-term monitoring, to relief application developers from the burden to provide custom operations to preserving data, and, lastly, to keep services permanently available.

3 System Support for Mobile Modules

In the following, we detail our support for resource-efficient distribution of native modules using our custom object format (*Minilink*) and a memory-optimized node-level linking process.

3.1 Background: Linking and Loading in WSNs

When compiling a C/C++ file into an object file, the compiler translates the source code into binary code whereas the linker is responsible for the actual memory layout making the code ready for execution. Thus, the compiler writes placeholders into the code and adds an entry for each variable and function to the *relocation table*. Further on, all functions and variables that might be externally accessed are added to the *symbol table*. Next, the linker uses the symbol tables and the target memory location of the code to resolve all references in the relocation table for substituting the placeholders.

For adding a new module to a sensor node, it must be placed in memory and linked against the functions provided by the kernel. If both the modules and the kernel are known in advance, this can be performed at a different machine outside the WSN (*pre-linking*) [7]. However, even slight differences, such as the use of different compiler or linker versions, can cause incompatible modules. Additionally, the placement of modules is fixed which can lead to collisions due to the limited available memory if further modules are added over time.

Alternatively, one can link on the node itself (*runtime-linking*), which has been proven an effective way of distributing code in a network with slightly heterogeneous kernels [8]. Dunkels et al. implemented a linker for the Executable and Linking Format (ELF) format, which they identified as too resource consuming for sensor nodes (see also Section 5). For this reason, they increased the efficiency by introducing Compact ELF (CELF), a custom ELF inspired binary format that is tailored to a 16-bit address space instead of 32 bit. However, a small code size is only one aspect that has to be taken care of, because the symbol table of a typical sensor OS kernel is several kilobytes and the linking process requires random access to the symbol table and the linked module. Whereas the first aspect substantially reduces the available memory, the second leads to a time and energy consuming linking process. This is even worsen by the fact that an ELF binary is subdivided into multiple sections, e.g., different program section like code (`.text`)

and variables (`.data`). This design was made for flexibility and is not suitable for resource-restricted systems that rely on flash memory. First, flash can typically only be modified at the granularity of segments and, secondly, it is usually not possible to buffer the whole program section of a module in RAM. Due to the use of multiple sections, this causes a lot of costly random access during the linking process of a module that usually has to reside on a slow external flash.

We address both problematic aspects of runtime-linking by an optimized symbol table and a further compacted binary format. In combination, this enables an efficient linking process building the essential basis for supporting dynamic migratable modules.

3.2 Resource-efficient Linking using Minilink

In the following, we describe the different aspects of Minilink.

Placement of the Symbol Table. The symbol table of our implementation basis, the Contiki OS kernel, occupies about 5 to $6\,\mathrm{KB}$ of memory. This is rather large compared to the $48\,\mathrm{KB}$ internal flash of the TelosB[1] node, the platform we used for evaluations. However, most sensor nodes are equipped with external storage. The TelosB platform, for example, has an external flash of $1\,\mathrm{MB}$, dedicated to store data. As the symbol table is only accessed during the process of linking, it can be placed on the external flash. The latter saves valuable internal memory for running applications.

Optimizing the Symbol Table. Many functions provided by the kernel have a common name prefix to indicate their relation to a module, e.g., `eeprom_read` and `eeprom_write`. As a consequence, we order the symbol table alphabetically and instead of repeating a matching prefix, we store the size of the common prefix and the remainder of the function name. The symbol table in Fig. 2 shows an example for the `eeprom` function set. Each of the three functions starts with `eeprom_`. The first entry shares no characters with its previous entry, the following two the leading 7. Thus, $11\,\mathrm{byte}$ can be saved (14 saved by compression, 3 lost for indicating the prefix size).

In addition, we take advantage of the fact that symbol names are encoded in the 7-bit ASCII character set [16] by using the unused last bit to terminate them. This saves one byte per table entry. Finally, we exploit that symbols are sorted by name and, therefore, symbols of the same module are in consecutive order. Accordingly, they are also co-located within the code and can, under some checked precondition, be addressed relative to the previous symbol. This demands for only one byte instead of two bytes for the absolute address.

