Evaluation of Local Pipelining for Reprogramming Wireless Sensor Networks

Hiroshi Mineno, Takuya Miyamaru[†],
Yoshiaki Terashima[§], Yuichi Tokunaga[§] and Tadanori Mizuno[‡]

Faculty of Informatics, Shizuoka University
3-5-1 Johoku, Naka-ku, Hamamatsu, 432-8011, Japan
mineno@inf.shizuoka.ac.jp
[†]Presently, Nagoya Works, Mitsubishi Electric Corporation
5-1-14 Yadaminami, Higashi-ku, Nagoya, 461-8670, Japan
Miyamaru.Takuya@ap.MitsubishiElectric.co.jp
[§]Information Technology R&D Center, Mitsubishi Electric Corporation
5-1-1 Oofuna, Kamakura, 247-8501, Japan
Terashima.Yoshiaki@eb.MitsubishiElectric.co.jp
Tokunaga.Yuichi@ds.MitsubishiElectric.co.jp
[‡]Graduate School of Informatics, Shizuoka University
3-5-1 Johoku, Naka-ku, Hamamatsu, 432-8011, Japan
mizuno@inf.shizuoka.ac.jp

Abstract -
Wireless reprogramming is a useful service for wireless sensor networks to upload new code and modify functions. The latest reprogramming protocols use the technique called pipelining. Although it can accelerate the speed of code distribution, it requires a lot of control packets which affects energy efficiency and reliability. Improving energy efficiency and reliability is an important challenges in reprogramming. In this paper, we present a technique for code distribution called Local Pipelining. Local pipelining assigns a number of segments to a group that consists of a neighborhood. The number of segments is based on the number of control packets and the speed of code distribution. By adjusting this value according to remaining energy that group has, Local pipelining can control the amount of control packets and improve energy efficiency and reliability in the entire network.

Keywords: Reprogramming, Wireless sensor networks

1 Introduction

The recent advances in MEMS and low power wireless communication technology have led to the development of wireless sensor networks (WSN). A WSN consists of a number of sensor nodes, and they collect and transfer sensing data to the network autonomously. Many WSN applications, which including environmental monitoring, security, and position tracking, have been developed.

In WSNs, reprogramming that updates code on sensor nodes is one of the most important services. Because WSNs are a relatively new field of study, many applications contain developing technologies (ad-hoc routings, data processing, position estimations, etc.), and these technologies are implemented as specific code on the sensor nodes. It is therefore possible that these codes will be modified or extended in the future for long-running applications using WSNs. Thus, a method to easily reprogram many deployed sensor nodes is necessary. Recently, much research on wireless reprogramming has been conducted[1][2]. Wireless reprogramming distributes new code easily to a lot of sensor nodes using wireless multihop communication. The purpose of general protocols in WSN is to aggregate a lot of small data from the edge nodes to the base station, whereas the purpose of wireless reprogramming protocols is to distribute large data from the base station to the edge nodes[3][4][5][6]. The pipelining method, which quickly distributes bulk data to the entire network has been proposed in some studies[7][8][9]. In pipelining, code is divided into several segments, which are transferred in parallel. By dividing code into smaller segments, we can increase the degree of parallelism and speed up the distribution. However, the number of control packets is also increased, and this results in higher energy consumption and lower reliability.

Here, we present the code distribution technique called local pipelining, in which a number of segments is assigned to a group consisting of several sensor nodes. Normal pipelining fixes the number of segments as one value in the entire sensor network. So if we set a large value for the number of segments in order to speed up updating, this causes an increase in the number of control packets. In contrast, the local pipelining scheme can adjust the number of control packets depending on the condition of each group. First, we present a method of local pipelining that can freely adjust the number of segments and control packets depending on the remaining energy. Adjusting this value for each group contributes to improved energy efficiency and transfer efficiency. Second, we analyze the case of several pipelines that have a different number of segments. This is helpful in cases where we have to reprogram various multiple networks.

