Task Scheduling in Wireless Sensor Networks

Andrei Voinescu*
Dan Ştefan Tudose†
Nicolae Ţăpuş‡
Faculty of Computer Science and Computer Engineering
Polytechnic University of Bucharest
Bucharest, Romania
*Email: *andrei.voinescu@cti.pub.ro, †dan.tudose@cs.pub.ro, ‡ntapus@cs.pub.ro*

***Abstract*—This paper discusses a scheduling algorithm for the sub-tasks of an application in a Wireless Sensor Network. With the next generation of sensor networks, task-based systems are needed to provide services to entities outside the network. Allocation of tasks on different wireless nodes must take into account energy constraints, compatibility of tasks to a given node or topology and must be in agreement with the purpose of the network. The problem described in this paper requires maximizing network lifetime and satisfying these allocation constraints. The solution proves to be correct though in some cases it can lead to algorithmic complexity, such that further work on an approximation algorithm might be indicated.**

***Keywords*-wireless; sensor; task; scheduling;**

I. Introduction

Wireless Sensor Networks (WSNs) have different necessities than normal distributed systems, and as such scheduling has to be WSN specific. Wireless Sensor Networks are generally application-oriented networks, the requirements for these networks are often high-level (e.g., detect fires, monitor temperatures). As such, distribution of necessary work over the network can be transparent to the application that is built on top of it. Given an application that is divisible into a number of interdependent tasks, different allocation of these tasks on the nodes of the network will consume different amounts of energy. Energy is paramount to wireless sensor networks because it is assumed that their only power supply are batteries. Changing batteries relates to costs and network downtime, which has to be kept to a minimum. The goal of a task scheduler in this environment would be to maximize network lifetime, i.e., assign the tasks such that the application would run for the maximum amount of time. Since energy consumption in WSNs is mainly related to wireless communication, this paper proposes an algorithm that would allocate tasks to a WSN such that communication between nodes is minimal.

This paper describes two methods of solving similarly-defined problems of scheduling in the context of Wireless Sensor Networks (WSNs) in Related Work, then the authors' definition of the scheduling problem, followed by the algorithm itself.

II. Related Work

Related work for task scheduling is generally found under “task mapping“ or ”task allocation“. In recent times many articles discuss this topic, as it is fundamentally different to actively researched topics that mark a certain resemblance, such as Task Scheduling in high performance computing systems. WSNs have unique requirements in respect to network lifetime, availability, reliability that make research previously done not applicable. The objective of these articles is to find schedules with balanced energy consumptions for tasks. These schedules will allow a higher processing power of the resource-restricted WSN. Some research advocates splitting WSNs into lesser nodes and more powerful nodes, around which the other nodes are gathered, cluster heads. The lesser nodes deal with lower level sensing tasks, while cluster heads deal with higher level processing tasks. This generates an asymmetrical network that puts too much pressure on the cluster heads. Even if the cluster heads have much higher processing power and are not energy restricted (they are connected to a power outlet), this deviates from the concept of a WSN as well as not being as applicable. Hence the following related work is picked from those articles that either treat a WSN as homogeneous, or as a heterogeneous network in which there isn't a lot of difference in processing power between types of nodes.

A. EcoMapS

EcoMapS(***E**nergy-**co**nstrained Task **Map**pping and **S**cheduling*) is a scheduling system aimed to map and schedule tasks of an application with minimum schedule length subject to consumption constraints [1] . EcoMapS is application-independent, as it uses simple tasks with data dependencies noted in a Directed Acyclic Graph (DAG)[2]. The network is considered to be semi-homogeneous, being divided into smaller clusters, each having a cluster head. The cluster head is responsible for assigning the tasks in each cluster, as well as mediating communication (each cluster has star topology, with the cluster head in the middle; cluster communications do not overlap). The algorithm consists of arranging the tasks into a list such that each task is set after its predecessors and the most critical path is put first. The tasks are then assigned to the nodes where they can start executing earliest, following the E-CNPT[3] algorithm, an extension of the CNPT[4] algorithm which is subject to energy consumption constraints. The algorithm is run several times for different numbers of "active" nodes, the one with the least number of "active" nodes that meets the deadline requirements is considered to be the most energy efficient.

B. *PA-EDF*

A different aspect of the scheduling problem is scheduling a given set of tasks (can be repeatable) on a single node, taking into account energy efficiency, as done in [5]. For the scheduler to be able to make correct, power-aware decisions, tasks that are set to execute on the node must specify a worst execution time and a deadline, as well as an indicator of "importance". This "importance", also denoted as a power index, shows the relative importance of a task in relation to the other tasks under low-power conditions.

The main idea is that to extend network lifetime, non-critical tasks will be scheduled at greater intervals. All tasks have an initial deadline; if the sensor node is in a low-energy state (30% battery left), then the tasks runs a piece of code to redetermine its deadline in respect to the remaining battery life. Extending the deadlines for most of the tasks executing on the node can mean that there's time left in which the MCU is idle, which is supposed to consume less power than in the active state, hence the average power consumption is lowered and network lifetime increases.

III. Problem definition

The problem we address is an unconventional scheduling algorithm, in the sense that the main constraint is not time, but energy. As previously shown, research on scheduling is generally focused on hard time deadlines. Instead, we propose a solution where time is of least importance, preceded by energy consumption, battery awareness, availability and affinity.

The task that we wish to schedule is the smallest indivisible part of an application. Tasks can be classified into sensing tasks, actuating tasks, computation tasks, etc. For example, we have a fire detection system implemented with a WSN. We can have smoke sensing tasks on nodes that have smoke sensors, an event detection task, which detects in a stream of sensor input when smoke levels have risen, and an alarm task, which handles the behaviour of the network in the case of fire (bell, speaker, opening doors, etc.).

For the previous scenario, the scheduler needs some basic information for each task: its importance, an affinity to a certain type of node (a smoke sensing task can only be assigned to wireless nodes that have a smoke sensor), a frequency with which to run (if the task is repeatable) and dependencies (both data sinks and data sources). The scheduler will have to choose which assignment is best for energy consumption, to put intermediary tasks on sensing nodes or to put the on the alarm nodes (a variation of the algorithm that takes into account low-battery states for nodes or just where a task can't be scheduled due to network over-encumbrance can be implemented, similar to[5]).

The assumption is made that this algorithm runs in a single-hop network, as our initial testing platform (AVR Raven[6] with Contiki OS) only supports single-hop routing. Multiple hops could be quantified into the energy cost of transmission. As this scheduler targets the application layer, energy costs were considered to be proportional to the quantity of data transmitted. Although this is not necessarily true, it would not bring much benefits to the algorithm to include lower-layer energy consumption patterns.