Stream-based Sequential Linking. Due the outlined memory-intensive linking process of ELF binaries, we propose a *stream-based sequential linking* approach, meaning that each byte needs only to be read once. Our linking process works in two stages: First, an *address index* is built. Secondly, the different sections are linked to their destination in a sequential order. To achieve this, our Minilink format structures modules in three sections: A header containing general information about the module, an alphabetically sorted and compressed list of used symbols, and the binary data itself. In contrast to the ELF file format, we do not have a relocation table as the relocation entries are directly woven into the binary data. The header contains only information

[1] The Xbow TelosB was formerly sold by moteiv under the name Tmote Sky.

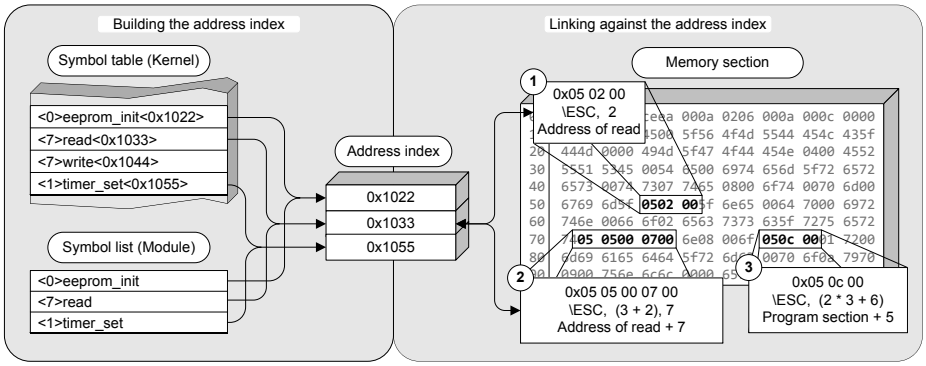

Fig. 2. Mapping used for relocation

that is essentially needed to link the module: the size of each section, the module name, the entry point, and the number of required external symbols. Based on this information, the linker ensures that the node offers enough memory to link the module.

Building the Symbol Index. A module starts with a list of symbols required for linking. They are saved applying the same concepts as used for compacting the symbol table but miss address information. The latter is provided by matching the module symbols against the symbol table. As a result, we get the address index that contains all resolved symbol addresses of a module and builds the input for the linking phase (see Fig. 2). As both lists are ordered alphabetically they can be processed in a sequential order. Together with the module header, the address index is the only data that has to reside in RAM during this process.

Linking Using the Symbol Index Using Minilink, all relocation entries are directly woven into the binary data. To identify them, these two bytes large entries are marked by a preceding escape sequence. We chose the escape sequence to be `h05` as this is neither a binary command on the TelosB platform nor is the escape-character itself commonly used in ASCII strings. Still `h05` can be encoded by `h050000`. The access to the address index starts with one, as zero is already reserved.

Fig. 2, (1) shows the escape sequence for the second symbol. Accordingly, the sequence `h050200` is replaced by the second value of the address index `h3310` that further on is written to memory. Sometimes, not only a symbol is referenced by a relocation entry, but additionally an offset is added, e.g., when accessing an element of a struct or array. For these cases, we reference the address index with an offset, and this time the value of the word following is added (2). Higher escaped values map to the different sections of the module containing the program code or variables (3).

4 Stateful Migration

In the context of in-network data processing, the migration of state in terms of statically and dynamically allocated variables is important as these often capture long-term execution results. In contrast, the execution stack representing the call history is of minor importance as applications in this domain are rather small and moderately complex.

Furthermore, transferring the stack would introduce additional costs in terms of data transfer. Therefore, we support *weak migration* [17], meaning only application state is transferred but no execution-dependent state, such as values on the stack and CPU registers. This is also in line with the lightweight thread model of Contiki. Here, so-called Protothreads [18] lose their stack and register values when yielding the CPU in favor of another thread. However, once a Protothread is resumed, it still continues execution at the same position the CPU was released. Accordingly, we keep this behavior by restarting an application at the same point where it was suspended before migration. Thus, from an application programmer's perspective, releasing the CPU for another application and migrating to a different node are equivalent: in both cases, the stack is lost and the execution is continued immediately after the last executed statement.