This paper is organized as follows. In section 2, we explain some issues related to pipelining and analyze the control packets needed in the transfer process. An overview of local

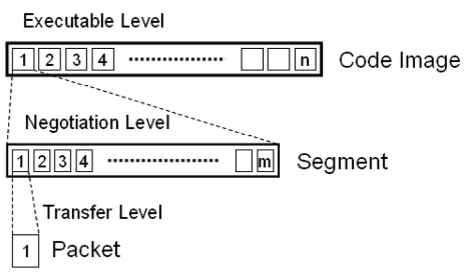


Figure 1: Structure of segment

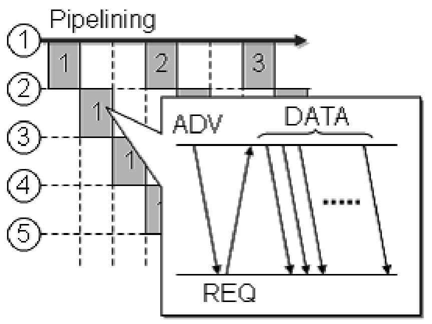


Figure 2: Handshake

pipelining and its features is introduced in section 3, which also describes the transfer algorithm. We describe the performance of local pipelining using several formulas in section 4, and evaluate it in section 5. Also included in section 5 is a description of the implementation of local pipelining on TinyOS [11]. This evaluation includes a simulation of the number of sending packets, completion times, and the propagation process. Finally, section 6 summarizes the paper and mentions future work.

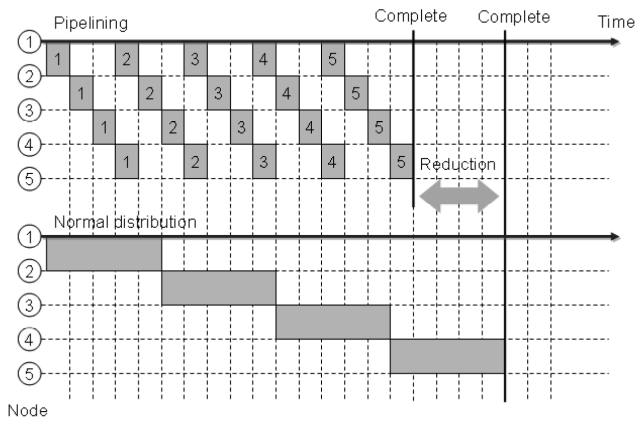


Figure 3: Pipelining

2 Related Issues

2.1 Pipelining

Many wireless reprogramming protocols share design challenges. We deal here with the three important challenges that follow [1].

- **Reliability:** The complete code must be correctly received by the target nodes, and the downloaded code must be executed correctly on the sensor node.
- **Energy efficiency:** The energy used in reprogramming is provided by the sensor node battery. This battery also supplies energy for sensing, which is the primary role. Thus, the energy consumption for reprogramming should be reduced as much as possible.
- **Completion times:** The completion time of reprogramming affects the sevice using a WSN. When we reprogram the network, we have to stop service and wait until the update is completed. Therefore we have to minimize the completion time of reprogramming.

Pipelining is proposed as a means to speed up distribution despite these challenges. In pipelining, code is divided into several segments, as depicted in Figure 1, and each segment consists of several packets, which form a transfer unit. Figure 3 shows how distribution can be sped up by overlapping the transferring segments. The figure compares the process of pipelining with normal distribution. There are five sensor nodes deployed linearly in Figure 3. In the pipelining scheme, while node 4 is transferring segment 1 to node 5, node 1 is transferring segment 2 to node 2 simultaneously. The result is that pipelining can complete downloading earlier than normal distribution. Thus, we can reduce the completion time by overlapping the segments. In this case, we need at least three hops spaced between segments to avoid the hidden terminal problem.

