DRAFT Version

An Automatic Software Decentralization Framework for Distributed Device Collaboration

Yohei Iwasaki and Nobuo Kawaguchi
Graduate School of Engineering, Nagoya University
Furo-cho Chikusa-ku Nagoya 464-8603, Japan

Abstract

Recently, many low-profile devices with RF communication features have been developed such as sensor network nodes and one-chip microcomputers. This caused a variety of gadgets join to the wireless network. However, in order to realize collaborations between these devices, generally we have to develop complicated distributed software. In this paper, we propose an automatic software decentralization method, which converts a stand-alone software program into distributed software programs. In order to execute the generated software on these low-profile devices, we employ the programming language nesC for a decentralization target. We also have implemented applications with real hardware devices to exemplify that the proposed method successfully improves the ease of development.

1 Introduction

Recently, many low-profile devices with RF communication features have been developed such as sensor network nodes and one-chip microcomputers. This caused a variety of gadgets such as sensors, audio-visual appliances, PC peripherals, and even a pen[1] join to the wireless network. Figure 1 shows example low-profile devices, which are Berkeley MOTE[6] and Nordic nRF24E1[14] devices. We wish these devices collaborated with each other via the network, provided more advanced application services which assist our daily life. However, in order to realize the device collaborations, generally we have to develop the complicated distributed software programs.

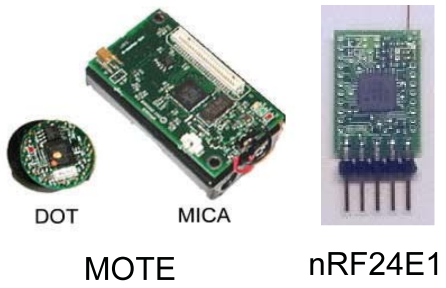


Figure 1. Low-profile Devices

In this paper, we propose an automatic software decentralization method, which converts stand-alone software into distributed software. Some software decentralization methods are already proposed, such as J-Orchestra[16] and Addistant[15]. However, these methods are targeting the JavaVM with a RPC (Remote Procedure Call) service, which is too large for the low-profile devices. In order to execute the generated software on these low-profile devices, we employ the programming language nesC[4] for a decentralization target. While conventional RPC communicates at each function call, our proposed method only communicates when an execution node of a control-flow is changed. This often reduces the number of communications. We also have implemented applications with real hardware devices to exemplify the usefulness of the proposed method.

2 Device Collaboration Frameworks

We categorized device collaboration frameworks into the two types, which are *centrally controlled collaboration* and *distributed collaboration* shown in Figure 2.

In the centrally controlled collaboration, device-specific functions in the devices are centrally controlled by the remote control node. This remote control is realized by a RPC (Remote Procedure Call) framework, such as Universal Plug and Play[10] and Jini[18]. Thus the application software which implements the device collaboration is single stand-alone software, which can be easily developed.

On the other hand, in the distributed collaboration framework such as Touch-and-Connect[7] and AMIDEN[11], each device has application-specific functions (*application protocol*). Devices directly communicate with each other to realize the device collaboration. Compared with the

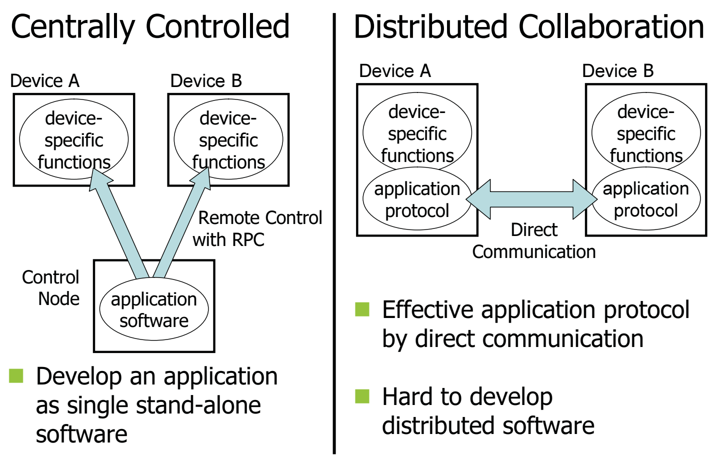

Centrally Controlled
Device A
Device B
device-specific functions
device-specific functions
Remote Control with RPC
Control Node
application software
Develop an application as single stand-alone software
Distributed Collaboration
Device A
Device B
device-specific functions
application protocol
device-specific functions
application protocol
Direct Communication
Effective application protocol by direct communication
Hard to develop distributed software