In contrast to the execution stack, the handling of pointers still needs special attention when migrating modules using Protothreads. To keep them valid despite migration, one option would be to place data at the same memory address. Due to the limited available resources and the use of multiple modules as well as the absence of a Memory Management Unit, this is not practical for sensor nodes. However, if variables are dynamically allocated there is no way to avoid the use of pointers. Therefore, mechanisms must be provided to properly access the data despite relocation. If the placement of pointers is known, they can be adjusted to the new memory layout. This reduces the burden for the developer to the necessary minimum.

4.1 General Process and Programming Model

In general, it is not reasonable to migrate all statically and dynamically allocated variables. If values can be easily recomputed or are dispensable to provide a service, they should be excluded from migration. We therefore provide two macros (`MIGRATABLE` and `MIGRATABLE_POINTER`), which assign a *section attribute* to the variable. This attribute instructs the compiler to put a variable in a special memory section. These extra memory sections are supported by our linker and handled separately. During migration only those two sections are copied to the target node, and the one containing the pointers is adjusted to the new memory layout.

Frequently, dynamic memory is used, which is allocated at run-time from heap space. To support the migration of dynamically allocated memory, we built two wrapper functions (`migmem_malloc` and `migmem_free`). Two extra bytes are used to add the newly allocated memory block to a linked list, which is managed by our framework and assigned to the module. Finally, when implementing a linked list and similar complex dynamic data structures, it is very likely that pointers reside inside the dynamically allocated memory. We provide a function (`migmem_register`) to make pointers placed in heap memory known to our framework. It saves the address of the pointer in a special array. The memory for the array is also taken from heap memory and dynamically adjusted in size. The list of registered pointers is transmitted and the pointers adjusted during the migration process. Of course, it is also possible to "unregister" a pointer. Furthermore, pointers are automatically removed from the list when freeing memory containing a registered pointer.

Preparation. Although we are able to migrate variables and execution state, it is not possible to migrate state that is directly bound to the node itself, such as a network

connection and a file handle. For this reason, a module is informed by a MIG_REQUEST event that it is about to be migrated. It then has the option to take appropriate actions, e.g., to close open sockets, before it is moved to a new node. In the case of ongoing communication with external hardware, the module is able to postpone migration by calling mig_delay(). It will automatically get a new migration request a few seconds later. The module can also deny migration by calling mig_deny().

Migration. Before the actual migration, the module is linked, but not started on the target node. The linker already allocates both memory sections containing the data and pointers. The source node serializes all memory blocks and also transmits the old memory address to the target. The static sections are copied to the memory allocated by the linker while memory for the other blocks is allocated from the heap memory. The old addresses are used to build a lookup table to map the addresses of the source node to the target node. Using this lookup table, the pointers in the pointer section are adjusted. In a next step, the list of pointers registered at runtime is transmitted. The lookup table must be used to find their new location, before they can be adjusted. Finally, the state of the Protothread is received and the local thread structure is updated accordingly.

Continuation. After the module is successfully migrated to the new node, it continues to run and receives a MIG_SUCCESS event so it can reestablish its connections and perform other preparations, e.g., initialize variables omitted from migration. The thread on the old node will be terminated and its memory freed. If an error occurred during migration, e.g., as there is not sufficient memory on the target node, the migration is aborted and the module continues to run at the original node. To notify the module that the migration was aborted, it receives a MIG_FAILED event.

4.2 Application Example

To illustrate our programming support, we describe an application example based on the environmental monitoring stream processing example presented in Section 2.1. We concentrate on the AGGREGATE operator as this one is migrated in the scenario.

Fig. 3 shows the simplified listing of the AGGREGATE operator. It takes a number of samples (window), calculates the average, and forwards the result. The number of samples taken to calculate the average can be adjusted at runtime. Therefore, the memory used to save these variables is dynamically allocated. Our data stream framework abstracts the network traffic and sends commands either directly to an operator or broadcasts incoming data to all operators hosted by the node.