2.2 Negotiation Scheme

Because pipelining deals with several segments, it is necessary to keep track of segments that are lacking. Therefore, a negotiation scheme is needed to request missing segments. This scheme uses a three-way handshake that has three types of messages (Advertise, Request, Data). This scheme is proposed to reduce message redundancy by SPIN [10]. The epidemic property is important since WSNs experience high loss rates, asymmetric connectivity, and transient links due to node failures and repopulation. The latest reprogramming protocol uses this scheme to improve reliability. Figure 2 illustrates the three-way handshake. First, a source node advertises an ADV message, which includes its own segment, to neighboring nodes. Second, if the destination node receives the ADV message, it compares its own segment with the received segment information and decides whether it needs the segment advertised by the source node. If it needs this segment, it requests the segment to source node by sending an REQ message. Finally, if the source node receives the REQ message

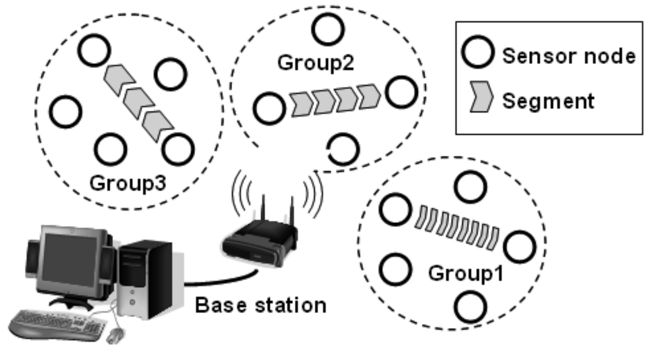


Figure 4: Local pipelining

from the destination node, it starts to forward a DATA message that consists of sequential packets. By using this three-way handshake, we can reduce the redundancy of transferred segments.

2.3 Energy issue in pipelining

In pipelining, we can accelerate the speed of code distribution by dividing code into smaller segments (increasing the number of divided segments) and increasing the degree of parallel. However, we cannot increase the number of segments without reason, because, this can affect the energy efficiency and reliability, depending on the control packets. Pipelining uses one negotiation per segment, and one negotiation requires control packets that include ADV and REQ messages. Thus, if we increase the number of segments, the ADV and REQ messages in corresponding segments will also increase. This affects the energy efficiency and reliability. First, a lot of energy is used to send messages, it is one of the most energy-consuming actions in the sensor node. The number of messages greatly affects energy efficiency. Second, when many messages are sent, message collisions may occur.

For these reasons, it is necessary to reduce or adjust the control messages required for pipelining.

3 Local pipelining

3.1 Overview

The goal of local pipelining is to freely adjust the number of control packets depending on the condition of each group. Normal pipelining, which is used by Deluge [7] and MNP [8] fix the number of segments as one value in the entire network. In this case, if we set a large value for the number of segments in order to speed up processing, this causes an increase in the number of control packets.

The amount of remaining energy varies depending on which nodes are deployed. Therefore, some sensor nodes bear a large burden of control packets. However, we cannot decrease the number of segments to fall in step with the subset of sensor nodes, because this degrades performance. Therefore, we propose a local pipelining to achieve a realistic distribution for each node while maintaining good performance.

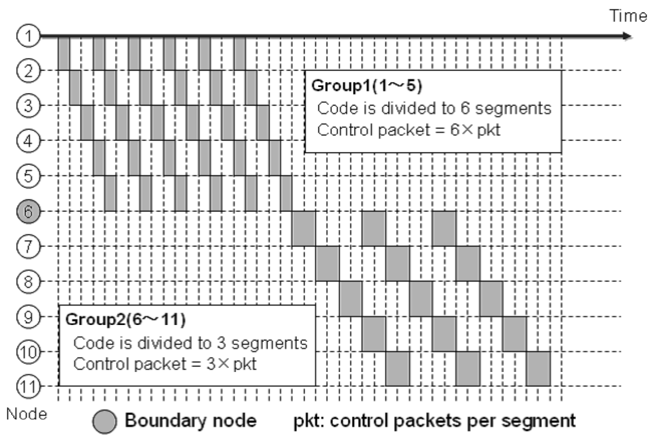


Figure 5: Distribution of Local pipelining

In local pipelining, the number of segments is not assigned to an entire network, but rather, a value is assigned to each group, which is a set of neighboring nodes, as shown in Figure 4. It is only necessary to set the value of each group depending on the available of energy. In this way, we can achieve the following setting. If there is some leeway in the energy, we can divide code into smaller segments for faster distribution or into larger segments to constrain the number of control packets.