Figure 2. Device Collaboration Frameworks

centrally controlled collaboration, the distributed collaboration can employ more effective application protocol because each device has application-specific functions, and does not need the remote control node. However, generally it is hard to develop distributed software which implements effective application protocol.

Our software decentralization method aims to automatically generate this distributed application software from stand-alone application software in the centrally controlled collaboration. This means we can receive advantages of both device collaboration types.

Bischoff et al. proposed a similar approach[2], which automatically generates distributed device applications from the high-level application definition. They develop particular rule-based language for the application definition. Our approach uses the same procedure-oriented language as conventional manner to write an application program, which might be widely acceptable to application programmers.

3 Automatic Software Decentralization Algorithm

In this section, we describe detailed steps of our automatic software decentralization algorithm, which convert a stand-alone software program into distributed software programs.

Figure 3 shows an example which applies this algorithm to a simple program. The index numbers of the algorithm's steps shown below correspond to the index numbers in the figure.

Here, a *node* means the terminal where the generated distributed software will be executed.

(1) Input a Source Program: First, a source program is inputted, which is written as stand-alone software. The uses of node specific features are described as function calls. Additionally, *a function placement* is inputted, which indicates correspondence between a function and its execution node.

An example source program is shown in Figure 3 (1), which represents device collaboration between a Sensor-node (which can sense temperature and battery voltage) and a LCD-node (a display device which can show characters).

The program read sensor measurements (temperature and battery voltage), put them on the LCD, and show "LOW BATTERY" message when the battery voltage is less than the specific threshold (4800). The functions placement is that the functions that start with "`Sensor.`" should be executed on the Sensor-node, and the functions that start with "`Lcd.`" should be executed on the LCD-node.

The algorithm splits this program into two distributed software, where one can be executed on the Sensor-node and the other on the LCD-node.

