Indian Journal of Science and Technology, *Vol 9(24), DOI: 10.17485/ijst/2016/v9i24/96068, June 2016*
ISSN (Print) : 0974-6846
ISSN (Online) : 0974-5645

Efficient Remote Software Management Method based on Dynamic Address Translation for IoT Software Execution Platform in Wireless Sensor Network

Minwoo Jung[1], Dae-Young Kim[2] and Seokhoon Kim[3*]

[1]Future Technology R&D Division, Gyeongbuk Institute of IT Convergence Industry Technology, Korea; mwjung@gitc.or.kr
[2]Department of Software Engineering, Changshin University, Korea; kimdy@cs.ac.kr
[3]Department of Computer Software Engineering, Soonchunhyang University, Korea; seokhoon@sch.ac.kr

Abstract

Background/Objectives: In this paper, an efficient remote code execution technique is proposed to implement a storage-less sensor in the Internet-of-Things (IoT) paradigm. **Methods/Statistical Analysis**: To realize a flexible code update mechanism and to optimize the use of an IoT device in terms of utilizing its various functions, we adopt the concept of remote on-demand code execution (ROCE) to implement a storage-less sensor. **Findings:** Instead of using a conventional on-chip flash memory for an instruction code, an instruction memory is used wherein the remote storage area based on the IoT platform is virtually mapped onto the address space of the instruction memory using a dynamic address translation technique. The proposed storage-less approach using the remote resource as a virtual code space may be adopted to reduce the high access current and chip area overhead of an on-chip code flash memory. **Application/Improvements:** The proposed technique reduces the energy consumption and the packet delay of an IoT device for executing the remote embedded software, as well as realizes a storage-less sensor architecture.

Keywords: Dynamic Address Translation, IoT, Locality of Reference, Remote Software Update, Storage-Less Sensor

1. Introduction

In recent years, the Internet-of-Thing (IoT) has become one of the most well-known technologies enabling the hyper-connection of embedded hardware, software, cloud infrastructure, and various application services. The IoT platform includes an application layer, a middleware layer, and a physical sensor network layer for the basic machine-to-machine (M2M) communication[1,2]. The sensor network layer serving as a physical interface is wrapped by a virtual software sensor layer. This architecture ensures that the physical sensor interface is virtually linked with the middleware layer. The middleware layer communicates with virtual sensors and saves the sensed data into the server-side cloud. Further, this layer integrates the heterogeneous virtual sensors on a virtual plug-in interface with the low-level physical sensor layer and provides a pre-defined software interface in the application layer. Application developers to connect various sensors in the IoT indirectly access via the virtual interface provided by the middle-ware layer. The overall performance issues between the devices in the IoT platform have been studied in terms of energy consumption reduction, security, congestion control, and packet delivery ratio. The IoT devices are implemented using the tiny embedded system based on a microcontroller including an on-chip flash memory for the embedded software area. Previous studies have been conducted on the use of these conventional on-chip flash embedded microcontrollers and have focused on the application

*Author for correspondence

layer. Internet-connected IoT devices share data from the remote cloud server; but the embedded software code, which is statically compiled and downloaded, is difficult to update the pre-defined functions. Our study proposes a newly designed remote software management technique for efficient instruction code management; this technique involves the modification of the on-chip software code bus architecture. The proposed physical sensor downloads executable code blocks for each function unit and stores these blocks in the internal SRAM, eliminating the requirement of a large on-chip flash memory. The area cost and large access current attributed to the on-chip flash memory are replaced by a small on-chip SRAM and on-the-fly internet connection overhead. Figure 1 shows existing remote software update and proposed architecture. The remainder of this paper is organized as follows. Section II discusses the motivation behind this study, as well as related works. Section III describes the details of the proposed technique and the system architecture of the proposed IoT device for software code execution. Section IV presents the implementation and measurement results. Finally, Section V provides the conclusion of this study.