IV. Algorithm

In this subsection we will formalize the problem defined. First, we will consider the total energy remaining in a node, W_{m_k}, m_k being the node. This energy can be deduced from the battery voltage and the discharge profile of the type of battery used. We will use this energy, together with the estimated consumption per second, to deduce the number of seconds that the node can run. The minimum of these times, calculated over each node in the network, will be the network lifetime. The purpose of the scheduling algorithm is to maximize this value.

Because the platforms that we use never enter sleep mode (and their consumption is always very low compared to the transmission/reception power consumption), we will consider the energy they use incorporated into P_{idle}, which can be specific to each node, $P_{idle_{m_k}}$. This value represents the energy consumed by the sensor node during idle mode in one second.

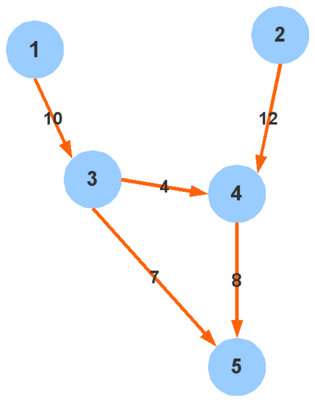

Figure 1. *A Directed Acyclic Graph with edges proportional in weight to transmission energy cost.*

We consider the energy wasted in trasmitting/receiving directly proportional to the number of bits in the payload, in part because we are interested in optimizing the schedule of tasks at an application level, and because costs that are not directly associated with the data do not scale with the increase in network traffic. To model the tasks and their dependencies we use a Directed Acyclic Graph (DAG), in which edges represent data dependencies, their cost being the maximal number of bits transmitted between the tasks (We consider that a transmission occurs after each period).

Let:

- $T(m_k)$ be the set of tasks allocated to node m_k.
- $P_{idle_{m_k}}$ the idle energy consumed by a node m_k during one second.
- $B(e_{ij})$, $e_{ij} \in E$ (E the set of edges in the task DAG) is the average number of bits per second transmitted by task i to task j
- $W_{tr\,b,m_k}, W_{rcv\,b,m_k}$ the energy cost of transmitting/receiving a payload bit on/from node m_k.
- $M(v)$ is the node to which the task v was assigned.

The power used by a task while receiving data would be:

$$P_{rcv,v_i} = \sum_{j,\, v_j \notin M(v_i)} B(e_{ji}) \cdot W_{rcv\,b,M(v_i)} \tag{1}$$

Aside from the exit point of the application, all tasks will also transmit data:

$$P_{tr,v_i} = \sum_{j,\, v_j \notin M(v_i)} B(e_{ij}) \cdot W_{tr\,b,M(v_i)} \tag{2}$$

Thus the network lifetime is:

$$\max_{k,\, m_k \in M} \frac{W_{m_k}}{P_{idle_{m_k}} + \sum_{i,\, v_i \in T(m_k)} (P_{rcv,v_i} + P_{tr,v_i})} \tag{3}$$

Taking into account that $P_{idle_{m_k}}$ is almost the same on all nodes, and presuming that we have the same amount of energy available from the batteries, the important part remains the energy consumed in radio transmission. Thus, to maximize network lifetime, a general goal would be to minimize this consumption. We do not consider tasks that need to be run on all capable nodes, we will address this as a restriction in the algorithm.

We have accounted for both energy while transmitting and while receiving. Even if they are not exactly the same, their sum should be uniform over each bit transmitted, so we can say that the power used by the network in radio communication, P_{radio} is

$$P_{radio} = \sum_{i,j,M(v_i) \neq M(v_j)} B(e_{ij}) * K \tag{4}$$

We have reduced the scheduling problem to a known graph-problem, for which a polynomial algorithm has been found in 1988. It is called the min k-cut problem. If we imagine in our setup the assignment of tasks to nodes, and ascertain that no task is duplicated among nodes (we can enforce that easily by duplicating in advance tasks that have to be run on all nodes), then the scheduling is in fact a partition of the node sets $\{C_1, C_2, ..., C_k\}$, each resulting set containing the tasks that have to run on that node. In graph theory, this is called a *k-cut*.

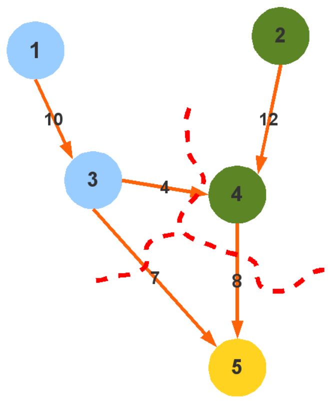

Figure 2. *A Directed Acyclic Graph with a minimal 3-way cut. Note that even though on this particular graph the cut is representable in 2D, this is not always true.*

A. *Minimal K-Cut*

Given a graph $G = (V, E)$ and a weight function $w : E \rightarrow$ and an integer $k \in [2..|V|)$ the k-cut is a partition of V into k disjoint sets $F = \{C_1, C_2, ..., C_n\}$ and its measure is the sum of the weight of the edges between the disjoint sets

$$\sum_{i=1}^{k-1} \sum_{j=i+1}^{k} \sum_{v_1 \in C_i v_2 \in C_j} w(v_1, v_2)$$

or otherwise written

$$\sum_{i,j,\, v_i, v_j \; in\; different\; sets} w(v_1, v_2)$$

The minimal k-cut algorithm is described in Listing 1

B. *The maximal source minimal s-t cut*

The (s, t) cut, needed by the min K-Cut algorithm, is a partitioning of the vertices of a flow graph such that the source is in S and the sink is in T. The min K-Cut algorithm needs a version where the source and sink are a set of nodes, not just one. To do this, we can collapse the nodes in the sink set into a super-node. Edges on the interior of the super-node do not count for the search of the minimal s-t cut, only those on its exterior. We then proceed to solve the maximum-flow problem on the graph, interpreting weights as flow capacities. Using the residual graph (graph with edges that have weight the capacity - flux passing at one time), we can start from the sink and expand until we hit 0 residual

Algorithm 1 Min K-cut algorithm

```
function KCut(V,k)
if k is even then
    k' = k - 2
else
    k' = k - 1
end if
S ← the set of subsets of k' elements from V
T ← the set of subsets of k-1 elements from V
Find $s \in S, t \in T$ such that W(cut(s,t)) = min
/* cut(s,t) splits V into s' and t'*/
/* Find the minimal cut(s,t) with maximal source set */
return  s' $\bigcup$ KCut(V-s', k-1)
```

capacity edges. The nodes found will be the smallest sink set of a minimal s-t cut, the source set being the rest of the nodes.

