Design and Implementation of Low power Wireless IPv6 Routing for NanoQplus

Jongsoo Jeong*†, Jaeseok Kim*†, Pyeongsoo Mah*†

*Department of Computer Software and Engineering, University of Science and Technology,
217 Gajeong-ro Yuseong-gu Daejeon, Republic of Korea

†Electronics and Telecommunications Research Institute,
138 Gajeong-ro Yuseong-gu Daejeon, Republic of Korea

jsjeong@etri.re.kr, jaeseok@etri.re.kr, pmah@etri.re.kr

Abstract— **Internet Protocol is a long-lived, stable, and highly scalable communication technology. In order to extend application space of wireless sensor networks into Internet, technologies that can provide Internet connectivity to Wireless Sensor Networks have been developed and lead the concept of Internet of Things. However, most of operating systems that support IPv6 are based on the event-driven model. Although the multi-threaded operating systems have an advantage of developing higher-level applications and services more intuitively, there are no researches about implementing the IPv6 on top of them. In this paper, we implement the IPv6 network stack including RPL on top of NanoQplus. And we introduce a secure protection method that mitigates thread stack overflow, which may frequently occur with long computations such as handling IP packets in multi-threaded operating systems. As a result, the proposed method ensures that relatively long IP-related functions are executed safely. Our experiments on the real test-bed demonstrate that our RPL implementation also works correctly, and achieves above 91% delivery ratio using hop count as a routing metric.**

Keywords— **IPv6, RPL, wireless sensor networks, OS, multi-threaded, NanoQplus**

I. INTRODUCTION

Interoperability is essential to give commercial values to wireless sensor networks. Many researchers have looked for methods to make WSNs interoperable since several years ago. As a result, 6LoWPAN Adaptation Layer [1] that is standardized by IETF [2] paved the way for transmission IPv6 packets over IEEE 802.15.4 networks. 6LoWPAN Adaptation Layer provides a methodology for one-hop transmission only, and does not define connecting nodes that are multi-hops away to the Internet.

IETF established ROLL (Routing over Low power Lossy networks) Working Group [3] and has suggested RPL (IPv6 Routing Protocol for Low power and Lossy Networks) [4] for LLNs such as IEEE 802.15.4 LoWPAN, low-power WiFi, and PLC (Power Line Communication) links. RPL is based on the Route-Over topology model [6], and has many recognized routing techniques such as Trickle timer, datapath validation, and so on. It is now poised to become an Internet standard.

The IPv6 including RPL is being implemented by many researchers of communities of TinyOS and Contiki [9][28][29]. These two operating systems are based on the event-driven programming model. The event-driven model has many advantages of no necessity of context switching, low memory cost of a single stack. These features also eliminates stack overflow problem caused by long computations such as handling IP packets. Therefore, they don't need to worry about memory when they implement the IPv6 network stack.

However, multi-threaded operating systems are different. Since they must maintain a small sized stack for every thread, stack overflow can occur when a thread executes excessively long computations. For this reason, thread stack usages must be considered when we implement the IPv6 network stack on them.

In this paper, we introduce a secure protection method that mitigates thread stack overflow, which may frequently occur with long computations such as handling IP packets in multi-threaded operating systems. In addition, we implement the IPv6 network stack including RPL on top of NanoQplus [5]. Based on the experiments, our method ensures that relatively long IP-related functions are executed safely. The memory foot-print of our IPv6 implementation is proper to the target hardware. According to the experiments on the real test-bed, our RPL implementation works correctly. It constructs and maintains a directed acyclic graph successfully. It achieves above 91% delivery ratio with using a hop count as a routing metric.

The rest of the paper is organized as follows. We talk about other embedded operating systems and IPv6 implementations for small wireless sensors in Section II, and then introduce an approach to handle IP packets safely in Section III. The overview of RPL and our objectives are described in Section IV. Then we present our experimental methodology and results in Section V, future works in Section VI with conclusions in Section VII.