3.2 Distribution Process

Figure 5 shows the distribution process of local pipelining. In this figure, we assume that group 1 includes node 1 to node 5, group 2 has node 6 to node 11, and group 1 divides code into six segments, and group 2 divides code into three segments. If one pkt control packet is needed for each segment, group 1 needs $6 \times pkt$ and group 2 needs $3 \times pkt$.

Because local pipelining deals with several networks each of which has a particular number of segments, it is necessary to transfer data by using a distribution technique that is different from normal pipelining. This process involves changing the number of segments and retransmitting new segments in the transfer. This process is handled by a boundary node deployed at the edge of a group. Node 6 is the boundary node in Figure 5. The boundary node must wait until all segments from other groups have been received, and when the download is completed, it starts forwarding with its own number of segments. That is to say, the boundary node has the same role as a base station.

3.3 Transfer Algorithm

To achieve distribution like this, we extended the negotiation scheme described in section 2. First, we added a field that includes information about a group to ADV and DATA messages. Group information can be determined depending on the amount of remaining energy and other information. A corresponding group table containing group information and the amount of remaining energy is established in advance. For

Table 1: Example of group table

Group	Remaining energy
1	80% - 100%
2	60% - 80%
3	40% - 60%
4	20% - 40%
5	0% - 20%

```
[ When a segment is received. ]

IF ownGroup == rcvMsg.group
        /* send segment as the same number of segment */
        GOTO Advertise phase
ELSE
        IF downloadComplete == TRUE
                /* send segment as the new number of segment */
                setOwnSegmentSize()
                GOTO Advertise phase
        ELSE
                /* wait until all segment data is complete */
                GOTO Receive phase
        ENDIF
ENDIF
```

Figure 6: Transfer algorithm

example, if the remaining energy is in the range from 80% to 100%, it corresponds to group X. If the remaining energy is in the 60% to 80% range, it corresponds to group Y. By making a corresponding table like Table 1, we have the option of changing the information depending on the group. Second, we added the transfer algorithm shown in Figure 6. This algorithm is used when a segment is received. Received messages include the information about the group the source node belongs to. First, the node compares the group in the received message with its own group. If the received group is the same as own group, it goes to the advertise phase. In the advertise phase, the node sends ADV messages that include the received segment. This process is the same as in normal pipelining. If the received group is different from its own group, the node is a boundary node. When a boundary node receives a segment, it prohibits transferring of the segment. It must wait until all segments have been completed. This is because to reset a new number of segments it must have all the data. If all segments are downloaded, it sets its own number of segments and goes to the advertise phase. Otherwise, it stays in the receive phase.

4 Performance Analysis

4.1 Speed of local pipelining

In this section, we analyze the performance of local pipelining, normal pipelining, and cases without pipelining. In particular, we focus on completion time and the number of messages, which affect energy efficiency and speed of distribution. First, we explain the effect of normal pipelining. We

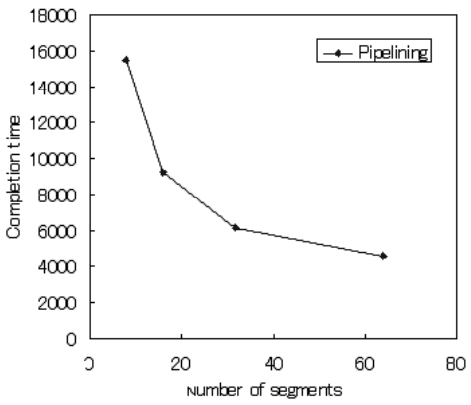


Figure 7: Completion time of normal pipelining

assume a linear deployment as in Figure 3, and the network size is n hop. Then we try to forward the static size code image, which is divided into m segments, and where t times is needed to send one segment. To avoid the hidden terminal problem, we need at least three hops between segments. Although handshake is supposed in our proposed local pipelining scheme to overcome this problem, we need at least three hops between segments to avoid the hidden terminal problem completely in the case of linear deployment as shown in Figure 3. Therefore, completion time T_p is expressed as follows.