$$r(i,j) = c(i,j) - f(i,j)$$

Algorithm 2 Maximal source minimal s-t cut

```
/* replace sink and source set by supernodes */
$V' = V \bigcup \{s,t\} - (S \bigcup T)$
modify the edges external to supernodes
$E' = \{e_{ij} | i,j \notin S,T\} \bigcup \{e_{si} | e_{ji} \in E \, and \, j \in S\} \bigcup$
$\{e_{ti} | e_{ji} \in E \, and \, j \in T\} \bigcup \{e_{st} | e_{ij}, i \in S, j \in T\}$
$F_{ij} = 0, \forall i,j \in V$
loop
    find path $p$ from $s$ to $t$ in residual graph
    $m \leftarrow$ minimum residual capacity on path $p$
    for all edges $e_{ij}$, such that $e_{ij}$ on path $p$ do
        $F_{ij} \leftarrow F_{ij} + m$
        $F_{ji} \leftarrow F_{ji}$ - $m$
    end for
end loop
$A \leftarrow$ set of nodes reachable by BFS from $t$
$B \leftarrow V - A$
```

C. Adaptation of K-Cut

When reducing the scheduling problem to K-Cut, some constraints were ignored that now must be satisfied. Since every step of the algorithm is partitioning the node set in half, one being final and one remaining to be further partitioned, we can say that each step represents scheduling tasks on a single node. Constraints must be inserted in each step, relevant to the node whose tasks are being scheduled. In the pseudo-code of the algorithm, finding the source set S is what must be modified to satisfy constraints.

We have several constraints that must be added to the algorithm:

- Some tasks can only run on compatible nodes
- Some tasks have to run on all capable nodes (e.g. sensing tasks)

To enforce the first constraint we have to include only compatible tasks (tasks v such that $NA(v,m) = 1$ for the current node m) in the source set, as well as filter the cuts in which the first resulting set contains incompatible tasks. For the second constraint, we will include the tasks that have to be duplicated in the sink set of the cut (so that they will be available for the next step of the algorithm), then in the end we add those tasks to those obtained in the minimal (s,t) cut.

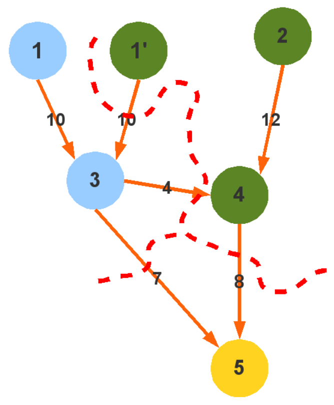

Figure 3. *A directed graph with a task that has to be duplicated on all capable nodes*

Algorithm 3 Adapted min K-Cut

```
function AKCut(V,k,$m_i$)
if k is even then
    k' = k - 2
else
    k' = k - 1
end if
MT ← tasks that have multiplicity
V' ← V - MT
S ← the set of subsets of k' elements from V'
T → the set of subsets of k - 1 elements from V'$\bigcup$ MT
Find $s \in S, t \in T$ such that W(cut(s,t)) = min
/* cut(s,t) splits V into s' and t' */
/* Find the minimal cut(s,t) with maximal source set */
$T(m_i) = \{s'\} \bigcup \{v_j | v_j \in \text{MT}, NA(v_j, m_i) = 1\}$
return  $T(m_i) \bigcup$ AKCut(V-s', k-1, $m_{i+1}$)
```

Assigning the tasks for each node at each step gives great versatility in constraints management for the algorithm to better model and solve the problem. For instance, battery status has no role yet, but it is easy to add: We set a threshold for the battery/tasks ratio, if the current node crosses that threshold we will settle on another solution, not necessarily with the same min k-cut, but with less strain on the node.

D. Complexity

The complexity of the min k-cut with the algorithm we described is:

$$T(n) = O(n^{k'+k+1}) \cdot O(n^3) + T(n-1),$$

where $O(n^3)$ is the bound for the minimum (s,t)-cut algorithm.

For k even, we have:

$$O(n^{2k-3}(n^3 + n^{2k-4}(n^3 + ...n^4(n^3 + n^3)))))) = O(n^{k^2})$$

A precise evaluation gives, as found by [7]:

$$O(n^{k^2-3k/2+2}),\ k\, even$$

$$O(n^{k^2-3k/2+5/2}),\ k\, odd$$

V. SENSEI STUFF

A. Check framework

In WSNs packets are typically forwarded in first-come first-served order. However, this scheduling does not work well in real-time networks where packets have different end-to-end deadlines and distance constraints. There are different healing strategies based on locality awareness of the sensor node and/or energy efficient algorithms: graph healing, binary tree healing, automatic fault recognition, the forgiving tree, DASH or SASHA. One of the generic design principles of SENSEI is the efficient utilisation of system resources, and the scheduling component of the Check framework is meant to address this point. Monitoring and actuation frameworks for heterogenous WS&AN islands crossing the borders of different organisations, each having a different network setup is a major challenge. The monitoring of all the components of Environmental Sensor Networks (ESN), Community Sensor Networks (CSN), and Body Sensor Networks (BSN), such as load, link quality, processor and radio usage on the nodes, as well as enabling actuation in these networks, is not yet solved in a heterogenous environment. The Check framework will therefore address this point by implementing multi-hop task-scheduling algorithms, based on multi-hop routing schemes for homogenous wireless sensor networks. The multi-hop task- scheduling scheme from the Check framework needs topology information from the network, which will be gathered using network discovery algorithms. Check is also used to provide a high-level, service-oriented self-healing [137][138] strategy. The WS&AN is regarded here as a service provider. Check thus offers a high-level framework for assuring service availability in WS&ANs. The self-healing [139][140] component identifies failing or poorly performing services and it orders the scheduler to reallocate them. Additionally, Check offers a centralized monitoring, control and reconfiguration framework, which will work toward the realization of the scalable internetworking, horizontalization and heterogeneity design goals of SENSEI. It allows the move of services in heterogenous WS&AN islands if data dependencies allow for it. The solution is based on the MonALISA [141][142] framework. A big challenge will be to enable push and pull algorithms to get monitoring data and issue commands to and from the monitoring nodes transparently, efficiently and reliably.

B. Check - Scheduling Algorithms for Task-based WS&ANs

Description: Task-scheduling is a fundamental requirement for the middleware subsystem in WS&ANs. It can be used, for example, in the smart places scenario to reschedule the appointments at the post office or the shopping mall, and in the worker in a plant to enable the parallel execution of tasks and to prevent workers from performing conflicting operations. The Check framework will implement a multi-hop task-scheduling algorithm, based on multi-hop routing schemes used in conjunction with other SENSEI subsystems (e.g. EMR) for homogenous wireless sensor networks. The multi-hop scheme needs topology information from the network, which will be gathered using network discovery algorithms that we will implement in our framework, as well as possible solutions from other middleware components (e.g. Titan [98]). Network Control will be provided by the Check scheduling service, which schedules tasks to given nodes through SENSEI specific RAI interfaces. Possible commands sent to sensors will be starting a task, ending a task, subscribing to a tasks output (which means that the task will send its output to multiple data sinks). The Check scheduling service will also respond to the self-healing service and reallocate tasks as needed together with other management components (e.g. Check Self-healing or SYNAPSE++). The service will base its reallocation on performance metrics and capability specifications obtained from the network.