II. RELATED WORKS

Most of the operating systems for wireless sensor networks are based on the event-driven programming model. Since this

model does not require context switching or memory space for thread stacks, it is able to keep operating systems light weight. It has also an advantage that it does not need to worry about stack overflow problem that can occur due to the long computations such as handling IP packets.

TinyOS [7] is the most famous event-driven based operating system for wireless sensors. It has the IPv6 implementation, named as BLIP [32]. Since TinyOS has a single thread and a single stack, the stack overflow does not matter when handling IP packets.

Contiki [8] uses Protothread [15] to take advantages of both multh-threaded and event-driven models. It keeps sequential program structure like multi-threaded OS with executing applications and OS services in event-driven manner. However, since it does not use context switching and preemption, it does not need to maintain a stack for every thread. Therefore, uIPv6 [26], the IPv6 implementation of Contiki is also designed without any considerations about original multi-threaded environments.

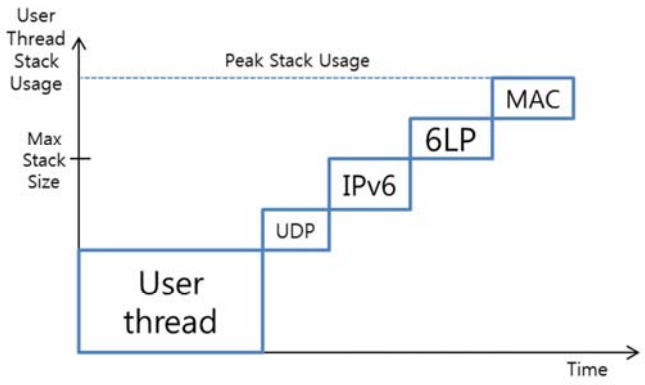
However, under the event-driven model, there is a critical disadvantage that developing higher-level services is not easy. Developers need to manage yield points or continuations, or partition a long-running computation to avoid missing events. It has been an obstacle to the advent of complex applications for WSNs. On the other hand, despite difficulties of maintaining thread stacks, multi-threaded operating systems have an advantage of programming intuitiveness. Since TinyOS team also recognized its importance, they have provided TOSThreads [14] package to support multi-threading in the event-driven kernel since 2.1.0. Thus, embedded multi-threaded operating systems are newly expected to promote births of future applications and services for Internet of Things.

III. Task Queuing to Handle IP Packets Safely

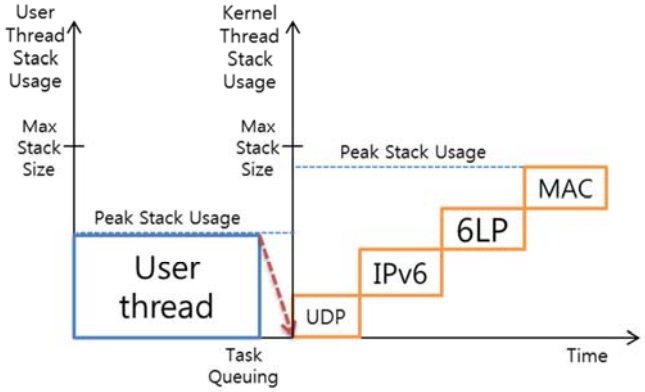
NanoQplus is multi-threaded operating system. It supports 7 user threads, and can extend number of threads up to 15. To execute threads concurrently, each thread has small space as thread stack. Since each thread stack has only 200 byte as default, calling long function chain must be rejected. In contrast, IP-related operations such as receiving, sending, and forwarding packets require relatively long computation. As described in Figure 1-(a), user threads that do not have enough remaining stack space, are vulnerable to thread stack overflow when they call one of the long IP-related functions.

In order to solve this problem, we introduce a task queuing method. NanoQplus has a task queue where tasks to be executed sequentially are posted to. All threads can post tasks, and all tasks in the task queue are executed by the system thread with the highest priority in the system. We make all IP-related functions to be posted to the task queue, instead of to be executed by user threads directly.