$$T_p = (n + 3(m-1)) \cdot t \tag{1}$$

In the case without pipelining, completion time T_n is expressed as follows.

$$T_n = n \cdot m \cdot t \tag{2}$$

These formulas conduce the result in Figure 7. This result is the theoretical completion time of normal pipelining, and it is assumed that n is 100 hops, and the time until all data sending has ended is 1024. For example, if $m = 16$, the completion time of normal pipelining is $(100 + 3(16 - 1)) \cdot 1024/16 = 9280$. This figure is obvious proof of the relationship between the number of segments and completion time. This relationship means that as we increase the number of segments, we can accelerate the speed of distribution. In contrast to pipelining, the speed of distribution in cases without pipelining is slow. In this condition, the theoretical completion time is fixed as $T_n = 100 \cdot 1 \cdot 1024 = 102400$. If n and m are large enough, it is obvious that $T_p < T_n$.

Second, we express the effect of local pipelining. Local pipelining involves several networks that have a different number of segments depending on the group. When the boundary node receives the segment of another group, it waits until all data is complete, and retransmits its own number of segments. Therefore, the completion time of local pipelining can be expressed as the sum of the completion time for each group. Then, there are k groups in the linear network, each group network size is n_i, they have m_i segments, and they need time t_i to send one segment. The theoretical completion time of local pipelining T_{lp} is expressed as the following.

$$T_{lp} = \sum_{i=1}^{k}(n_i + 3(m_i - 1)) \cdot t_i \tag{3}$$

In each group, $T_p < T_n$ is approved. Therefore, the sum of completion time T_{lp} is less than the case of not using pipelining. This means that local pipelining is superior to not using pipelining in completion time, but it is inferior to normal pipelining.

4.2 Control packets

Next, we describe the control packets needed in normal pipelining and local pipelining. First, we introduce the control packets per segment. A segment has two types of control messages, ADV messages and REQ messages. These messages are not necessarily one message. If a source node advertises an ADV message and receives no requests for it, it needs to retransmit the ADV message. By the same reason, if a destination node send an REQ message to a source node and the DATA does not arrive, it needs to retransmit the REQ message. At this point, we assume the number of ADV messages per segment as N_{adv}, and the number of REQ messages per segment as N_{req}. One segment needs $N_{adv} + N_{req}$ control packets. Thus, one node needs $m \cdot (N_{adv} + N_{req})$ control packets. In normal pipelining, the sum of control packets in an entire network is as follows.

$$C_p = n \cdot m \cdot (N_{adv} + N_{req}) \tag{4}$$

In contrast, local pipelining has several m, and several k groups which have n_i sensor nodes. The sum of control packets in an entire network is as follows.

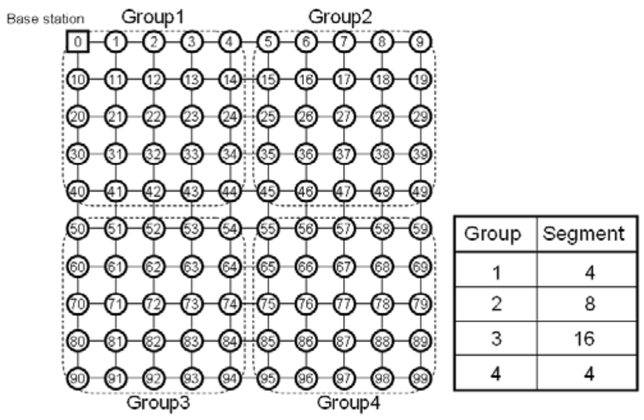
$$C_{lp} = \sum_{i=1}^{k} n_i \cdot m_i \cdot (N_{adv} + N_{req}) \tag{5}$$

These formulas show that local pipelining can freely adjust and reduce the number of control packets depending on circumstances while maintaining the speed of distribution. For example, there are four groups in an entire network, and each group has five nodes. Each group is assigned a number of segments as follows. Group 1 has 16, group 2 has 8, group 3 has 16, and group 4 has 8. In this case, $C_{lp} = 48 \cdot 5 \cdot (N_{adv} + N_{req}) = 240 \cdot (N_{adv} + N_{req})$. In contrast, the case of normal pipelining, m is fixed as 16. Therefore, $C_n = 20 \cdot 16 \cdot (N_{adv} + N_{req}) = 320 \cdot (N_{adv} + N_{req})$. It is obvious that $C_p > C_{lp}$.