2. Motivation and Related Work

There are many reasons physical sensors have to be remote code update in IoT. In the case of some deployment scenarios, of for which reaching all physical sensors are impractical to the sensing process. However, it is critical to add or update the software on those physical sensors, post-deployment. For this purpose, the developer must be able to develop a program remotely for adding and/or updating the nodes with new functionalities. This feature is also beneficial for software maintenance and in-situ debugging. A drawback of the physical sensors is that they have limited power; in that respect, it is likely that an application running on the physical sensor may consume most of the memory of the physical sensors owing

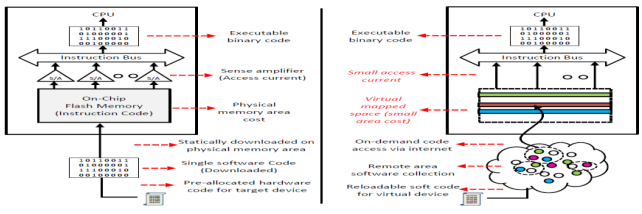

Figure 1. On-demand software execution framework architecture for efficient remote software management.

to data confidentiality and encryption. In such cases, it may not be feasible to transmit all of the newly updated code images in order to update the code of the physical sensor. A drawback of the existing techniques in this regard is that they need to maintain the code image in the memory, which imposes a memory overhead on the physical sensor, which has a constrained memory. The sensor network device used in the IoT platform is implemented on a tiny embedded system as a physical layer. The embedded system includes an on-chip flash memory in which the software code is programmed. The long-term lifetime of an IoT device depends on the static current in the sleep mode and the dynamic operating current that is responsible for executing the embedded software in the active mode. The dynamic operating current of the IoT device consists of the logic current and the code access current to the on-chip flash memory. The on-chip flash memory, in which the embedded software is installed, requires a large area of the entire IoT device chip and is responsible for the generation of a high access current in the sense amplifier in order to identify the programmed binary code. Several studies have been conducted on various approaches that can be used to reduce both the area overhead and the access current caused by the on-chip flash memory. However, it is more important to realize both a smaller on-chip flash memory and a low access current in battery-operated low-cost IoT devices requiring ultra long-term activation. The proposed technique replaces the area occupied by the on-chip code flash memory with the Internet-connected remote area, which is a virtual code space. The instruction code blocks to be executed are dynamically transferred via the connectivity to the internal code scratch pad, which is implemented with a small-sized SRAM. The between larger SRAM and the network overhead will be considered. In this study, we consider a method to reduce the energy consumption and memory overhead of physical sensors by the reduction of code images.

Software update has been raised as an important issue in the IoT paradigm. Existing approaches for software update can be classified into four main categories as followings: full image replacement, differential image replacement, virtual machines and dynamic operating systems. In the full image replacement, new binary images for an application and an operating system are transmitted over the network. It provides a very fine-grained control through the possible reconfiguration owing to the information that the transmitted images are compiled and