Since the system thread has a stack space independently to the caller thread's, node fails due to the stack overflow does not occur if the posted tasks do not require bigger stack space than the default size. Figure 1-(b) shows how the task queuing method can mitigate the caller thread's stack overflow. We also separate some excessively long chains of functions into short tasks to prevent the kernel thread overflow. Table 1 shows the examples of task queuing of the IP-related functions.



(a) Case of calling udp_sendto() in a user thread directly.



(b) Task queuing udp_sendto() to be executed by the kernel thread.

Figure 1. Task queuing to avoid thread stack overflow

TABLE 1. Task Queuing Examples

Function	Procedure
Sending a UDP datagram	• `udp_sendto()` – **`taskq_reg()`** • `ip_send()` – `6lp_send()` – `6lp_fragment()` – `mac_tx()`
Receiving a UDP datagram	• (Receiving a MAC frame) – **`taskq_reg()`** • `mac_rx()` – `6lp_recv()` – `ip_recv()` – `udp_recvfrom()` – **`taskq_reg()`** • `user_callback()`
Processing an ICMP message	• (Receiving a MAC frame) – **`taskq_reg()`** • `mac_rx()` – `6lp_recv()` – `ip_recv()` – `icmp_recv()` – (Processing) – **`taskq_reg()`** • `icmp_send()` – `ip_send()` – `6lp_send()` – `6lp_fragment()` – `mac_tx()`
Forwarding an IP packet	• (Receiving a MAC frame) – **`taskq_reg()`** • `mac_rx()` – `6lp_recv()` – `ip_recv()` – `6lp_send()` – `6lp_fragment()` – `mac_tx()`

Using the system thread is also good from the aspect of memory utilization. This is because our method uses the

system thread's stack space for temporary protocol headers such as IPv6, UDP, TCP. It enables to process packets faster by suppressing time-intensive dynamic memory allocations.

IV. RPL Overview and Our Objectives

RPL is designed with considering characteristics of LLNs (Low-power and Lossy Networks) and applications. First, the size of routing state should be minimized, and the routing protocol should be scalable. In LLNs, nodes usually have very limited memory space. In contrast, the number of nodes consist of a network will be from a few dozen to hundreds of thousands. Therefore, the routing state size of each node must not be affected by the network size as much as possible. Second, topologies of LLNs tend to change very frequently due to instable link states. To keep stable connectivity, the routing protocol must recover and optimize paths in response to packet losses. Third, nodes sometimes may be powered by batteries. Unnecessary transmissions of control messages may shorten lifetime of nodes and networks. Finally, there is no routing metric that requires all applications and environments. The only solution is enabling applications to participate routing algorithms such as choosing routing metrics and calculating paths directly. Table 2 shows whether existing routing protocols and RPL meet these requirements [25].

TABLE 2. Comparison of Existing Routing Protocols

Protocol	State	Loss	Control	Link Cost	Node Cost
OSPF/ IS-IS	Fail	Fail	Fail	Pass	Fail
OLSRv2	Fail	?	?	Pass	Pass
TBRPF	Fail	Pass	Fail	Pass	?
RIP	Pass	Fail	Pass	?	Fail
AODV	Pass	Fail	Pass	Fail	Fail
DYMO	Pass	?	Pass	?	?
DSR	Fail	Pass	Pass	Fail	Fail
RPL	Pass	Pass	Pass	?	?

In order to satisfy all requirements of routing protocol for LLNs, RPL uses several recognized routing techniques as follows:

A. Supporting three traffic patterns; MP2P, P2MP, and P2P

RPL supports three traffic patterns; Multipoints-to-Point (MP2P, upward), Point-to-Multipoints (P2MP, downward), and Point-to-Point (P2P) as depicted in Figure 2 [30] by constructing a Destination Oriented Directed Acyclic Graph (DODAG). To forward packets upward, each RPL router just forwards packets to its preferred parent. On the contrary, there are three modes of operation to forward packets downward; storing, non-storing, and upward-only. In the storing mode, each RPL router must maintain a downward routing table. Therefore, all routers are required to maintain an additional routing table. RPL routers can conserve memory by using non-storing mode (source routing with RH4 [18]), or disabling downward routing (upward only). P2P deliveries can be done by a combination of upward and downward deliveries. Since these routing policies do not provide the shortest paths, they might not be efficient. However, considering most of they are resource-constrained embedded devices such as wireless sensor nodes, low-power PLC nodes, the size of routing state for each node might be reasonable.