In the first three lines, the necessary variables are defined. The in-variable is a pointer and is therefore marked as such. The other two save the window size and the position to write the next incoming data. Lines 5-7 and 23 contain macros generating the structures needed by the Contiki OS to manage the Protothread. In line 11, the operator waits for an incoming event. If the event is a command, the data pointer contains additional data. If the operator is instructed to resize its window size, the old memory is freed (line 16) and a new memory is allocated (line 17). For simplicity of the example, the data is lost upon window resize. As connection handling and all further system-dependent tasks are performed by our framework, there is no need to inform the operator about a migration. Thus, due to migration support of statically as well as dynamically allocated memory, a migration is fully transparent to the operator.

```
1  MIGRATABLE_POINTER static u16 * in;
   MIGRATABLE static u8 pos;
3  MIGRATABLE static u8 window;

5  PROCESS(p_migagg, "Mig.Aggr");
   AUTOSTART_PROCESSES(&p_migagg);
7  PROCESS_THREAD(p_migagg, ev, data)
   {
9    PROCESS_BEGIN();
       while(1) {
11       PROCESS_WAIT_EVENT();                    // Wait for an event
         if(ev == EV_MODULE_CMD) {                // A command event
13         if(*data == MOD_CMD_SIZE) {            // Resize window command
             window = *(++data);                  // Set window size
15           pos = 0;                             // Reset write position
             if(in != NULL) migmem_free(in);      // Free old window
17           in = migmem_alloc(window * 2);       // Allocate new window
           }
19         else if(ev == EV_MODULE_DATA) {        // Handle incoming data
             in[pos++ % window] = *(u16 *)data;   // Copy data
21           // Calculate average and send it
       } } }
23     PROCESS_END();
   }
```

Fig. 3. Simplified listing of the data stream `AGGREGATE` module

5 Evaluation

While our implementation runs on the native TelosB hardware, we performed most of our evaluations on top of the Cooja [19] simulator. Cooja utilizes the MSPsim simulator [20], an instruction level simulator for the MSP430 micro controller. Thus, code can be added and executed at runtime. For our evaluation, we considered the basic characteristics of our binary format, the linking process and the support for migrating stateful modules.

5.1 Support for Linking and Loading

Overall Resource Demand. Our system support for sending, receiving, linking, starting and stopping modules as well as migration is 7 KB in size and has a memory footprint of 160 B. It also includes helper functions such as for remote monitoring (e.g., listing the currently installed modules) as well as commands for managing the external flash.

Symbol Table Footprint. To measure the memory savings provided by our compression of the symbol table we analyzed the symbols of a "hello world" kernel using the default kernel for the TelosB platform, having 316 symbols in total (see Table 1).

The Contiki ELF-Linker stores all kernel symbols in an array containing the address and a pointer to the string. While this improves performance when searching for a certain symbol, we omit this additional pointer, which saves two bytes per symbol. The reason why the use of the Minilink format reduces the size not by 623 B (symbols $\times$ 2), but only 573 B, is because of our slightly larger header and 4 symbols[2] that are excluded in the Contiki symbol list. The largest savings are achieved using our simple prefix-compression (`prefix`). As the character set for symbols is limited to 7 bit ASCII we

[2] `__bss_size`, `__data_load_start`, `__data_size` and `__stack`

	Size	Saved (relative)	Saved (total)
Contiki	5732 B		
Minilink	5159 B	10.00 %	10.00 %
Minilink+option(`prefix`)	3399 B	34.12 %	40.70 %
Minilink+option(`prefix,7bit`)	3083 B	9.30 %	46.21 %
Minilink+option(`prefix,7bit,offset`)	2918 B	5.35 %	49.09 %

Table 1. Comparison of the Contiki symbol table and the Minilink symbol table.

Application	**hello_world**	**mod_agg**	**mod_agg + rudolph2**
code size	74 B	910 B	2230 B
symbols	1	15	36
relocations	5	48	145
ELF (overhead)	752 B (+916 %)	2956 B (+225 %)	6028 B (+170 %)
CELF (overhead)	179 B (+142 %)	1611 B (+77 %)	3793 B (+70 %)
Minilink (overhead)	96 B (+30 %)	1164 B (+28 %)	2772 B (+24 %)

Table 2. Comparison of ELF modules and Minilink modules for modules of three different sizes.

can use the 8th bit to substitute the NULL-terminator (`7 bit`). Finally, we are able to save some extra bytes by using relative addressing where possible (`offset`). In sum, our symbol table is almost half the size of the Contiki symbol table. This speeds up the linking process as only half the data must be transferred from the external flash. These savings are even more valuable if the symbol table must be saved together with the kernel on internal flash, e.g., if there is no external flash available.