integrated for each iteration. However, these methods result in bandwidth overhead because the unchanged parts of an application can be retransmitted. To transmit a large image object from source nodes to many other sensors over a wireless network, Deluge, which is a reliable data dissemination protocol, is proposed by W. Hui et al[3]. In the differential image replacement, the changes between an executable deployed image and a new image are transmitted. While this method reduces bandwidth consumption, there is the fundamental difficulty related to replacing images. Panta et al.[4] represented a multihop incremental reprogramming protocol. Further, they proposed Zephyr that involves the transfer of the delta between the old and the new software. It enables the sensor nodes to rebuild the new software using the received delta and the old software. In addition, it reduces the delta size using application level modifications to mitigate the effects of function shifts. Then, it compares the binary images at the byte level with those obtained images using a novel method in order to generate a small delta, which is then transmitted to all the nodes. Zephyr is advantageous over Deluge in that it causes less traffic through the network and it requires less time for reprogramming than the existing incremental reprogramming protocol. However, this increases the similarity between the two versions, as well as imposes additional memory and performance defects. Jeong et al.[5] proposed an incremental network programming mechanism which quickly reprograms wireless sensors by transmitting the incremental changes for the new program version. This mechanism generates the difference between the two program images, enabling only the distribution of the key changes of the program, using the Rsync algorithm. In addition, the mechanism does not assume any prior knowledge of the program code structure and can be applied to any hardware platform. Virtual machines decrease the energy consumption related to transmitting a new functionality. However, in general, virtual machines allow only application updates and interpret the virtual machine codes. Therefore, they lead to runtime overhead and decrease the lifetime of sensors. Levis and Culler[6] designed Mate, which is a tiny communication-centric virtual machine for sensor networks. Its high-level interface allows complex programs to be very short (up to 100 bytes), reducing the energy cost for deploying new programs. In the Mate, a code is divided into small capsules of 24 instructions, which can self-replicate through the network. It is available to deploy ad-hoc routing and data aggregation policy through packet transmission and reception capsules. High-level program representation simplifies programming and allows large networks to be frequently reprogrammed in an energy-efficient manner. In addition, safe execution environment suggests the use of virtual machines to offer the user and kernel boundary on sensors which have no hardware protection mechanisms. Muller[7] presented SwissQM, which is a virtual machine designed to address all limitations. SwissQM includes a platform-independent programming abstraction that is geared towards data acquisition and in-network data processing. Brouwers[8] proposed a new kind of virtual machine and tool chain. The new virtual machine allows a significant subset of Java language to execute on a microcontroller for sensor networks. Dynamic operating systems offer the advantages of both the image replacement and the virtual machines, in that they enable fine-grained code updates at low transmission and run-time overhead. Dunkels[9] proposed Contiki, which is a light weight operating system that provides dynamic loading and replacement of individual programs and services. Although Contiki is built around an event-driven kernel, it provides optional preemptive multi-threading that can be applied to individual processes. Further, in the method, dynamic loading and unloading are feasible in a resource constrained environment, while ensuring that the base system is light weight and compact. In addition, Contiki allows only one-way linking for loaded modules and obligates more energy-intensive, polling-based service routines for interrupts. Its architecture restricts reconfigurations to application components. Han[10] proposed SOS, which is a new operating system for sensor nodes. It takes a more dynamic point on the design spectrum. SOS is composed of dynamically loaded modules and a common kernel, which implements messaging, dynamic memory, module loading and unloading, among other services. However, because of compiler limitations, it is needed for the SOS to exploit a position-independent code, and it is not completely supported on common sensor network platforms. Mottola et al.[11] proposed FiGaRo as a programming model supported by an efficient run-time system and distributed protocols. It is collectively enabling a novel fine-grained control over what is being reconfigured, and where. The programmer can explicitly handle dependencies of component and version constraints, as well as precisely chooses the subset of nodes targeted by reconfiguration, leaving the others unaltered.

3. Proposed Architecture and Techniques

We have proposed remote on-demand code execution (ROCE) to reduce the energy consumption and packet delay of physical sensors. Existing approaches for remote code update have considered that a server provides the entire code file to a physical sensor, which has a large on-chip flash memory and adopts a complex procedure to update the code file. On the basis of locality of reference, ROCE has significantly reduced the number of codes that are transmitted from a server to a physical sensor. Given that a single code is partitioned into function block units, the developer should register and update the code block in the cloud server. Further, the developer should create an efficient search table to find a suitable code block in the database; this table will facilitate a reduction in the overhead of the physical sensor, because the sensor will wait until it receives the next code block. The physical sensor loads the received code block on the small on-chip SRAM through a scheduler. The scheduler-which is the core component of the physical sensor-manages communication with the server. The functions of the scheduler are outlined as follows:

i. Generate a request message for the next function code
ii. Check the response message
iii. Load the received code block in the RAM after it has been checked
iv. Translate the dynamic address

One of the most promising features of ROCE is that a single code is propagated with a code block unit. Apart from this feature, ROCE offers several other advantages such as minimal energy consumption and minimal delay. We can use locality of reference, is a occurrence describing the same value or related storage locations, being frequently accessed, for a single code due to propagate with code block unit. The locality of reference facilitates the implementation of a memoryless sensor. ROCE can reduce the network overhead, which can occur because of the transmission of large data blocks. Existing architectures propagate the entire single code in an Executable and Linkable Format (ELF). Therefore, these architectures should use an on-chip flash memory to store the ELF, which results in the generation of a larger overhead than that caused by ROCE owing to the fact that redundancy codes are also included in the ELF. Energy consumption can be reduced by eliminating the flash memory and by decreasing the code size. Figure 2 shows architecture of ROCE.