5 Evaluation

5.1 Simulation Environments

In this section, we describe an evaluation of local pipelining using the TinyOS network simulator (TOSSIM [12]). The goal of this simulation was to prove that local pipelining is superior to normal pipelining from the view point of energy efficiency, and that it can transfer data without problems.



Group	Segment
1	4
2	8
3	16
4	4

Figure 8: Sensor node deployment

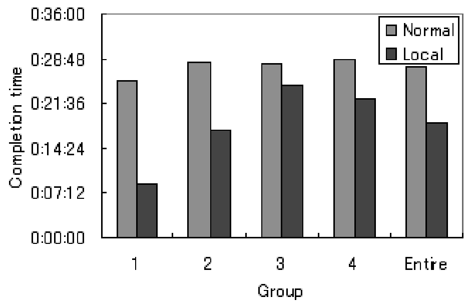


Figure 9: Average completion time of each node

First, we describe the implementation of local pipelining on TinyOS. The implementation was based on MNP [8] which is a state-of-the-art reprogramming protocol and includes normal pipelining. We extended the function of MNP's control packets described in section 3, and added a transfer algorithm. In this implementation, we had to be careful about the group arrangement. The groups were arranged as in Figure 8 without regard for the remaining energy. Because TOSSIM cannot duplicate the sensor node battery, so we assume that groups are determined depending on location, as shown in the Figure.

Next, we explain the simulation environment. We assumed each node had a transmission radius of 50 feet (meaning that nodes can receive messages within a 50-foot radius). Nodes were deployed in a reticular pattern (Figure 8), and each node had 40 feet of spacing. The network had four groups that had 5 × 5 subnetworks, and each group had the number of segments indicated in Figure 8. We assumed that normal pipelining (MNP) had a fixed number of segments, where the value was 16 divisions. The base station had a complete code image, and started forwarding each segment in the early stage of distribution.

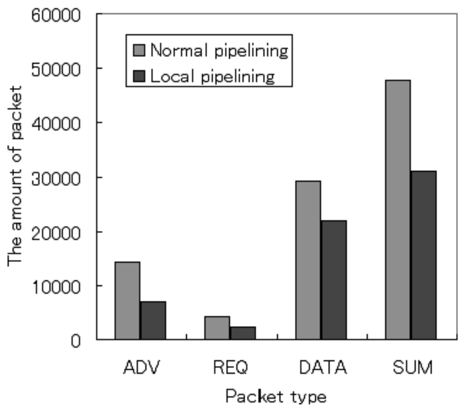


Figure 11: Number of packets

5.2 Completion times and propagation

In this section, we investigate the distribution speed between local pipelining and normal pipelining. Figure 9 shows the average completion time of each node under simulation, classified by group. Because the number of segments is fixed in normal pipelining, the completion time for each group has about the same value. In contrast, local pipelining has various numbers of segments, which affects the completion times. In this environment, this result indicates that local pipelining can distribute code faster than normal pipelining.

Figure 10 illustrates the propagation process of segments under simulation, when the node has downloaded all segments. In this figure, normal pipelining has equable propagation, where each node is received almost at the same time. This is because the same number of segments is used in the entire network. On the other hand, local pipelining is inequable propagation. The borders of the groups bring about the delay. This delay is caused by the network's waiting until all segments have been downloaded completely. There are especially large delays in the boundary nodes placed where that the difference in the segment numbers is very large.