Results: Currently we are successfully allocating tasks and resources on single-hop WS&ANs. Tasks can be regarded as functionalities offered by specific WS&AN islands by using SENSEI resources to achieve the desired design goal. The multi-hop scheme is currently under development. As a platform for development of our task-scheduling algorithms we are using Raven sensor boards and the Contiki Operating System [97]. Although the scheduling service is meant to be platform independent, the implementation of the middleware system local to a sensor board is deeply architecture specific. This means that the protocol can be defined and standardized for generic scenarios, while its specific implementation will depend on the hardware platform. The Contiki OS uses a cooperative model for processes. Each process that runs on the processor at any given time must at some point relinquish its position, or the system would come to a halt. Contiki's cooperative process subsystem is based on event queues, where events are associated with processes and wake them up. In reality, when one process yields it's control of the processor, the scheduler looks for a new process to run, so an event is taken out of the queue and the process associated with it takes hold of the processor.

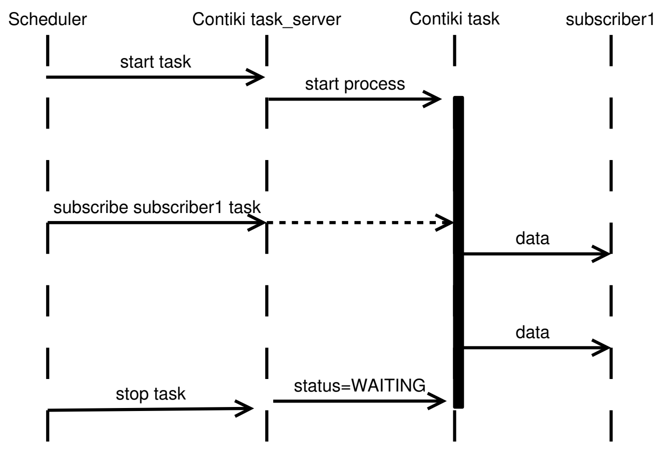

Figure 4. Task Scheduling in Contiki OS

There are two possibilities regarding the implementation of generic tasks, one is to have them all run under a single process, using timer events with a scheduler similar to real-time operating systems. A timer would be set to expire when the first task is due. The downside of this method is a difficulty in coming up with a metric for processor use versus idling. The other possibility is to treat a task as a process. The task manager starts and stops tasks, alters settings, append subscribers, etc. A performance metric can thus be calculated using this method concerning the scheduler. As the system keeps more and more tasks running and using even more CPU time, the time slices between schedules will consequently increase.

Starting the task is similar to starting a process in Contiki, while stopping a task means marking it as a waiting process, as can be see in Figure 5-1. While being marked as waiting, the task is in its (presumed) main infinite loop and it waits until it is taken out of this state by the scheduler. Obtaining information from the task can be done in one of two ways, namely: collecting data directly, with get/set ¡parameter¿, or using the data sink method, where the entity that connects to the sensor can register itself or another entity as a data sink for the output that is generated from the task. Parameters can be given, such as the frequency, with which data is forwarded to the subscriber. A list of processes can be kept with the linked list API available in Contiki OS. The scheduling service thus has three ways of controlling the running of tasks in the WS&AN island: by starting/stopping a task on a sensor; choosing data sinks for output data of a task; and by adjusting the frequency with which data is outputted. The scheduler then takes decisions taking into account the state of the network, the priority and complexity of the task to be executed.

Task scheduling in a star network topology involves the gateway sending commands directly to the sensor nodes, while in mesh networks the current implementation of task-scheduling uses the gateway to schedule tasks on remote nodes by sending commands via a multi-hop routing scheme, as shown in Figure 5-2. The state of the network, other than the topology of the network which is not discussed here, consists of the load on each sensor node. This can be estimated by measuring the time between two schedules of a given task, and taking that value, or a weighed sum over a number of iterations. This data is obtained by the scheduler either by subscribing to this task or by querying it directly from the task server interfaces on each sensor [99]. The energy left in the sensor must also be taken into account when scheduling tasks on a node. The energy estimation module (energest) provided by the Contiki platform, together with ADC data from the power supply are used to determine whether the given node is able to finish the task or not. Since providing continuous service for the task is paramount, it must first be determined if the task is not repeatable. If this is the case the node with the highest available energy is chosen when scheduling the task.

Currently there are two types of tasks being considered, namely: tasks that only have output (e.g. periodic tasks, that just calculate different metrics or send sensor data to data sinks), and tasks that process data. The latter can aggregate sensor data from several sensors (they are subscribed by the scheduler to the other sensors' data) and, for example, transmit a mean value or detect sudden change in the data. The scheduler then picks the nodes or sensors that can handle the extra energy loss due to the extra communication required to execute this type of tasks and schedules the tasks on these new resources.

Integration and Dependencies: The component from the Check framework implementing task-scheduling in WS&ANs contains five functional blocks that can be used through both standard and non-standard SENSEI interfaces. In Figure 5-3 one can see these elements, as well as some interfaces and functionalities the Check Scheduling component offers to the SENSEI system.

The Check-Scheduling component offers through its five functional blocks the following functionalities:

- Collect data:
  - Gathers data from the sensors inside a WS&AN island e.g. RAI.get(parameters)
- Start/Stop Tasks:
  - Start a task inside a WS&AN island e.g. RAI.set(run)
  - Stop a task inside a WS&AN island e.g. RAI.set(stop)

