# Dynamic Operator Replacement in Sensor Networks

Moritz Strübe*, Michael Daum*, Rüdiger Kapitza*, Felix Villanueva† and Falko Dressler*
*Dept. of Computer Science, University of Erlangen, Germany
†School of Computer Science, University of Castilla-La Mancha, Spain
{struebe,daum,kapitza,dressler}@informatik.uni-erlangen.de, felix.villanueva@uclm.es

***Abstract*—We present an integrated approach for supporting in-network sensor data processing in dynamic and heterogeneous sensor networks. The concept relies on data stream processing techniques that define and optimize the distribution of queries and their operators. We anticipate a high degree of dynamics in the network, which can for example be expected in the case of wildlife monitoring applications. The distribution of operators to individual nodes demands system level capabilities not available in current sensor node operating systems. In particular, we present a system for seamless and on demand operator migration between sensor nodes. Our framework, which we implemented for Contiki running on TelosB nodes, supports stateful module migration including selected parts of the code and data sections.**

## I. Introduction

Wireless Sensor Networks (WSNs) consist of nodes that are widely distributed. In-network data processing is considered as a key methodology to provide robustness and to handle dynamics and heterogeneity in a resource and energy efficient way [1], because, as data volume grows rapidly, the typical central base station processing approach becomes more and more impracticable. Distributed data processing using operators like filtering and aggregation helps to reduce communication in the vicinity of the sensors.

Data stream processing as it provided by Data Stream Management Systems (DSMSs) is also an option for distributed processing in WSNs that is more efficient than sending all data to a central processing unit [2]. If operator placement decisions get disadvantageous at runtime, replacement and migration of operators is required. Some DSMSs like Borealis [3] support operator migration by switching the flow of streams at the appropriate point in time without state migration (*pause-drain-resume* strategy). Without state migration, this is called *pause-draine-resume* strategy in the related literature [4]. Others like CAPE [4] do real state migration. These DSMSs run on full-fledged PCs.

In the scope of our *Resource-constrained Distributed Stream Processing* (RDSP) project,[1] a central server coordinates distributed stream processing and organizes query reorganization if necessary. We decided to apply stateful operator migration as reorganization strategy in the WSN, as we assume transmitting a module and its state to a neighboring node being cheaper than the entire deployment

[1]http://rdsp.informatik.uni-erlangen.de

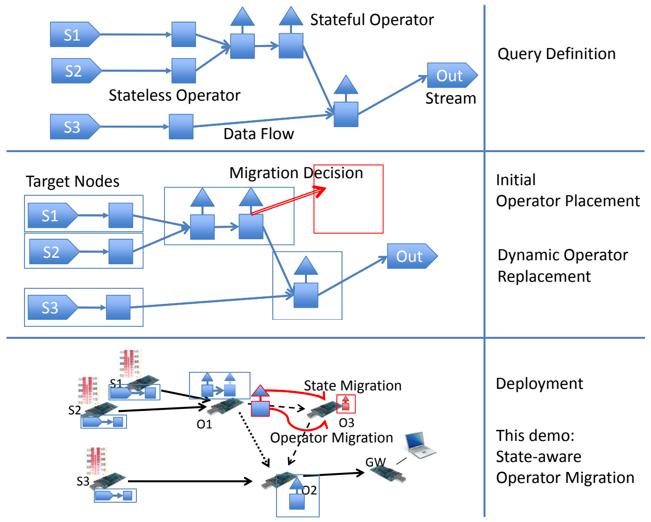

Figure 1. Mapping of queries and stateful operator migration based redeployment: First, the global query is defined. Based on a global optimization of the query into subqueries, the initial operator placement is estimated. The key concept presented in this demo, however, is the on-demand state-aware migration of operators.

process. Most importantly, loss of state might be a more severe problem than arbitrarily deploying new nodes. Figure 1 outlines the distribution of a global query to a subset of nodes within the sensor network.

We rely on Abstract Query Language (AQL) for describing the queries in our framework. However, as the introduction of our AQL is beyond the scope of this paper, the query's notation is a common directed query graph using some stateless operators in the vicinity of the nodes that produce data and a stateful operator that is placed between the stateless operators and the query's data sink. It is useful to place these operators there if they help to reduce communication costs, e.g. filter operators. After these filter operators, there is an aggregating data-fusing operator that is stateful. Aggregation operators help to reduce data rates, too. The placement decision is made upon a cost model that considers all relevant communication costs.

In this introductory example, both filtering and aggregation operators are pushed as near to the data sources as possible. Further, in this example, we assume that it is not possible to place both filter and aggregation to one node. This capacity issue can easily be decided by the central manager of the RDSP project, as it knows both the modules' sizes and the available memory of each node.