5.3 The number of messages

Next, we evaluate the energy efficiency depending on the number of messages. Figure 11 plots the number of the messages in the entire network, with a comparison between normal (MNP) and local pipelining. It is clear that local pipelining requires fewer messages than normal pipelining. ADV, and REQ are control messages, and DATA includes segment data (but also includes start-download and terminate-download messages in MNP). Control packets depend on the number of segments, as explained in section 4. Therefore, local pipelining, which has different numbers of segments, is effective in adjusting the number of control packets. In linear deployment, which uses the same parameters, control packets are indicated by $(4+8+16+4)\cdot 25\cdot (N_{adv}+N_{req}) = 800(N_{adv}+N_{req})$ in local pipelining. On the contrary, normal pipelining requires $16\cdot 100\cdot (N_{adv}+N_{req}) = 1600(N_{adv}+N_{req})$ control packets. Therefore, the number of control packets in local pipelining is smaller than in normal pipelining.

6 Summary and future works

In this paper, we presented our local pipelining technique, which can freely adjust several numbers of segments corresponding to groups. By adjusting this parameter, we can reduce the number of control messages depending on the circumstances. This method improves energy efficiency, because sending messages is one of the actions that consumes the most energy. To verify the effectiveness of local pipelining, we evaluated it using the TOSSIM simulator. In this simulation, we mainly evaluated the number of messages and the completion time. We found that local pipelining can reduce the number of messages, and the average completion time of each node is shorter than in the case of normal pipelining. This means that local pipelining achieved a partial improvement in energy efficiency.

Our future work is as follows. First, we will try to reduce the delay of local pipelining. At the group border, some delays occur, which are the waiting times for all segment downloading to be completed. This design is easy to implement and ensures correct distribution. The delay can be improved by transferring the received segments incompleted. If a node receives enough segments so that we can change the number of segments, we can transfer the received segments with own number of segment divisions. Our second task in the future is to study other metrics affected by adjusting the number of segments. In this paper, the metric considered was the remaining energy. However, we believe that local pipelining can improve other metrics (hardware richness, condition of communication, etc). Thirdly, we have to evaluate our proposed local pipelining performance under more realistic conditions, such as several topologies and groups as well as ideal grid condition. Although there are some assumptions in our proposed local pipelining yet, we will study more useful reprogramming wireless sensor networks method based on this work.

REFERENCES

[1] Qiang Wang, Yaoyao Zhu, Liang Cheng, ”Reprogramming wireless sensor networks: challenges and approaches,” IEEE Network, vol.20, no.3, pp.48-55 (2006).

[2] Pedro J. Marron, Andreas Lachenmann, Daniel Minder, Matthias Gauger, Olga Saukh, Kurt Rothermel, ”Management and configuration issues for sensor networks,” Journal of Network Management, vol.15, pp.235-253 (2005).

[3] Crossbow Technology, Inc., ”Mote In-Network Programming User Reference,” Mica2 Wireless Measurement System Datasheet, 2003.

[4] Thanos Stathopoulos, John Heidemann, Deborah Estrin, ”A Remote Code Update Mechanism for Wireless Sensor Networks,” CENS Technical Report 30 (2003).

[5] Philip Levis, Neil Patel, David Culler, Scott Shenker,

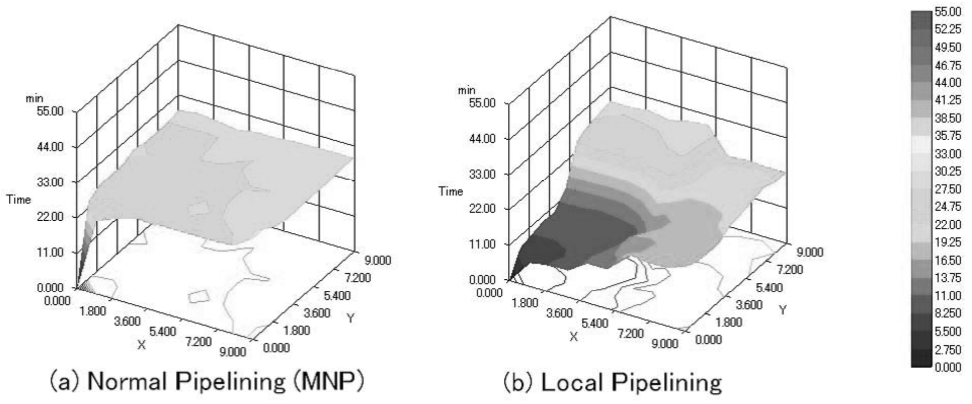


Figure 10: Propagation

"Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks," In proc. 1st Symp. Networked Sys. Design and Implementation, pp. 15-28 (2004).