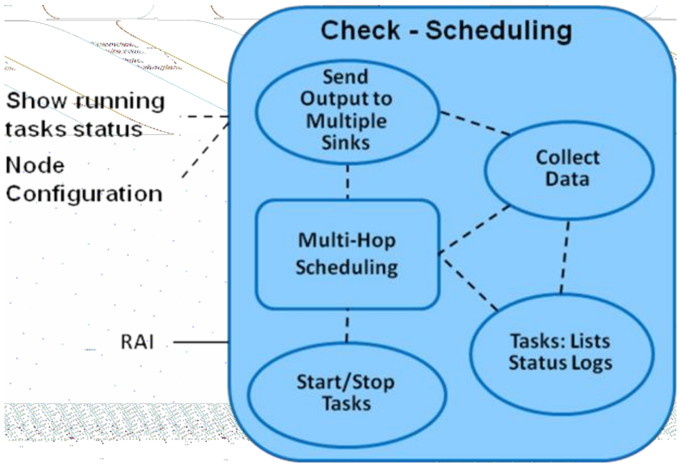

Figure 5. Check Scheduling as building block

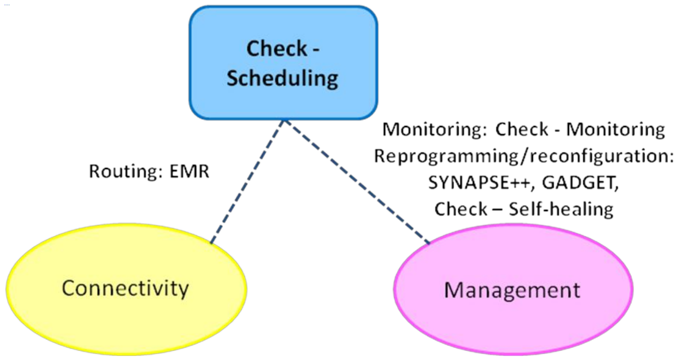
Check - Scheduling
Routing: EMR
Monitoring: Check - Monitoring
Reprogramming/reconfiguration: SYNAPSE++, GADGET, Check – Self-healing
Connectivity
Management

Figure 6. Integration and Dependencies of Check - Scheduling

- Set the task parameters on multiple nodes according to the scheduling algorithm inside a WS&AN island e.g. RAI.set(stop) or RAI.set(run)

- Send Output to Multiple Sinks:
  - The output of a certain task can be obtained by multiple sinks by subscribing to a listener associated to that specific task
- Tasks: Lists, Status, Logs:
  - List available tasks and show running tasks status
  - Show logs of scheduled/run tasks, thus allowing for statistics to be performed
- Multi-hop Scheduling:
  - Collect data, start/stop, send data to multiple sinks, gather status and logs of tasks inside a WS&AN island over multiple hops

The Check Scheduling component has the following dependencies, as shown in Figure V-B:

- Connectivity Subsystem:
  - Multi-hop Routing EMR (Energy-efficient Multi-hop Routing)
- Management Subsystem:
  - Monitoring Check-Monitoring for WS&AN islands
  - Reprogramming/Reconfiguration: SYNAPSE++, GADGET and Check-Self-healing

C. Check - Service Self-healing in WS&ANs

Description: Check is a framework used to provide a high-level, service-oriented self-healing 99 strategy. The WS&AN is regarded as a service provider. Check offers a high-level framework for assuring service availability in WS&ANs. Whenever a component of a WS&AN island fails, it is of paramount importance that the functionality it provides is not lost, to ensure the availability and reliability of the services being offered. Check Self-Healing is the SENSEI component providing the recovery strategies employed when these events occur. A service model is used for data input, output and processing in between various WS&AN nodes. The service model includes data sources and sinks which are connected to form a service graph. Services can be allocated to nodes by the middleware component Check-Scheduling. The self-healing component identifies failing or poorly performing services and signals and orders the scheduler to reallocate them or reallocates them itself and configures the nodes directly.

The self-healing algorithms 99 can manage multiple WS&AN islands through the Check Monitoring and Reconfiguration management component, and can thus move services in heterogenous WS&AN islands if data dependencies allow it. It must also be noted that this coupling of the Self-Healing and Monitoring components, by being able to gather information from a wide variety of devices, using many operating systems and offering numerous services, will provides a hardware- and operating system-independent mechanism for service-level self-healing in a SENSEI system.

Results: The first implementation of the service-level self-healing component of the Check framework is available and will be adjusted to conform to the required interfaces. The implementation has been tested in conjunction with a simulated WS&AN running the middleware Titan component, using the TOSSIM simulator. The following aspects were thus tested and verified:

- Node and service discovery through the Titan component:
  1) Determine the WS&AN topology
  2) Determine the services currently running on each node
  3) Determine the services that each node are capable of running
- Identification of failing nodes through communication analysis:

    1) Failing nodes are identified when their replies time-out too often
- Identification of failing nodes through data analysis:
    1) Failing nodes are identified when the sensor data they report contains a significant proportion of erroneous samples (e.g. values that cannot be found in a normal environment here there is a possibility of connecting with the outlier detection middleware component)
- Automatic engagement of the self-healing procedure:
    1) Upon detection of a failing node, the system successfully launches a self-healing procedure with the current WS&AN topology data.
- Manual engagement of the self-healing procedure:
    1) A manual procedure can be initiated with the intent of optimizing the current WS&AN, even if no hard failure is detected on any node
- Basic service-level self-healing algorithm:
    1) A Greedy algorithm was implemented for service reallocation, with constraints such as inter-service communication and node resources
- Near-transparent switch-over of services from failing nodes to healthy ones:
    1) The system was tested in configurations where communication between services is based on packets of raw or processed sensor data with little state information. In this case the self-healing procedure does not affect the correct functioning of the services, the only noticeable effect being a delay that can be interpreted as missing samples.
- Correct invocation of external task allocator:
    1) The self-healing component can interface a task scheduler and instruct it to allocate certain tasks away from a failing node.
- Correct reconfiguration of nodes:
    1) The self-healing component is also able to instruct the nodes directly to start necessary services and establish communication between them, bypassing any local WS&AN island task-scheduler.

Figure V-C illustrates the fault tolerance offered by the service self-healing component of the Check framework. In the first case when a leaf becomes unavailable, data from the other leaves continues to be aggregated and the service continues to be available. In the second case, when the aggregating node breaks down, the remaining functioning leaves are reallocated to another node, thus making the service available again.

The self-healing component is developed alongside management Check Monitoring and Reconfiguring component. The following functionality is currently implemented and available on a development WS&AN island using Sensinode nodes:

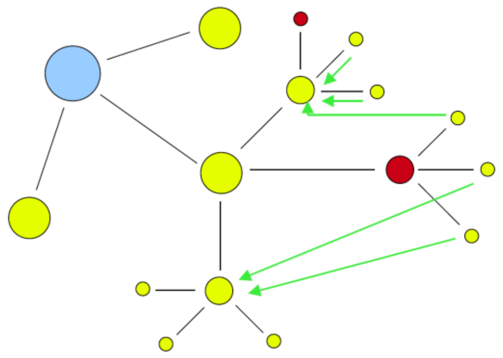
Figure 7. Service self-healing fault tolerance: leaf and aggregation node failure (red dots) - corresponding recovery/service migration strategies (green arrows)

- Node and service discovery:
    1) Determine the services currently running on each node of the island
    2) Determine the services that each node is capable of running
- Node parameter collection:
    1) Performance parameters from nodes are collected and stored in a repository, from where they are available to the self-healing component
- Identification of failing nodes using specific performance metrics like: the link quality, service availability, number of service time-outs, number of erroneous samples, etc.
- Setting of node parameters according to the deployed service self-healing algorithm.