Minilink Module Footprint. We also compared the size of ELF modules provided by Contiki and their optimized variant CELF with our Minilink format (see Table 2). As a first sample, we chose a *hello_world* module, which basically outputs a "Hello World" string. The next sample is a module implementing the *AGGREGATE* operator, as outlined in the stream processing example (*mod_agg*). For evaluating a larger module we also linked the *rudolph2* (and polite) network protocol [21] to the *AGGREGATE* operator as one module. Further on we added a call to `watchdog_periodic()` to this module to evaluate the symbol table lookup performance (see following paragraph on Minilink linker performance for details).

The code size represents the size of the program and the data section (not `.bss`). This is the minimum size even a prelinked module must have without its header. We also list the number of used symbols and relocations. The numbers in braces represent the overhead compared to the minimum prelinked module. Apparently the ELF has a big static overhead. This has a huge impact on small modules, as can be seen for the `hello_world` example, with an overhead of over 900 %. While CELF performs much better, Minilink has an overhead of only about 30 %.

Minilink Linker Performance. Table 3 compares the time required to link a module using the previously introduced examples. We measured the duration to build up the address index and the linking process independently. Contiki OS provides the Coffee file system, which has a high standard deviation when opening file handles (132 ± 36.6ms according to Tsiftes et al. [22]) therefore we excluded these from our measurements.

However, the time to access the actual data on the external flash is included. When calculating the percentage of relocations, it has to be taken into account that each relocation results in a memory address and is therefore two bytes in size. It can be seen, that the time to build up the address index does not entirely depend on the number of different symbols, but where the last used symbol is located in the symbol table. The reason for this, are the sorted symbols. Once a symbol is found, the next symbol in the module must be further down the symbol table and there is no need to restart the search. The linking process itself scales with the size of the module. This is mainly due to the slow access to flash. Half the time is spent writing to internal flash.

5.2 Stateful Migration

The migration support is responsible for 2 of the 7 KB needed for the module and migration support. It includes all functions required for the serialization of modules.

We analyzed the migration of the *mod_aggr*-module. It has 8 B of data, one pointer and 10 B allocated from heap, which totals in 20 byte that must be migrated. The size of the serialized data was 44 byte. This is mainly caused by static overhead, like the individual sizes and addresses of the different memory sections (6×4 B), the total number of dynamically allocated memory blocks and registered pointers. The only additional overhead are 4 byte (address and size) for every additional dynamically allocated memory block.

A single-hop migration takes about 1 s. Almost all of the time is spent due to the network stack: To save power, each node has a short listening period every 300 ms. Also, the route is not previously known to the mesh network stack and has to be found first. Even under optimal conditions (route known, and the target listening), the migration would take at least 19 ms. Thereby, less then 2 ms is caused by our migration support. The serialization itself takes about 0.4 ms and the deserialization – including the allocation of additional memory and adjustment of the pointer – takes 0.6 ms.

We also compared our approach to the one suggested by [12]: Their framework provides a buffer and the module does the serialization by itself. We implement a very simple serialization using `memcopy()` and all variables placed in a struct without any error handling. This approach adds another 183 B to the *mod_aggr* Minilink module (+208 B in the case of ELF).

Module	**hello_world**	**mod_agg**	**mod_agg + rudolph2**
Symbols	1	14	27
Last symb.	puts	rimeaddr_copy	watchdog_periodic
Match symb.	55 ms	82 ms	114 ms
Size (text + data)	78 B	910 B	2230 B
Relocations	5 (12.8 %)	48 (10.6 %)	119 (10.7 %)
Link	5 ms	67 ms	159 ms
Bytes/ms	14.4	13.6	14.0

Table 3. Comparison of the time to build the address index and linking for different modules.

6 Related Work

We approach related work by examining how code modules are deployed and updated in WSNs and then investigate systems supporting state migration.

Loading Code at Runtime. Using a Virtual Machine (VM) is one of the most flexible solutions for code deployment as it typically employs some from of high-level byte code that is less machine and OS dependent than plain native code. Furthermore, due to its higher expressiveness, the code size of a module is usually smaller and consequently less data needs to be transferred. In consequence, several VMs are available for sensor networks, e.g., Maté [9] and VM* [23]. In [8], VMs are compared against runtime linking of native modules, showing that installing a module using a VM can be more efficient than linking in terms of integration time, but offers less performance at runtime. Our work reduces the downsides of runtime linking while retaining the performance offered by native execution.