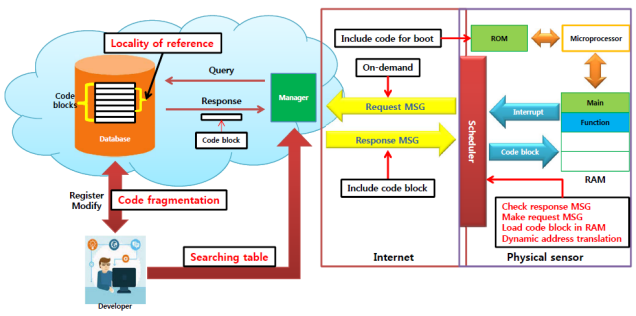

Figure 2. Overall of ROCE architecture.

3.1 Requirements

ROCE should be designed such that the following requirements are met:

i. The code block, which is transmitted from a server in the network, must reach all the physical sensors over the Internet.
ii. An execution file should be stored as the function block unit.
iii. The operation of ROCE should not influence the lifetime of the network (minimization of energy consumption).
iv. Memory usage should not be very high, because ROCE does not include a memory.

Further, in order to ensure the correct operation of ROCE, the following requirements should be satisfied. First, in order to construct a sensor network for the IoT system, a reliable remote code update is required to ensure that the code block is transmitted to all physical sensors in a stable network environment.

3.2 Locality of Reference

The principle underlying locality of reference implies that an application does not access all of its data at once with equal probability. Instead, it accesses only a small portion of it at any given time. Because of this principle, we can reduce the amount of transmitting data, owing to which we can be sure that the physical sensor can be implemented without a memory for achieving remote code update and the packet delay can be reduced.

3.3 Dynamic Binary Code Translation

The proposed approach assumes that the dynamically loaded software code bytes are relatively allocated on the physical code area of the target sensor. The conventional static compilation and linking approach for a new target device requires the programming of a flash memory by halting the previously installed IoT device. Although the on-the-fly software access reduces the overall performance of the system code execution, dynamic binary code translation enables the on-demand software code execution transparently without any system architecture modification. Figure 3 shows dynamic adddress translation. ROCE involves the use of a server and various physical sensors. The ROCE has each behavior of server and physical sensor. When a physical sensor requires other functions, it communicates with the server through ROCE. In this section, we deal with each behavior and interaction between a server and the physical sensors.

3.4 Physical Sensor Design for ROCE

The sensor adopts a particular procedure so that it can carry out ROCE. Once the sensor is turned on, the hardware resources are initialized. The sensor first requests for a global variable, stack size, and an interrupt vector table. After the sensor is initialized, it requests for the main function block, which once received is entered into the scheduler. The scheduler exhibits various functions such as frame check and binary translation. When the binary code of the main function block is transferred to the RAM, the main function block begins to execute instructions. When jump instruction is satisfied, the Program Counter (PC) shifts to the scheduler function. It calculates the start address of the next function block in the server and checks whether the function block exists or not in the schedule table. If the function block exists in the schedule table, it will be executed. If not, a request for the function block is sent to the server. The sensor carries

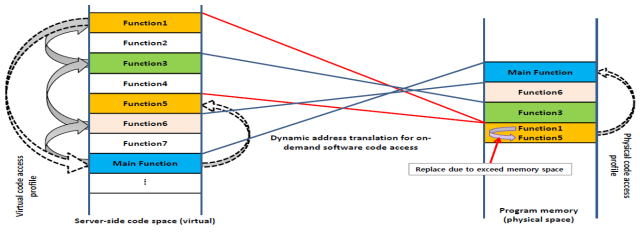

Figure 3. On-demand instruction code management for storage-less IoT device.

Algorithm 1 Internal procedure of physical sensor

```
1:  procedure Start − up(@Globalvariable, Stacksize, Interruptvectortable)
2:  if first request function then
3:     START BIT = Request message[0][0X88]
4:     END BIT = Request message[6][0XFCFC]
5:     ID = Request message[1][0X00]
6:     Address = Request message[2][0X00000000]
7:  else
8:     START BIT = Request message[0][0X88]
9:     END BIT = Request message[6][0XFCFC]
10:    ID = Request message[1][IDofnextfunction]
11:    Address = Request message[2][addressofnextfunction]
12: end if
13: for scheduler do
14:    check response message
15:    load code block in SRAM
16: end for
17: for execution do
18:    if meet function call then
19:       move scheduler
20:       if exist function in scheduler table then
21:          move execution
22:       else
23:          move request function
24:       end if
25:    else
26:       maintain execution
27:    end if
28: end for
29: return status
```