In order to support stateful operator migration in the domain of sensor networks, either abstraction layers or virtual machines would be needed, or direct system level support is required for the operator migration, which is clearly better suited for memory-constrained sensor nodes. Among others, popular sensor network operating systems such as Contiki, TinyOS, and SOS support a modular programming model. Depending on the implementation, modules can even be loaded at runtime. However, none of these systems support a programming model for stateful module migration between sensor nodes.

Our presented framework extends our earlier work to support Contiki [5] as a reference system. The theoretic foundations of the system level support are outlined in [6]. We developed this framework explicitly to support dynamic operator replacements in the scope of DSMSs. This demo presents the current state of *seamless operator migration* among TelosB sensor nodes at runtime *including state information* of running threads.

## II. System Architecture

Our system architecture can essentially be split in two different parts: First, a central server that controls and coordinates the placement and inter-connection of operators for in-network stream processing; secondly, a set of sensor nodes forming the actual network that are collecting and preprocessing data from a set of available sensors. The data-stream processing is achieved by providing run-time support for dynamic deployment of native modules that encapsulate stream operators, so-called *operator modules* that are dynamically wired with each other. The initial deployment and successive (re-)deployment steps are coordinated by the central server that can also make adjustments to the topology of the data streams.

From a system perspective, the operator deployment is supported by an optimized node-based runtime linker, thus giving the freedom of re-using the same module on different nodes without restrictions to kernel versions, memory allocation or the need of a reboot. Our deployment protocol contains a set of basic operations to deploy, configure, and start operator modules. In addition, we extended the deployment protocol by operations for migration and provide system-level support to migrate stateful operator modules from one node to another while preserving their state of execution. Such a migration is typically transparent to the operator module and the whole distributed application.[2] This is achieved by system support for re-directing event streams, but more importantly by an advanced programming model and extended execution support.

[2]We assume that the operator module does not access device and system APIs directly. Otherwise, the operator module is notified about the imminent migration and can act appropriately.

Similar to high-level programming languages such as Java, we decided to provide programming directives and extended API support to make modules serializable. In our solution, variables have to be marked, so their value is kept once a module is transferred to a different node. We also provide an API to allocate memory dynamically from heap. This memory is tied to the module and gets copied to the target node upon migration. To allow the use of dynamic memory, not only migration of data, but also the use of pointers must be supported. In the course of migration of stateful operator modules, it cannot be guaranteed that the data is placed at the same physical memory location. While all addresses within the linked code are correctly set by the linker, all pointers pointers copied during the migration process must be adjusted, too. To adjust the pointers, our framework must know the addresses of the pointers. We provide two ways of making pointers known to the framework. Static pointers can be tagged, just like variables, whereas pointers within memory allocated from heap must be registered using our API.

Besides the memory management, the state of the thread itself needs to be transferred. Contiki uses so-called *protothreads*, which are very similar to coroutines and loose the contents of their stack when returning to the operating system. Therefore, only the state of the protothread, which does not contain any pointers or node-specific information, needs to be transferred.

In summary, the migration of an operator is organized as follows: In a first step, the target node must be prepared. This is done by transferring a copy of the module to the node, which is then linked into available memory. Next, the source node is told to migrate the operator to the new node. Before migration, the operator gets a migration event, which allows the operator to cleanup system functions such as timers. Afterwards, the different memory sections are serialized and transferred to the new node including information about their placement on the old node. Finally, the protothread state and the list of registered pointers are transferred. At the target node, the pointer-sections are adjusted first. In a second step, the new placement of the registered pointers is calculated and they are adjusted as well. In a last step, the operators sending data to the module must be rerouted to the new node.

## III. Stream Processing Example

This section outlines the idea of stream processing based in-network data operation; furthermore, the dynamic operator migration is demonstrated. The module to be migrated contains stateful operators that are part of a demo query outlined in Figure 1.

```
 1 MIGRATEABLE_POINTER static uint16_t * pData; // Pointer to buffer
 2 MIGRATEABLE static uint8_t pos;         // Position within buffer
 3 MIGRATEABLE static uint8_t window_size; // Size of the buffer
 4
 5 PROCESS_THREAD(agg_process, event, event_data)
 6 {
 7  PROCESS_BEGIN();
 8    while(1) {
 9      PROCESS_WAIT_EVENT();               // Wait for an event
10      if(event == EVENT_RESIZE) {         // A command-event
11          window_size = *event_data;      // Set window_size
12          pos = 0;                        // Reset write position
13          if(pData != NULL) migmem_free(pData); // Free old buffer
14          pData = migmem_alloc(window_size * 2); // Allocate new buffer
15      }
16      else if(event == EVENT_MODULE_DATA) { // A data-event
17        pData[pos] = * (uint16_t *) event_data; // Write data to buffer
18        pos += (pos + 1) % window_size;   // Move to next position
19        sp_send(avg(pData));              // Send data
20      }
21    }
22    PROCESS_END();
23 }
```