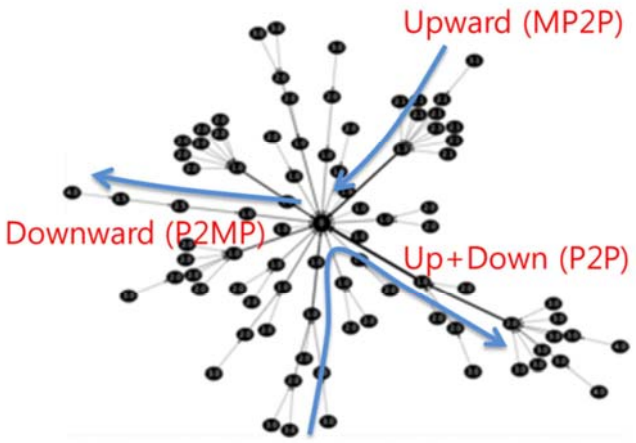


Figure 2. RPL traffic patterns

B. Constructing a topology proactively and suppressing control messages in steady-state.

Initially, RPL nodes begin constructing a DODAG proactively from a root at the center. In other words, it eliminates an overhead of flooding Route-Request packets that is used in several well-known ad-hoc routing protocols, i.e., AODV [10]. However, periodic transmissions of control messages in the proactive fashion are also wasteful when there are few packets to transmit. RPL uses Trickle timer [11] to suppress the interval of transmissions of control messages up to about 2 hours after entering steady-state. Therefore, RPL can satisfy the control overhead requirement.

C. Datapath validation

RPL root has a lowest value as its rank. The rank value increases in proportion to the distance from the root. Each data packets in RPL domain must contain a Hop-by-Hop IPv6 extension header with a RPL option [17] that contains a rank value of forwarder, and forwarding direction. In order to prevent a loop, RPL routers check this option hop-by-hop. For example, if a downward packet is arrived from a node of a higher rank, or an upward packet is arrived from a node of a lesser rank, they reset Trickle timer to recover paths reactively. This way is similar with Collection Tree Protocol [21].

D. Separation Objective Functions from the core protocol

In RPL, the types of routing metrics and the methods to calculate paths are not fixed. Because objectives maybe different according to the characteristics of LLNs and applications. In order to support various LLNs, and to achieve various applications' goals, RPL is separated into a generic core protocol and objective functions. ROLL Working Group has suggested Objective Function 0 (OF0) [20] that uses hop

count as a routing metric, and will continue to make various OFs in the near future.

All above features of RPL have not been developed yet. We focused on implementing mandatory functions first. In order to achieve it, we arranged several objectives of our RPL implementation as follows:

1) Upward Routing Only: In RPL, upward and downward routing mechanisms are separated. And enabling or disabling downward routing is also available. Since we want to evaluate mandatory operations first, we postpone implementing and evaluating downward routing to the future.

2) Objective Function 0: RPL can change OFs to achieve applications' goals. Our implementation only supports OF0 currently. Beginning from OF0, we will extend the number of supported OFs.

3) Interaction with 6LoWPAN Neighbor Discovery: All RPL operations are on the basis of Neighbor Discovery Protocol. In the LoWPAN, LLN-optimized 6LoWPAN ND [12] is suggested to eliminate inefficiency problems that can occur when adopting the classic IPv6 Neighbor Discovery Protocol [13] to the LoWPAN environment directly.

V. Evaluations

Figure 3 shows the implementation of IPv6 for NanoQplus. All components are RFC-compliant, and designed to be operated in multi-threaded kernel safely. In this section, we evaluate its complexity, robustness, and performance.