Figure 4. Algorithm on internal procedure of physical sensor.

out this procedure repeatedly. The function firstly checks start bit and end bit. Then, these bits are compared with those listed in the schedule table through the extracted address in order to check whether a duplicate function block exists or not. If a duplicate function block exists, the GetRecInfo function is executed. If it does not exist, the acquired information is added to the schedule table, and then, the GetRecInfo function is executed. The GetRecInfo function carries out binary translation and transfers the received function block to the RAM. Binary translation is carried out when the received frame comprises a jump or branch instruction. The jump or branch instructions are replaced with new binary which can move to start point of scheduler. Another function transfers received function block to RAM after checking of received frame. The sensor includes a temporary buffer in order to check the received frame. Figure 4 shows internal procedure of physical sensor.

3.5 Server Design for ROCE

The server required for ROCE is designed by a developer. The developer develops programs that are unique to a given sensor. The program is stored in the server, and users access the required files for executing the program over the Web protocol. The developer should provide an

appropriate file for the start-up of the sensor, containing attributes such as the global variable, interrupt vector table, and stack size. The developer also creates a search table for a request message. This table includes the function name, address of the function block, and the function size. The developer should reference the ELF file, which is generated as a binary file, in order to create the search table. The ELF file defines the entire execution protocol. The server searches for the requested function block from the sensor in the search table. Subsequently, the server generates a response message after extracting the function block.

4. Evaluation

We consider an 8-MHz TI MSP430 microcontroller with a Chipcon CC2420 IEEE 802.15.4 radio transceiver in order to verify the effect of ROCE. For the sake of evaluation, we consider the communication of a single test sensor node with a sink node. We implement the ELF files such that their sizes vary with the number of functions. Further, we extract the code size for locality of reference. We then compare the proposed ROCE with the existing remote code update system in terms of energy consumption and packet delay. Energy is consumed when the physical sensor receives and processes a code. Packet delay refers to the time spent in transmitting the required code.

4.1 Energy Consumption of ROCE

We compared energy consumption of ROCE and exisiting method. We implemented ROCE on TelosB platform. TelosB includes MSP430 as micro-processor and CC2420 as zigbee transceiver. We referenced specification of TelosB. We defined energy consumption that is sum of current during specific time. We calculated energy consumption when a physical sensor receives a file for execution.

$$E_{exe} = (N_f \times E_{RX}) + (S_L \times E_p) \quad (1)$$

N_f is the number of needed frame which is transmitted from server to physical sensor. E_{Rx} is value of energy consumption when the physical sensor receive a frame. SL is the size of needed function for execution. Ep is value of energy consumption when the physical sensor processes an execution file. Most of physical sensor for WSN has a payload which is the fraction of frame for data transmission. We could calculate the number of needed frame using payload. We extracted size of ELF file and functions for execution.

$$N_f = S_E / S_p \quad (2)$$

S_E is the size of ELF file, S_p is the size of payload.

We considered various sizes of needed functions for execution. Existing methods consider that the physical sensor receives entire ELF file. However, ROCE only is needed fractions of ELF file for execution. We observed that energy consumption can be drastically reduced. Figure 5 shows energy consumption between existing method and ROCE.

4.2 Packet delay of ROCE

The packet delay means duration in order to complete transmitting entire file for execution of physical sensor. We calculated the packet delay using frame size.

$$D_p = N_f \times T \quad (3)$$

D_p is the packet delay, N_f is the number of needed frame which is transmitted from server to physical sensor. T is duration that a frame is transmitted. We considered various numbers of needed functions for execution. Existing methods consider that the physical sensor receives entire

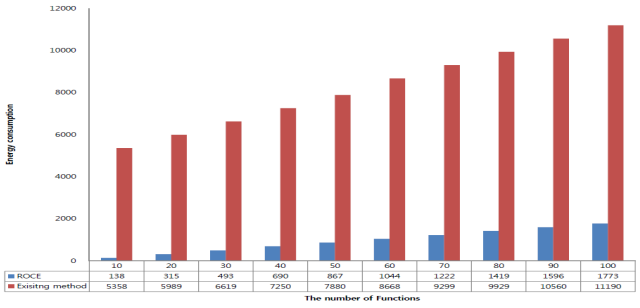

Figure 5. Compare energy consumption between existing method and ROCE.