Integration and dependencies: The service self-healing component from the Check framework over multiple WS&ANs contains four functional blocks which can be accessed through standard as well as non-standard SENSEI interfaces. Figure 6-67 shows these elements as well as some interfaces and functionalities the Check Self-healing component proposes to the SENSEI system.

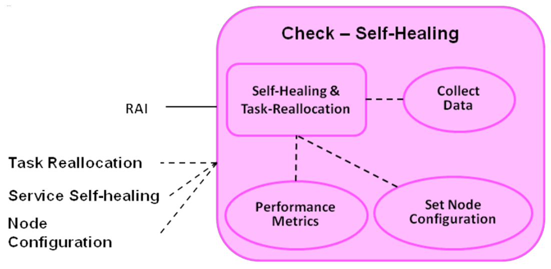

Figure 8. Check - Service Self-healing as building block

The Check Service Self-healing component offers through its four functional blocks the following functionalities:

- Collect data:
  - Collect node parameters and service data from the sensors in the network (RAI.get)
  - Discover available service providers in WS&ANs islands through middleware service discovery components like Titan
- Set Node Configuration:
  - Set multiple node parameters according to the self-healing algorithms over multiple WS&AN islands using other Management components for Monitoring and Reprogramming / Reconfiguration (e.g. RAI.set)
  - Start and stop services on different WS&AN islands using Middleware components (e.g. RAI.add, RAI.invoke)
- Performance Metrics:
  - Identify failing or poorly performing services according to system or user-specified performance metrics (e.g. RAI.get)
- Service Self-healing and Task-Reallocation:
  - Respond to the service self-healing requests (e.g. from SYNAPSE++ or Gadget) and determine a reallocation scheme for tasks as required (e.g. RAI.add, RAI.set, RAI.remove, RAI.invoke)
  - Task-Reallocation (ordered for example by SYNAPSE++ or Gadget) is based on performance metrics obtained from the specific functional block. The services needing reallocation are handled either:
    - ∗ Directly by the self-healing component by direct node configuration and service restart (e.g. RAI.add, RAI.set, RAI.remove)
    - ∗ Through the middleware component Check Scheduling which is able to set the required parameters in a single WS&AN island and start the necessary task according to the request from the Service Self-healing component (e.g. RAI.add, RAI.set, RAI.remove, RAI.invoke)

The Check Service Self-healing component has the following dependencies and possible integration possibilities, as shown in Figure 9:

- Connectivity Subsystem:
  - Multi-hop Routing EMR (Energy-efficient Multi-hop Routing)
- Middleware Subsystem:
  - Resource Discovery Titan
  - Task-scheduling Check-Scheduling for a WS&AN island
- Management Subsystem:
  - Reprogramming/Reconfiguration: SYNAPSE++, GADGET
  - Monitoring Check-Monitoring for WS&AN islands

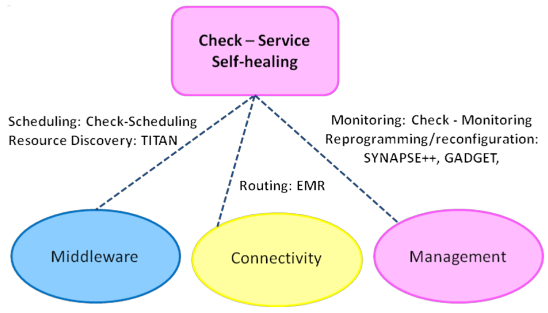
Check – Service Self-healing
Scheduling: Check-Scheduling
Resource Discovery: TITAN
Monitoring: Check - Monitoring
Reprogramming/reconfiguration: SYNAPSE++, GADGET,
Routing: EMR
Middleware
Connectivity
Management

Figure 9. Integration and Dependencies of Check-service Self-healing

D. Check - Monitoring and reconfiguration of WS&AN islands

Description: Monitoring and reconfiguration in heterogenous WS&AN islands is a non-trivial task that is a essential for any system aiming at providing complex functionalities across different types of wireless devices. Check Monitoring and Reconfiguration is the SENSEI component providing a wide spectrum of monitoring capabilities for WS&AN islands implemented with different technologies as well as facilitating their reprogramming and reconfiguration. A centralized monitoring, control and reconfiguration framework for WS&AN islands is thus being developed. A data collecting service runs on a gateway node and uploads relevant data to an Internet-based repository. Data is gathered either by polling specifically, or by intercepting normal traffic. The repository can then be accessed by client software running on PCs or PDAs, which summarizes the data to provide relevant information queried by administrators. Commands can also be given in the reverse direction, and the configuration of components in WS&AN islands can also be modified.

Our solution is based on the MonALISA [141][142] framework (http://monalisa.cern.ch/), developed by a team from Politehnica University Bucharest in collaboration with Caltech and CERN, which has been successfully used in monitoring large distributed systems, as depicted in Figure 6-6. The software monitors parameters related to the health and performance of all the components of ESN, CSN, and BSN networks, such as load, link quality, processor and radio usage on the nodes, etc. Automatic detection will be implemented as well, to support automatic monitoring of new nodes that enter the network, in conjunction with the Check Service self-healing component that is also being developed as a management component. Other types of sensor node hardware, as well as network software will be supported.

The main difficulty in setting up a monitoring framework for WS&AN is the network. Just like computing cluster

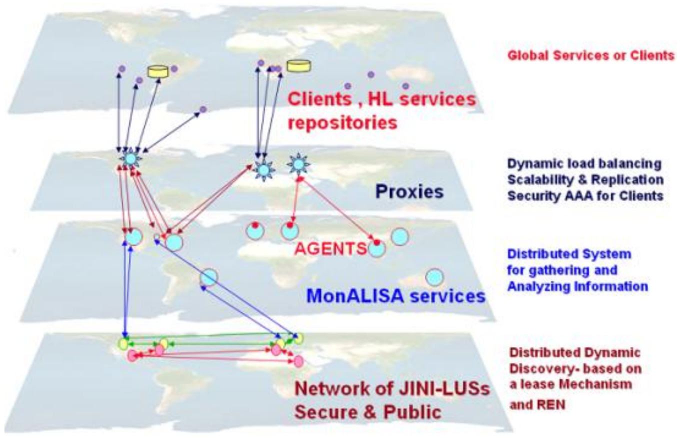
Global Services or Clients
Clients , HL services repositories
Dynamic load balancing
Scalability & Replication
Security AAA for Clients
Proxies
AGENTS
Distributed System for gathering and Analyzing Information
MonALISA services
Distributed Dynamic Discovery- based on a lease Mechanism and REN
Network of JINI-LUSs Secure & Public

Figure 10. General Architecture of the Monitoring Framework

grids, WS&ANs will cross borders of different organisations, each having a different network setup in place with its own rules. We plan to deploy an agent based network that will consitute a basic overlay over the main network, that enables us to send and receive data from monitoring services without interfering with network firewall policies. This solution is currently known to support a running monitoring capacity of over 1.5 million parameters per second and it has proven itself as mature.