TABLE 3. ROM and RAM Usages

Component	ROM	RAM (static)	RAM (dynamic)
IPv6	5572	30	20 / address 21 / neighbor 21 / prefix
ICMPv6	1290	0	
6LoWPAN ND	6426	3	14 / border router
RPL	5308	3	52 / DODAG
RPL OF0	1098	0	
Trickle	1136	0	

A. Memory footprints

We implemented the IPv6 network stack for ATmega128L [19] MCU that has 128KB as program memory and 4KB as SRAM. Table 3 Describes ROM and RAM usages of our implementation. There are two RAM usages; static and dynamic. Static usage is for global variables. Dynamic usage is for allocated from heap to maintain states such as routing table entries, neighbor cache entries, and so on.

According to this table, the memory foot-print of our implementation is proper to the target MCU. However, it is not suited for some MCUs that have small program memory. 4KB RAM is also not considered enough so much. Since the IETF has standardized various constrained protocols such as CoAP [22] in addition to RPL, introducing better hardware platforms such as Imote [31], Egs [16] that have sufficient resources to cover all IP-based protocols should be considered hereafter.

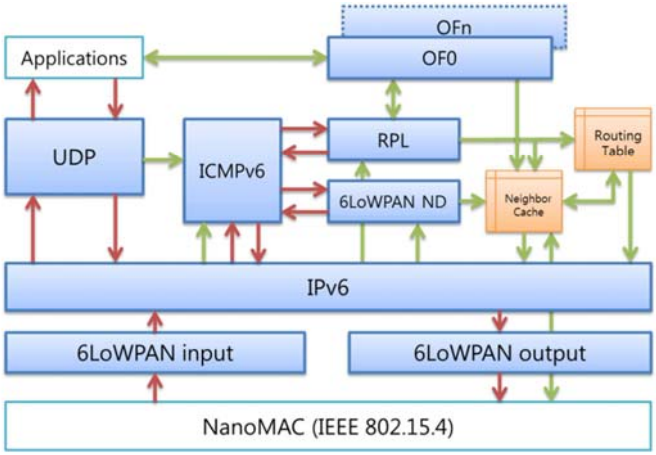


Figure 3. IPv6 Network Stack for NanoQplus

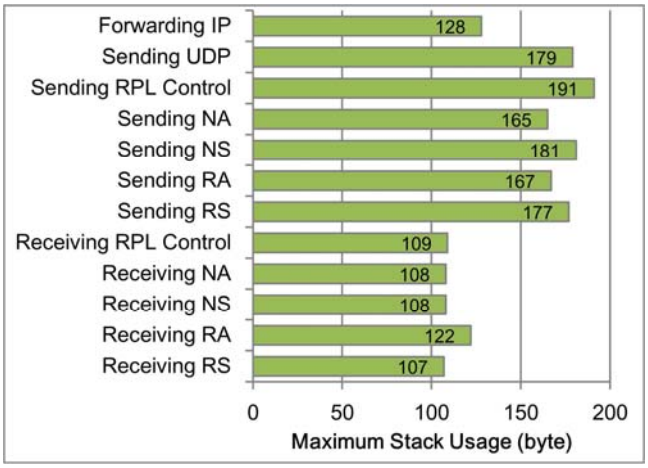


Figure 4. Maximum thread stack usages of IP-related functions

B. Thread stack safety

To verify our approach to handle IP packets safely from thread stack overflow, we measured the stack pointer values of MCU when the IP-related functions are executed.

Since all functions use the system thread's stack instead of user threads', we can say that they are secure if they are executed without using stack over 200 bytes (default size). Figure 4 shows the maximum thread stack usages of the IP-related functions. All of they are executed without any stack overflow. Sending functions tend to use more memory than receiving functions. This is because the IPv6 for NanoQplus is designed to use this stack space for temporary protocol headers to reject the time-intensive dynamic memory allocations.

C. *Network stability*

In order to evaluate network reliability, we experimented on the real test-bed where is an indoor place of 42m x 24m as shown in Figure 5. We used ATmega128L based platforms such as MICAz [33], Zigbex [34], and Nano24 [35] as nodes. All nodes are installed on the ceiling, and power-supplied. 'B' is a border router, and acts as a RPL root. And '0'~'9' nodes act as RPL routers.