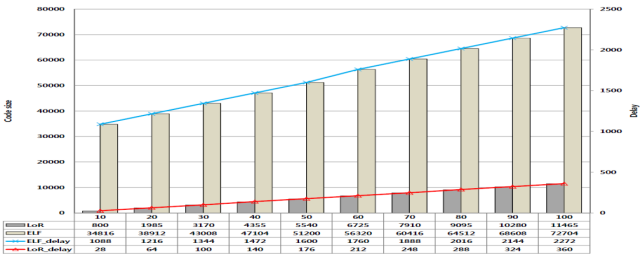

Figure 6. Reduction of packet delay and code size.

ELF file. However, ROCE only is needed fractions of ELF file for execution. We observed that packet delay can be drastically reduced. Figure 6 shows reduction of packet delay and code size between existing method and ROCE.

5. Conclusions

We have proposed ROCE for efficiently managing physical sensors. The characteristics of ROCE are outlined as follows.

i. Storage-less sensor: ROCE facilitates the elimination of a flash memory from a physical sensor, implying that the storage-less physical sensor will exhibit lower power consumption and a smaller size than the existing sensor.
ii. Efficient management of physical sensors: ROCE can enable the storage of all the information pertaining to the physical sensors to the cloud server. Further, it can control and easily monitor the sensor network in the IoT platform. It can also easily update the physical sensors.

ROCE can be used in several technologies in future, with one of them being the fragmentation technology. In the case of this technology, a single code of a physical sensor is stored as a function block. A drawback of such a type of storage mechanism is that the memory is used inefficiently. If fixed block, we can reference paging method. In future, we intend to conduct studies on a hybrid fragmentation method for source .

6. Acknowledgment

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2014R1A1A2060035). Also, this work was supported by the Soonchunhyang University Research Fund.

7. References

1. Jayavel K, Nagarajan V. Survey of Migration, Integration and Interconnection Techniques of Data Centric Networks to Internet- Towards Internet of Things (IoT). Indian Journal of Science and Technology. 2016 Mar; 9(11):1–8.
2. Sindhuja P, Balamurugan MS. Smart Power Monitoring and Control System through Internet of things using Cloud Data Storage. Indian Journal of Science and Technology. 2015 Aug; 8(19):1–7.
3. Hui JW, Culler D. The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. Proceedings of the ACM International Conference on Embedded Networked Sensor Systems (SenSys '04). 2004; 81–94.
4. Panta RK, Bagchi S, Midkiff SP. Zephyr: Efficient Incremental Reprogramming of Sensor Nodes using Function Call Indirections and Difference Computation. Proceedings of the 2009 conference on USENIX Annual technical conference (USENIX'09). 2009.
5. Jeong J, Culler D. Incremental network programming for wireless sensors. IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (IEEE SECON 2004). 2004. p. 25–33.
6. Levis P, Culler D. Mate: A Tiny Virtual Machine for Sensor Networks. Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems. 2002.
7. Müller R, Alonso G, Kossmann D. A Virtual Machine for Sensor Networks. Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems. 2007; 145–58.
8. Brouwers N, Corke P, Langendoen K. A Java Compatible Virtual Machine for Wireless Sensor Nodes. Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems (SenSys'08). 2008. p. 369–70.
9. Dunkels A, Gronvall B, Voigt T. Contiki - a lightweight and flexible operating system for tiny networked sensors. Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks. 2004; 455–62.
10. Han CC, Kumar R, Shea R, Kohler E, Srivastava M. A Dynamic Operating System for Sensor Nodes. Proceedings of the 3rd ACM International Conference on Mobile Systems, Applications, and Services (MobiSys'05). 2005. p. 163–76.
11. Mottola L, Picco GP, Amjad A. FiGaRo: Fine-Grained Software Reconfiguration for Wireless Sensor Networks. Lecture Notes in Computer Science. 2008; 4913:286–304.