Stateful Mobile Modules for Robust In-network Processing

Moritz Strübe (PhD student), Rüdiger Kapitza, Klaus Stengel (Master student),
Michael Daum (PhD student), Falko Dressler
Department of Computer Science, University of Erlangen-Nuremberg, Germany

***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 valuable 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 linking process and an advanced programming support.

I. 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]. In many scenarios the optimal pre-processing strategy 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 operators performing the pre-processing is an essential 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.

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. Taking these facts into account, we propose the concept of *stateful mobile modules*. It enables dynamic migration of native modules and their associated state inside a WSN. This is achieved by a unique set of measures: Firstly, a programming model supporting the migration of statically as well as dynamically allocated memory. Secondly, an optimized binary format as well as a node-level linking process. Thirdly a migration framework, which is built on top.

II. Providing Statful Mobile Modules

We implemented *stateful mobile modules* and the associated programming model as a resource-efficient layer using the Contiki Operating System (OS) [3] but are not strictly tied to it.

Programming support. Similar to high-level programming languages such as Java, we decided to provide programming directives and extended API support to make important state serializable. Accordingly, variables and pointers that should be preserved during migration have to be tagged using dedicated macros. These macros instruct the compiler to place the variables into special memory sections (variables into one section and pointers into another) being part of a custom object file format. For dynamically allocated memory and pointer variables therein, we support a smart-pointer approach provided by an easy to use API. This is needed to register and manage pointers placed in memory allocated at run-time such as needed for implementing dynamic data structures (e. g., linked lists).

Memory-efficient linking process and optimized binary format. After compilation, the object files are converted into an optimized binary format that is small[1] and can be efficiently linked due to including relocation information. This way, a module can be directly integrated on the target node while memory sections containing variables and pointers are preserved.

Migration. Assuming the code is not already there, a module migration starts with a code transfer and its linking at the target. Next, the source node sends the two memory sections containing variables and pointers, as well as all dynamically allocated memory. It also transmits the old addresses of all sections and the state of the thread executing the module. The two memory sections are then copied to the memory already allocated by the linker. Knowing the previous memory location, the statically allocated pointers can be adjusted to the new memory layout. Finally, the list of registered in-dynamic memory pointers is transmitted and their addresses are adapted.

III. Stream Processing Example and Demo Setup

Fig. 1 depicts a distributed stream processing query targeting the long-term monitoring of micro climate changes on a rock. The query is composed of a set of connected stream operators

[1]The *minilink* format's overhead is less than half the size of CELF [4], which can be considered the state of the art.

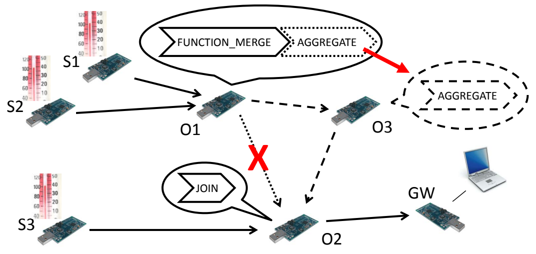

Fig. 1. Migrating an operator module from one sensor node to another

```
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();
     }
```

Listing 1. An example for a data aggregation operator

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 (GW) 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.

Scenario outline. In our demo scenario, we want to compare the temperature on top of a rock (sensor S1, S2) with the temperature beneath (sensor S3). The values of the upper sensors might be erroneous due to insolation (e.g., one sensor is exposed to direct sunlight). Both nodes send their measurements to node O1. Here, the FUNCTION_MERGE operator reduces the data based on a minimum function. Afterwards, the AGGREGATE operator smoothes the result to limit the influence of outliers by calculating the mean of a configurable number of samples. The result is then forwarded to O2 where the value is *joined* with the measurements of the node beneath the rock, and forwarded to the gateway.

Next, the migration of the AGGREGATE operator instance is described. 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. 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.

Programming support. Listing 1 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 lightweight 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 (l. 16) and a new memory is allocated (l. 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.

Preliminary results. Including debugging statements, the aggregate-operator module has the size of $1164\,\mathrm{B}$ and its linking on a TelosB mote takes about $150\,\mathrm{ms}$. Thereby, our binary format provides a much smaller module compared to ELF ($2956\,\mathrm{B}$) and CELF ($1611\,\mathrm{B}$ [4]). The time to migrate a module from one node to another strongly depends on the underlaying communication layer. While the overall migration of the aggregate-operator takes up to $1\,\mathrm{s}$, the actual (de-)serialization of the state accounts only for $1\,\mathrm{ms}$.

IV. Conclusion and Ongoing Work

Stateful mobile modules build a powerful abstraction to improve robustness and adaptivity of in-network sensor processing, enabling the dynamic (re-)deployment of stateful operators. This is achieved by a set of system measurers including advanced programming support, a tailored object format, a memory-efficient linking process and framework support controlling the process of migration. We are currently about to finish our prototype implementation and extensively evaluate its system characteristics. In the course of the EuroSys poster session we will perform a live demo showing the principle functioning of our approach based on the outlined monitoring scenario.

References

[1] J. K. Hart and K. Martinez, "Environmental sensor networks: A revolution in the earth system science?" *Earth-Science Reviews*, vol. 78, pp. 177–191, 2006.

[2] J. Gehrke and S. Madden, "Query Processing in Sensor Networks," *IEEE Pervasive Computing*, vol. 3, no. 1, pp. 46–55, Jan. – March 2004.

[3] A. Dunkels, B. Grnvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *1st IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, FL, Nov. 2004.

[4] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-time dynamic linking for reprogramming wireless sensor networks," in *4th ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, CO, Nov. 2006, pp. 15–28.