Furthermore another component of this system is the integration module with the gateway. The main challenge will be to enable push and pull algorithms to get monitoring data in and out of the monitoring nodes with regard to energy consumption. Using this architecture we will enable the use of private company policies inside the WS&AN network while still allowing a unified network monitoring system. The third component makes this unified WS&AN overview system to all clients by enabling the discovery and registration of the collected data.

Results: Results for monitoring and reconfiguration of SENSINODE and AVR Raven nodes are available as shown in Figure 6-7. Nodes from ETHZ, UO and SEN will also shortly be visible through the monitoring framework.

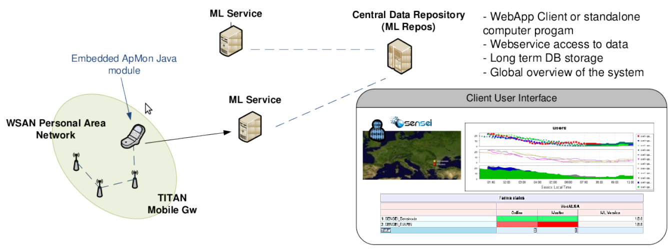
ML Service
Central Data Repository (ML Repos)
- WebApp Client or standalone computer progam
- Webservice access to data
- Long term DB storage
- Global overview of the system
Embedded ApMon Java module
WSAN Personal Area Network
ML Service
Client User Interface
TITAN Mobile Gw

Figure 11. Monitoring WS&ANs with Check

The framework has also been adapted to collect data from the Titan / TinyOS network. Monitoring data has been sent through an XDR protocol named ApMon in our monitoring system. The code has been ported to support Java enabled mobile phones used by the Personal Area Network testbed of ETHZ. The delay of reading measurements was one second, and workarounds to further compress this time delay have been found. All data was recorded in a central database named a MonAlisa Repository so that algorithms like the self - healing Check mechanism are able to interrogate this data source. An embedded C/C++ port of this library is under way to support Contiki nodes natively.

As a management application, this system has been tested so that it can control different node types by using unmodified node firmware and drivers. This communication is performed by using the MonAlisa Proxy services so it does not need direct network access. A monitoring and management module is being developed for the Gateway node so it can receive commands from a MonAlisa service, as presented in Figure 12.

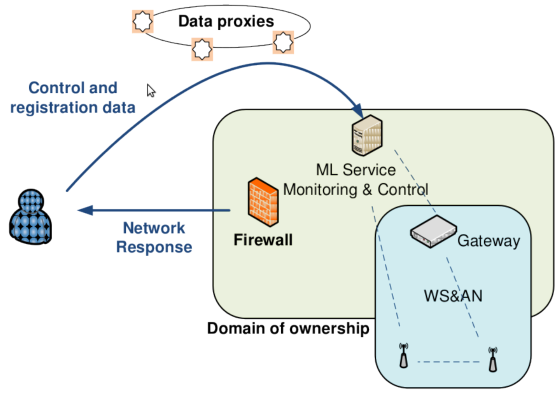

Figure 12. Check - Monitoring and Control Network

Integration and dependencies: Currently this system is modified to support communication through the Sensei RAI interfaces and is part of the implementation plan of WP5. Figure 6-9 depicts the components involved in the monitoring and reconfiguration. By design, the MonAlisa framework offers all- or-nothing remote control access the underlying services by using public-private key authentication. We will be able to interconnect to WS&AN nodes using fine grained AAA methods by using a dedicated management module on the Sensei Gateway. An added feature to that will be that data producers will use both push and pull methods from the nodes further enhancing the use of the nodes energy.

The Check framework implements the monitoring and reconfiguration component over multiple WS&ANs using four functional blocks. These can be used through standard and non-standard SENSEI interfaces. These elements are shown schematically in Figure 6-10. The Check Monitoring and Reconfiguration component offers through its four functional

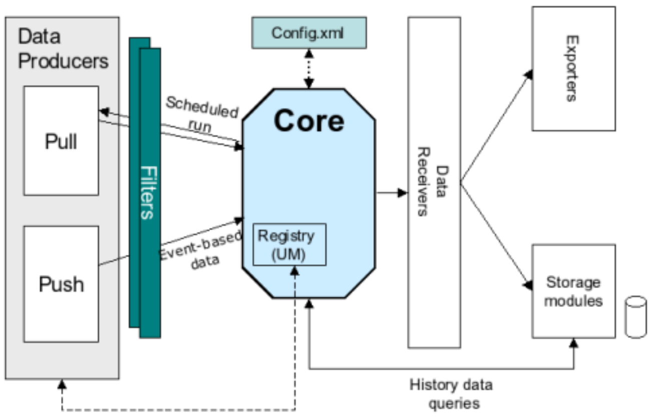

Figure 13. Check - Monitoring and Reconfiguration Core

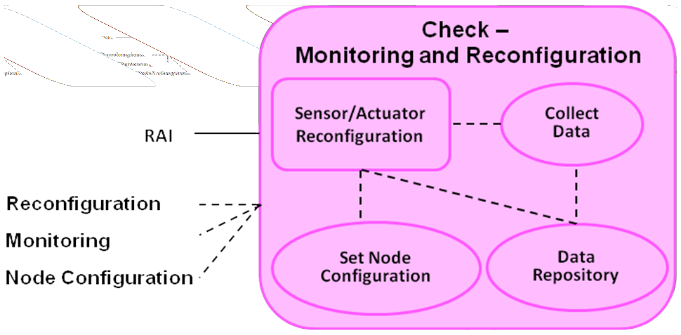

Figure 14. Check - Monitoring and Reconfiguration as building block

blocks the following functionalities to the SENSEI system:

- Collect data:
  - Gathers data (e.g. load, link quality, processor, and radio usage) from the sensors and actuators distributed over multiple WS&AN islands a monitoring service runs on the gateway of each WS&AN island being monitored. (e.g. RAI.get)
  - Changes in the network configuration can be observed by using middleware service discovery component Titan. (e.g. RAI.get, RAI.invoke)
- Data Repository:
  - Store all the data gathered from various WS&AN islands on a dedicated internet- repository site. (e.g. RAI.add)
  - The stored data can be used by the reconfiguration functional block as well as for statistical purposes. Other management subsystems like SYNAPSE++ or GADGET can also query this data repository. (e.g. RAI.add, RAI.set)
- Set Node Configuration:
  - Set multiple node parameters according to the specification of the reconfiguration requested by users or algorithms. This can be done directly by Check or by using other management subsystems e.g. SYNAPSE++, GADGET. (e.g. RAI.invoke)
- Sensor/Actuator Reconfiguration:
  - Reconfiguration of nodes across multiple WS&AN island is offered through the services running on the gateway of each WS&AN island. (e.g. RAI.add, RAI.set, RAI.remove)