The experiment was continued for about 17 hours. To generate work load, we configured all nodes to send the root a UDP datagram that has 30 byte data on every minute. When the experiment ended, each node generated about 1570 packets.

Figure 6 shows the number of transmissions every minute during 30 minutes after initialization. There are lots of transmissions of control messages in the initial-state due to neighbor discovery and constructing a DODAG. However, they are suppressed successfully by Trickle timer after entering steady-state. In addition, according to similar curves of RPL control messages and 6LoWPAN-ND messages, our RPL and 6LoWPAN-ND are interacting with each other well.

TABLE 4. UDP PACKET DELIVERY RATIOS

No	Source IP address	Ratio (%)
0	fc00::ff:fe00:30	100
1	fc00::ff:fe00:31	99.96
2	fc00::ff:fe00:32	99.79
3	fc00::ff:fe00:33	99.30
4	fc00::ff:fe00:34	99.96
5	fc00::ff:fe00:35	99.01
6	fc00::ff:fe00:36	98.03
7	fc00::ff:fe00:65	94.44
8	fc00::ff:fe00:66	91.39
9	fc00::ff:fe00:67	98.48

Table 4 describes packet delivery ratios of all nodes. All nodes deliver packet successfully at the rate of above 91%. Figure 7 shows them according to packet source's rank. In OF0, the root has '1' as its rank, and ranks of nodes are increased in unit of '1024' proportionally as the number of hops from the root increases. As a result, packets sent from nodes that are 4 hops away from the root are delivered at the rate of only 91%. These ratios are not better than existing routing protocols'. This is because our implementation used only OF0 that uses hop count as a routing metric. Adopting new OFs that use more effective metrics such as LQI [23] or ETX [24] will make better success rates than OF0.

VI. FUTURE WORKS

As our experimental result, our IPv6 including RPL implementation achieves all three objectives that are mentioned in Section IV. However, there are still some issues to be improved.

Our RPL implementation does not show better performance than existing routing protocols due to the weak routing metric of OF0. We will continue implementing multiple OFs to support various applications and environments. In addition to this, we will provide user applications an interface to participate in routing algorithms directly. In order to support the RESTful [27] protocols such as CoAP, or HTTP that are based on bi-directional communications, we also have to update our implementation to support downward routing and forwarding as soon as possible.