Listing 1. An example for a migratable data aggregation operator

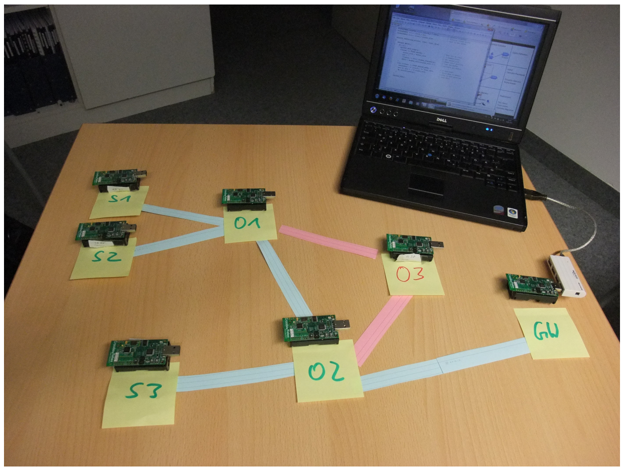

Figure 2. Demo setup:Three nodes are used to produce sensor data, three additional nodes represent potential positions for the operators, and a further gateway node connects the network to an attached PC.

Our query model is oriented to the *box-and-arrow* model as it is used for example by Borealis. In the sensor network, the nodes create three input streams: `S1`, `S2`, and `S3`. These input streams have temperature fields amongst others. The demo query creates output items if the temperature of `S3` is lower than in the area of `S1` and `S2`. It delivers these items to a sensor node that is connected to the gateway. We assume that the values of `S1` and `S2` might be erroneous, so we implemented a simple way of sensor data cleaning by using a minimum function as spatial granule and an aggregate operator for smoothing outliers.

In this example, we migrate operators between nodes. In the following, the migration of the `AGGREGATE` operator instance is described. It calculates the mean of a configurable number of samples. For a fictive reason, e.g. if there is a severe shortage of system resources, the central server decides to migrate this operator to a neighboring node. This includes transferring the module and its state, and rerouting the data flow.

The routing between the operators is done by our framework. Incoming data is received by the operators using the event system provided by Contiki. Outgoing data must be sent using special functions which route the data to the next operator.

Listing 1 shows a simplified excerpt of the aggregate module. It needs three main variables to store the window size, the position within the window and a pointer to reference the memory allocated from heap. As the window size and position are normal variables, they are marked with the `MIGRATABLE` keyword. For the pointer the `MIGRATABLE_POINTER` keyword is required, as the value of the pointer needs do be adjusted during the migration process.

When the operator gets a resize event (line 10), previously allocated memory is freed, just like normal heap memory, but using our API. Afterwards new memory is allocated from the heap space to store the incoming 16-bit samples. When data arrives (line 16), this memory can be accessed using the pointer, just like normal memory allocated from heap.

As the operator does not use any resources provided by the operating system (e.g. timers) no further actions are needed upon migration. Otherwise the operator would have to handle the migration-request event and free theses resources, before the migration can take place.

The demo shows the migration results by clearly outlining the current data stream configuration as well as the processing results using a set of TelosB sensor nodes. The demo setup is outlined in Figure 2.

## References

[1] P. Edara, A. Limaye, and K. Ramamritham, "Asynchronous in-network prediction: Efficient aggregation in sensor networks," *ACM Transactions on Sensor Networks*, vol. 4, no. 4, pp. 1–34, August 2008.

[2] L. Ying, Z. Liu, D. Towsley, and C. H. Xia, "Distributed Operator Placement and Data Caching in Large-Scale Sensor Networks," in *IEEE INFOCOM 2008*, Phoenix, AZ, April 2008.

[3] D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The Design of the Borealis Stream Processing Engine," in *CIDR*, Asilomar, CA, January 2005.

[4] Y. Zhu, E. Rundensteiner, and G. Heineman, "Dynamic Plan Migration for Continuous Queries over Data Streams," in *ACM SIGMOD Conference 2004*, Paris, France, June 2004, pp. 431–442.

[5] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *IEEE LCN 2004*, Tampa, FL, November 2004, pp. 455–462.

[6] M. Strübe, R. Kapitza, K. Stengel, M. Daum, and F. Dressler, "Stateful Mobile Modules for Sensor Networks," in *IEEE/ACM DCOSS 2010*. Santa Barbara, CA: Springer, June 2010, pp. 63–76.