SOS [6] is one of the first WSN-class OSs that has native support for modules. Thus, single modules can be added without affecting the rest of the system. A limited number of kernel functions are exported using a jump table. All other function addresses are resolved at runtime using string comparison and a lookup table. Once resolved, function addresses are cached to speed up calling functions. Still, this entails a runtime overhead for every function call. Further on, all strings that are needed to resolve function calls must be saved in the internal flash.

Dunkels et al. [8] have built a runtime linker for ELF-based modules on top of the sensor OS Contiki. However, the ELF binary format is rather resource consuming. The symbol table for a standard Contiki kernel requires about $5\,\mathrm{KB}$ of flash memory and an ELF-module is normally more then twice the size of the resulting binary file. Accounting these facts, Dunkels et al. proposed an ELF inspired binary format called CELF, which is optimized for 16 bit microcontrollers. Dong et al. [24] also proposed an optimizing the ELF format that kept the basic design. However, they pre-fill the placeholders of a module with concrete addresses. This combines the advantages of pre- and runtime linking: If the correct kernel is installed, the module can be copied without linking; otherwise, the module can still be linked.

Based on the BTnut[3] we have built a host-based linker, which generates binary modules that are custom-tailored for a remote node [7]. This solution misses the flexibility provided by a node-based linker. However, only the final code has to be transferred, which further on can be directly executed. In sum, none of the presented approaches offers such a highly optimized binary layout for native modules like Minilink and a tailored memory-efficient linking process.

Stateful Migration. Chien-Liang et al. [13] proposed system support enabling the mobile agent paradigm in sensor networks. This was achieved using a VM concept with a dedicated instruction set that is based on Maté [9]. This way, code can be immediately executed; and it is portable so it can be shared between different nodes. However, this higher layer of abstraction adds a considerable run time overhead.

The closest related work to our knowledge is UbiMASS [12]. UbiMASS is a framework for Contiki OS supporting migration of agents at runtime – something,

[3] http://www.btnode.ethz.ch/

that can easily implemented with our approach. We believe our approach is superior in three main aspects: Firstly, the Contiki linker, which's file format we have proven inefficient, is used. Secondly, the developer of the agent must do the serialization of data. Meaning, instead of just marking each variable as migratable it must be passed to special function and recovered manually after migration. Thirdly, UbiMASS only supports the migration of integer and string variables, while we can migrate any data type, and even pointers if there were made known to the framework.

7 Conclusion

In-network processing of sensor data is a powerful technique to enable long-term monitoring using WSNs. However, changing runtime conditions demand for flexible state-preserving relocation of pre-processing modules to enable continuous monitoring. We presented a resource-efficient solution based on stateful mobile modules. At its core, we provide a size-optimized binary format and a memory-efficient linking process together with runtime support to migrate statically allocated variables as well as dynamically allocated memory. Finally, stateful mobile modules can be seen as a novel approach, which give application programmers the convenience to implement migration-enabled stateful applications using a simple API.

References

1. Hart, J.K., Martinez, K.: Environmental sensor networks: A revolution in the earth system science? Earth-Science Reviews **78** (2006) 177–191
2. Gehrke, J., Madden, S.: Query Processing in Sensor Networks. IEEE Pervasive Computing **3**(1) (Jan. – March 2004) 46–55
3. Edara, P., Limaye, A., Ramamritham, K.: Asynchronous in-network prediction: Efficient aggregation in sensor networks. ACM Transactions on Sensor Networks (TOSN) **4**(4) (August 2008) 1–34
4. Zhu, Y., Rundensteiner, E., Heineman, G.: Dynamic Plan Migration for Continuous Queries over Data Streams. In: ACM SIGMOD Conference 2004, Paris, France (June 2004) 431–442
5. Jeong, J., Culler, D.: Incremental Network Programming for Wireless Sensors. In: 1st IEEE International Conference on Sensor and Ad hoc Communications and Networks (IEEE SECON 04), Santa Clara, CA, USA (June 2004)
6. Han, C.C., Kumar, R., Shea, R., Kohler, E., Srivastava, M.: A dynamic operating system for sensor nodes. In: Proceedings of th 3rd ACM International Conference on Mobile Systems, Applications, and Services (ACM MobiSys 2005), Seattle, WA, USA (June 2005) 163–176
7. Dressler, F., Strübe, M., Kapitza, R., Schröder-Preikschat, W.: Dynamic Software Management on BTnode Sensors. In: 4th IEEE/ACM International Conference on Distributed Computing in Sensor Systems (IEEE/ACM DCOSS 2008): IEEE/ACM International Workshop on Sensor Network Engineering (IWSNE 2008), Santorini Island, Greece (June 2008) 9–14
8. Dunkels, A., Finne, N., Eriksson, J., Voigt, T.: Run-time dynamic linking for reprogramming wireless sensor networks. In: 4th ACM Conference on Embedded Networked Sensor Systems (SenSys 2006), Boulder, CO (November 2006) 15–28
9. Levis, P., Culler, D.: Maté: a tiny virtual machine for sensor networks. ACM SIGOPS Operating Systems Review **36**(5) (2002) 85–95