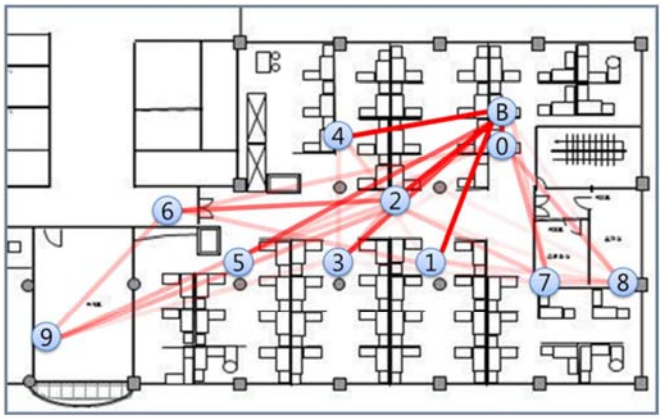


Figure 5. Test-bed to verify the operations of IPv6 and RPL

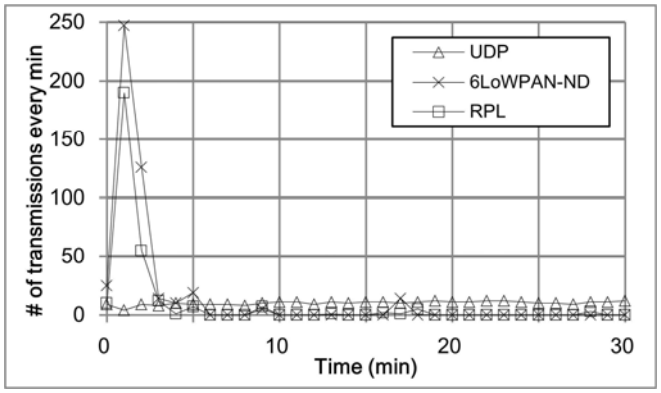


Figure 6. Control overhead

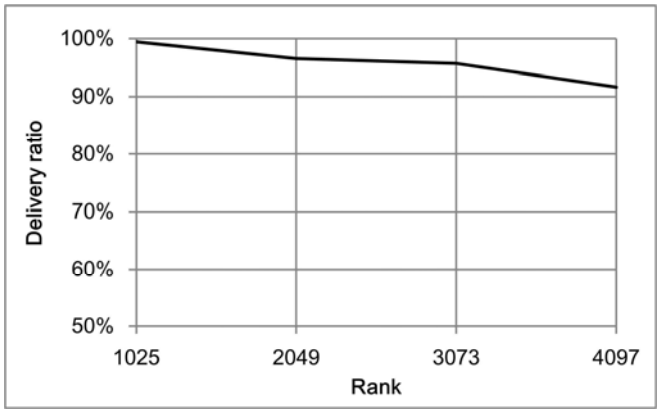


Figure 7. UDP Datagram delivery ratio vs. rank

VII. CONCLUSION

Multi-threaded operating systems have been spotlighted again due to the intuitiveness of programming. However, there are no researches about implementing the IPv6 network stack

for them without thread stack overflow. We introduced the secure protection method to handle IP packets in multi-threaded OS. Based on this, we implemented the IPv6 and RPL on top of NanoQplus. Our implementation did not show better performance than the existing routing protocols. However, if we continue implementing a full fledge RPL with multiple OFs and user interface that enables applications to participate in routing algorithms directly, NanoQplus with IPv6 is expected to promote opening a new era of the Internet of Things in the near future.

ACKNOWLEDGMENT

This work was supported by the Industrial Strategic technology development program, 2010-35-142, Development of Audio/Video Group Communication System Platform in Mobile Ad-hoc Environment, funded by the Ministry of Knowledge Economy (MKE), Korea.