[6] Vinayak Naik, Anish Arora, Prasun Sinha, Hongwei Zhang, "Sprinkler: A Reliable and Energy Efficient Data Dissemination Service for Extreme Scale Wireless Networks of Embedded Devices," IEEE Trans. on Mobile COmputing, vol.6, no.7, pp.777-789 (2007).

[7] Jonathan W. Hui, David Culler, "The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale," In proc. ACM SenSys 2004, pp.81-94 (2004).

[8] S. S. Kulkarni, Mimin Wang, "MNP: Multihop Network Reprogramming Service for Sensor Networks," In proc. IEEE ICDCS, pp.7-16 (2005).

[9] Lane Phillips, "Aqueduct: Robust and Efficient Code Propagation in Heterogeneous Wireless Sensor Networks," In Master's thesis, Univ. of Colorado at Boulder (2005).

[10] Joanna Kulik, Wendi Heinzelman, Hari Balakrishnan, "Negotiation-Based Protocols for Disseminating Information in Wireless Sensor Networks," Wireless Networks, vol.8, no.2/3, pp.169-185 (2002).

[11] TinyOS: http://www.tinyos.net/

[12] Simulationg TinyOS Networks: http://www.cs.berkeley.edu/ pal/research/tossim.html

(Received September 9, 2007)
(Revised November 4, 2008)

Hiroshi Mineno received his B.E. and M.E. degrees from Shizuoka University, Japan in 1997 and 1999, respectively. In 2006, he received the Ph.D. degree in Information Science and Electrical Engineering from Kyushu University, Japan. Between 1999 and 2002 he was a researcher in the NTT Service Integration Laboratories. In 2002, he joined the Department of Computer Science of Shizuoka University as an Assistant Professor. His research interests include sensor networks as well as heterogeneous network convergence. He is a member of IEEE, ACM, IEICE, IPSJ and Informatics Society.

Takuya Miyamaru received his B.I. and M.I. degree from Shizuoka University in 2006 and 2008, respectively. His research interests included reprogramming wireless sensor networks. Presently, he is a engineer of Nagoya Works, Mitsubishi Electric Corporation, Japan. Currently he has been engaged in developments for compact controller for factory automation systems. He is a member of Information Processing Society of Japan.

Yoshiaki Terashima received the B.S. degree from Saitama University in 1984 and the Ph.D. degree from Shizuoka University in 2005. He is a head researcher in Information Technology R & D Center, Mitsubishi Electric Corporation, Japan. He is currently leading the Disaster Rescue Network Project that realizes network-sensor data fusion based on ad-hoc disaster rescue network. His research interests include distributed processing architecture and testing methodologies for ad-hoc network. He is a member of Information Processing Society of Japan and the Institute of Electronics, Information and Communication Engineers.

Yuichi Tokunaga received the B.E. degree from Tokyo University of Science in 1990. He is a head researcher in Information Technology R & D Center, Mitsubishi Electric Corporation, Japan. He is currently leading the development of the diagnostic and monitoring system that uses wireless sensor networks. His research interests include time synchronization and positioning methodologies for wireless sensor networks. He is a member of Information Processing Society of Japan.

Tadanori Mizuno received the B.E. degree in industrial engineering from the Nagoya Institute of Technology in 1968 and received the Ph.D. degree in computer science from Kyushu University, Japan, in 1987. In 1968, he joined Mitsubishi Electric Corp. Since 1993, he is a Professor of Shizuoka University, Japan. Now, he is a Professor of graduate school of Science and technology of Shizuoka University. His research interests include mobile computing, distributed computing, computer networks, broadcast communication and computing, and protocol engineering. He is a member of Information Processing Society of Japan, the institute of electronics, information and Communication Engineers, the IEEE Computer Society, ACM and Informatics Society.