10. Brouwers, N., Langendoen, K., Corke, P.: Darjeeling, a feature-rich vm for the resource poor. In: 7th ACM Conference on Embedded Networked Sensor Systems (SenSys'09), New York, NY, USA, ACM (2009) 169–182
11. Felser, M., Kapitza, R., Kleinöder, J., Schröder-Preikschat, W.: Dynamic Software Update of Resource-Constrained Distributed EmbeddedSystems. In: IFIP International Embedded Systems Symposium (IESS'07). Volume 231/2007., Irvine, CA, USA (May 2007) 387–400
12. Bagci, F., Wolf, J., Ungerer, T., Bagherzadeh, N.: Mobile Agents for Wireless Sensor Networks. In: International Conference on Wireless Networks (ICWN'09), Las Vegas, NV, USA (July 2009)
13. Fok, C.L., Roman, G.C., Lu, C.: Rapid development and flexible deployment of adaptive wireless sensor network applications. In: 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05, Columbus, OH, USA (June 2005) 653–662
14. Dunkels, A., Grönvall, B., Voigt, T.: Contiki - a lightweight and flexible operating system for tiny networked sensors. In: 1st IEEE Workshop on Embedded Networked Sensors (Emnets-I), Tampa, FL (November 2004)
15. Dressler, F., Kapitza, R., Daum, M., Strübe, M., Schröder-Preikschat, W., German, R., Meyer-Wegener, K.: Query Processing and System-Level Support for Runtime-Adaptive Sensor Networks. In: 16. GI/ITG Fachtagung Kommunikation in Verteilten Systemen (KiVS 2009), Kassel, Germany, Springer (March 2009) 55–66
16. TIS Committee: Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification. (May 1995)
17. Fuggetta, A., Picco, G., Vigna, G.: Understanding code mobility. IEEE Transactions on Software Engineering **24**(5) (May 1998) 342–361
18. Dunkels, A., Schmidt, O., Voigt, T., Ali, M.: Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In: 4th ACM Conference on Embedded Networked Sensor Systems (SenSys 2006), Boulder, CO (November 2006)
19. Österlind, F., Dunkels, A., Eriksson, J., Finne, N., Voigt, T.: Cross-level simulation in cooja. In: European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session, Delft, The Netherlands (January 2007)
20. Eriksson, J., Dunkels, A., Finne, N., Österlind, F., Voigt, T.: Mspsim – an extensible simulator for msp430-equipped sensor boards. In: European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session, Delft, The Netherlands (January 2007)
21. Dunkels, A.: Rime — a lightweight layered communication stack for sensor networks. In: European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session, Delft, The Netherlands (January 2007)
22. Tsiftes, N., Dunkels, A., He, Z., Voigt, T.: Enabling Large-Scale Storage in Sensor Networks with the Coffee File System. In: 8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2009), San Francisco, USA (April 2009)
23. Koshy, J., Pandey, R.: Vmstar: synthesizing scalable runtime environments for sensor networks. In: 3rd International Conference on Embedded Networked Sensor Systems (SenSys 05), San Diego, CA, USA (November 2005) 243–254
24. Dong, W., Chen, C., Lie, X., Bu, J., Liu, Y.: Dynamic Linking and Loading in Networked Embedded Systems. In: 6th IEEE International Conference on Mobile Ad Hoc and Sensor Systems 2009 (MASS 2009), Macau SAR (October 2009)