REFERENCES

[1] G. Montenegro, N. Kushalnagar, J. Hui, and D. Duller, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks," RFC 4944, IETF, Sep 2007.
[2] Internet Engineering Task Force: http://www.ietf.org
[3] Routing over Low power and Lossy networks Working Group: http://tools.ietf.org/wg/roll
[4] T. Winter, P. Thubert, A. Brandt, T. Clausen, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, and JP. Vasseur, "RPL: IPv6 Routing Protocol for Low power and Lossy Networks," draft-ietf-roll-rpl-12, IETF, Oct 2010.
[5] S. C. Kim, H. Y. Kim, J. K. Song, M. S. Yu, and P. S. Mah, "NanoQplus – A Multi-Threaded Operating System with Memory Protection Mechanism for Wireless Sensor Networks," in *Proceedings of the 1st China-Korea WSN Workshop (CKWSN)*, Chongqing. China, 2008.
[6] J. Hui, and D. Culler, "IP is Dead. Long live IP for Wireless Sensor Networks," In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, 2008.
[7] TinyOS: http://www.tinyos.net
[8] Contiki: http://www.sics.se/contiki
[9] N. Tsiftes, J. Eriksson, and A. Dunkels, "Low-power Wireless IPv6 Routing with ContikiRPL," In *Poster Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, 2010.
[10] C. E. Perkins, E. M. Belding-Royer, and S. Das, "Ad-Hoc On-Demand Distance Vector (AODV) Routing," In *Proceedings of IEEE WMCSA '99*, New Orleans, LA, Feb 1999.
[11] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks," In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, Vol.1, 2004.
[12] Z. Shelby, S. Chakrabarti, and E. Nordmark, "Neighbor Discovery Optimization for Low-power and Lossy Networks," draft-ietf-6lowpan-nd-13, IETF, Sep 2010.
[13] T. Narten, E. Nordmark, W. Simpson, and H. Soliman, "Neighbor Discovery for IP version 6 (IPv6)," RFC 4861, IETF, Sep 2007.
[14] K. Klues, C. M. Liang, J. Paek, R. Musaloiu-E, P. Levis, A. Terzis, and R. Govindan, "TOSThreads: Thread-safe and Non-invasive Preemption in TinyOS," In *Proceedings of SenSys*, Nov 2009.
[15] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki – A Lightweight and Flexible Operating System for Tiny Networked Sensors," In *Proceedings of EmNets*, 2004.
[16] JG. Ko, Q. Wang, T. Schmid, W. Hofer, P. Dutta, and A. Terzis, "Egs: A Cortex M3-based Mote Platform," In *Poster Proceedings of IEEE SECON*, Jun 2010.
[17] J. Hui, and JP. Vasseur, "RPL Option for Carrying RPL Information in Data-Plane Datagrams," draft-ietf-6man-rpl-option-00, IETF, Jul 2010.
[18] J. Hui, JP. Vasseur, and D. Culler, "An IPv6 Routing Header for Source Routes with RPL," draft-ietf-6man-rpl-routing-header-00, IETF, Jul 2010.
[19] ATmega128L Datasheet, ATMEL.
[20] P. Thubert, "RPL Objective Function 0," draft-ietf-roll-of0-03, IETF, Jul 2010.
[21] Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection Tree Protocol," in *Proceedings of SenSys*, Nov 2009.
[22] Z. Shelby, B. Frank, and D. Sturek, "Constrained Application Protocol (CoAP)," draft-ietf-core-coap-03, IETF, Oct 2010.
[23] K. Srinivasan, and P. Levis, "RSSI is Under Appreciated," in *Proceedings of the 3rd Workshop on Embedded Networked Sensors*, 2006.
[24] D. S. J. D. Couto, D. Aguayo, J Bicket, and R. Morris, "A High-Throughput Path Metric for Multi-hop Wireless Routing," in *Proceedings of the ACM MOBICOM*, 2003.
[25] P. Levis, A. Tavakoli, and S. Dawson-Haggerty, 'Overview of Existing Routing Protocols for Low Power and Lossy Networks,' draft-ietf-roll-protocols-survey-07, IETF, 2009.
[26] M. Durvy, J. Abeille, P. Wetterwald, C. O'Flynn, B. Leverett, E. Gnoske, M. Vidales, G. Mulligan, N. Tsiftes, N. Finne, and A. Dunkels, 'Making Sensor Networks IPv6 Ready,' in *Proceedings of the ACM SenSys*, 2008
[27] R. Fielding, '*Architectural styles and the design of network-based software architectures*,' Unpublished doctoral dissertation, University of California, Irvine, 2000.
[28] X. Jiang, S. Dawson-Haggerty, and D. Culler, 'sMAP: simple monitoring and actuation profile,' in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, 2010.
[29] J. Hoglund, N. Finne, J. Eriksson, N. Tsiftes, A. Dunkels, C. Chauvenet, M. Pouillot, P.E. Goudet, B. Tourancheau, and D. Genon-Catalot, 'Interconnecting Low-power Wireless and Power-Line Communications using IPv6,' in *Proceedings of the 2nd BuildSys Workshop*, 2010.
[30] CTP: Collection Tree Protocol, http://sing.stanford.edu/gnawali/ctp
[31] R. Kling, R. Adler, J. Huang, V. Hummel, and L. Nachman, 'Intel Mote: using Bluetooth in sensor networks,' in *Proceedings of the 2nd international conference on Embedded networked sensor systems (SenSys)*, 2004.
[32] BLIP, the Berkeley Low-power IP stack: http://docs.tinyos.net/index.php/BLIP_Tutorial
[33] MICAz Datasheet, MEMSIC Corporation, http://www.memsic.com
[34] HBE-Zigbex Datasheet, Hanback Electronics, Ltd., http://www.hanback.co.kr
[35] Nano24, Octacomm Corporation, http://www.octacomm.net