The Check Monitoring and Reconfiguration component has the following dependencies, as shown in Figure 6-11:

- Middleware Subsystem:
  - Resource Discovery Titan
- Management Subsystem:
  - Reprogramming/Reconfiguration: SYNAPSE++, GADGET

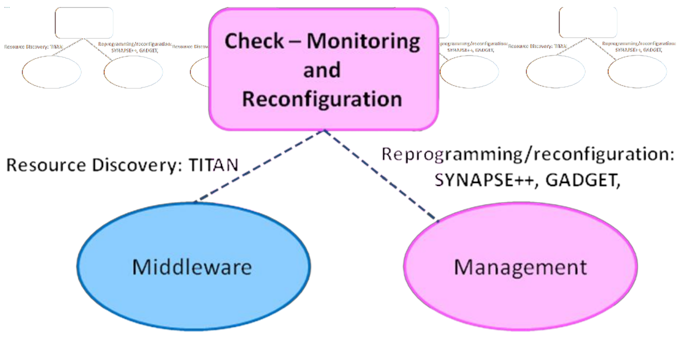
Check – Monitoring and Reconfiguration
Resource Discovery: TITAN
Reprogramming/reconfiguration: SYNAPSE++, GADGET,
Middleware
Management

Figure 15. Integration and Dependencies of Check - Monitoring and Reconfiguration

VI. Conclusion

The algorithm proposed solves the schedule problem with minimal energy consumption. It is designed for heterogeneous networks and it is application-independent. Although it was described as for a mesh topology network, the idea can be easily extended by introducing a “hop” factor in the communication between certain nodes - as dictated by topology.

The scheduler suffers from great algorithmic complexity, a different solution based on an approximation algorithm of the same problem is required for large-scale networks. A viable approach would be to use the theorem in [8], that states that the k-cut problem can be solved with Gomory-Hu trees within twice the optimal. An implementation based on this approximation (detailed in [9] and [10]) has proven to be within twice the optimal, as shown in Figure 4. The approximation solution to the scheduling problem also proves to be a viable alternative from the complexity point of view. Asymptotically it has the same complexity as the Gomory-Hu algorithm it is based on, although with a higher constant.

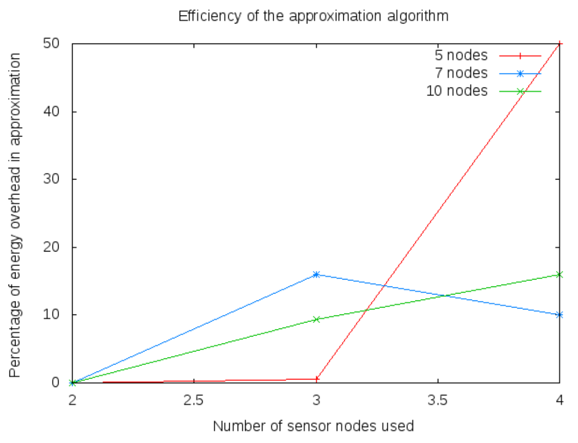

Figure 16. *Percentage of addition to the optimal of the approximation solution - Twice the optimal is represented by 100%.*

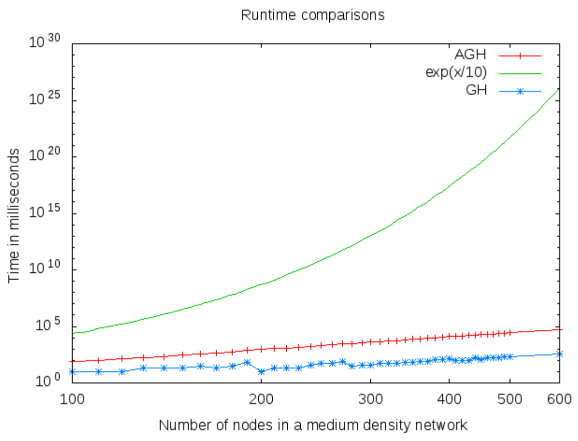

Figure 17. *Runtime comparisons (on semi-logarithmic scale) of variants of the approximation solution, GH is the standard Gomory-Hu algorithm, AGH is our solution of the scheduling problem based on Gomory-Hu*

Acknowledgment

The authors would like to thank Emil Sluşanschi, Răzvan Tătăroiu and Răzvan Rughiniş, for support and feedback.

This research is partially supported by the European Union through the FP7 IP Project Nr. 215923 SENSEI Integrating the Physical with the Digital World of the Network of the Future.

References

[1] Y. Tian, E. Ekici, and F. Ozguner, "Energy-constrained task mapping and scheduling in wireless sensor networks," in *Mobile Adhoc and Sensor Systems Conference, 2005. IEEE International Conference on*, Nov. 2005, pp. 8 pp.–218.

[2] Y. Tian, Y. Gu, E. Ekici, and F. Ozguner, "Dynamic critical-path task mapping and scheduling for collaborative in-network processing in multi-hop wireless sensor networks," in *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, 0-0 2006, pp. 8 pp.–222.

[3] T. Hagras and J. Janecek, "A high performance, low complexity algorithm for compile-time task scheduling in heterogeneous systems," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, April 2004, pp. 107–.

[4] ——, "A high performance, low complexity algorithm for compile-time job scheduling in homogeneous computing environments," in *Parallel Processing Workshops, 2003. Proceedings. 2003 International Conference on*, Oct. 2003, pp. 149–155.

[5] F. C. Delicato, F. Protti, J. F. de Rezende, L. F. R. da Costa Carmo, and L. Pirmez, "Application-driven node management in multihop wireless sensor networks." in *ICN (1)*, ser. Lecture Notes in Computer Science, P. Lorenz and P. Dini, Eds., vol. 3420. Springer, 2005, pp. 569–576.

[6] Atmel, "Rzraven hardware user's guide." [Online]. Available: www.atmel.com/dyn/resources/prod_documents/doc8117.pdf

[7] O. Goldschmidt and D. Hochbaum, "Polynomial algorithm for the k-cut problem," in *Foundations of Computer Science, 1988., 29th Annual Symposium on*, Oct 1988, pp. 444–451.

[8] B. Jacokes, "Lecture notes on multiway cuts and k-cuts," July 2006. [Online]. Available: math.mit.edu/~goemans/18434S06/multicuts-brian.pdf

[9] C. Chekuri and S. Guha, "The steiner k-cut problem," *SIAM J. Discret. Math.*, vol. 20, no. 1, pp. 261–271, 2006.

[10] C. Checkuri, "Approximation algorithms: Lecture on multiway cut problem," 2009. [Online]. Available: www.cs.illinois.edu/class/sp09/cs598csc/Lectures/lecture_7.pdf