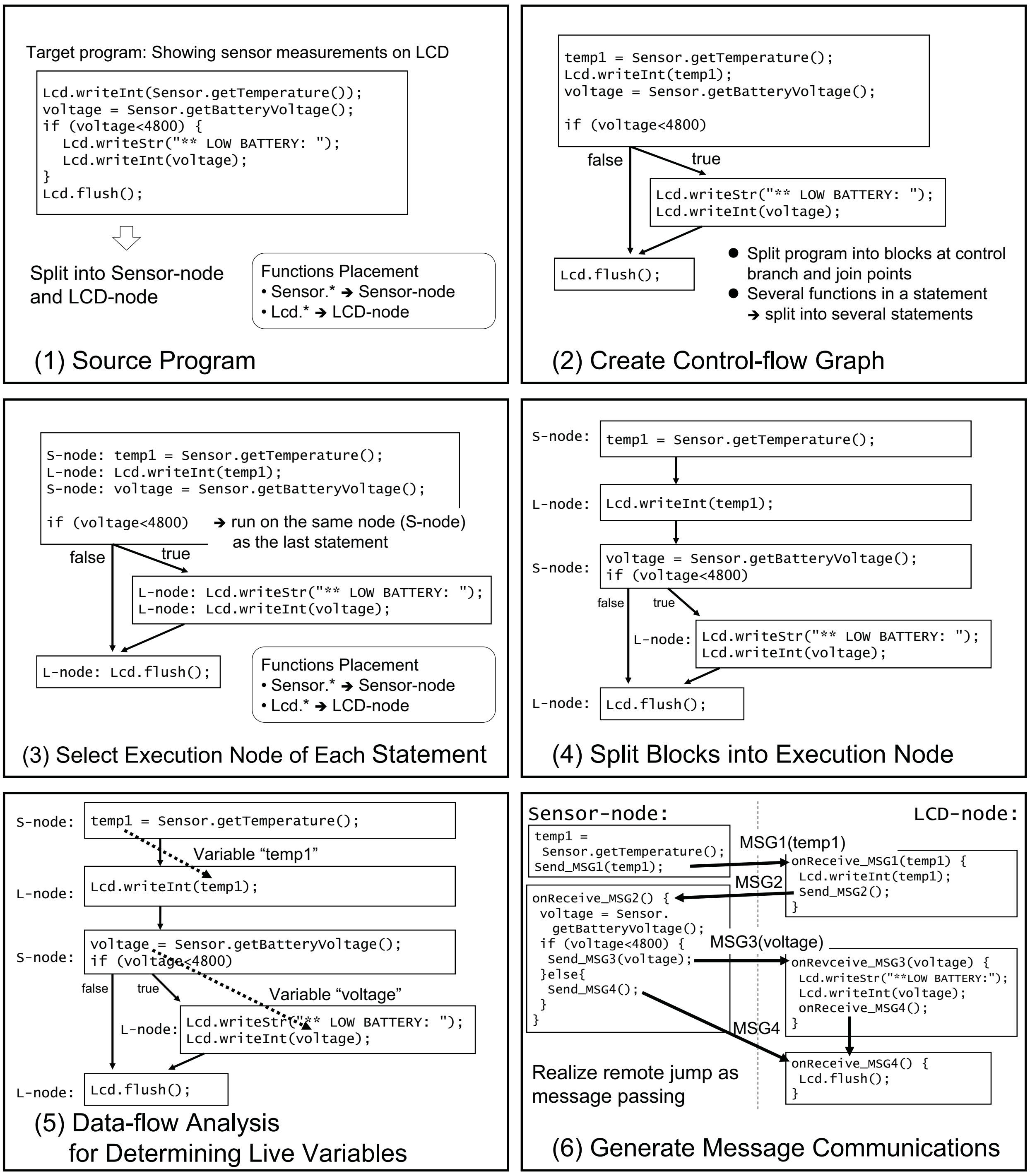
(2) Create Control-flow Graph: The algorithm creates a control-flow graph[12] from the inputted source program. The control-flow graph includes blocks which are the sequence of several statements, and control flows (represents as arrows in the figure) between these blocks.

In this step, the statement which contains several function calls is split into several statements by using temporary variables, because the algorithm places each statement on a single node. For example, the statement:

```
Lcd.writeInt(Sensor.getTemperature());
```

is converted to the two statements by using the temporary variable `temp1` as follows:

```
temp1=Sensor.getTemperature();
Lcd.writeInt(temp1);
```


Target program: Showing sensor measurements on LCD

```
Lcd.writeInt(Sensor.getTemperature());
voltage = Sensor.getBatteryVoltage();
if (voltage<4800) {
  Lcd.writeStr("** LOW BATTERY: ");
  Lcd.writeInt(voltage);
}
Lcd.flush();
```

Split into Sensor-node and LCD-node
Functions Placement
• Sensor.* ➔ Sensor-node
• Lcd.* ➔ LCD-node
(1) Source Program

```
temp1 = Sensor.getTemperature();
Lcd.writeInt(temp1);
voltage = Sensor.getBatteryVoltage();

if (voltage<4800)
```

```
Lcd.writeStr("** LOW BATTERY: ");
Lcd.writeInt(voltage);
```

```
Lcd.flush();
```

● Split program into blocks at control branch and join points
● Several functions in a statement ➔ split into several statements
(2) Create Control-flow Graph

```
S-node: temp1 = Sensor.getTemperature();
L-node: Lcd.writeInt(temp1);
S-node: voltage = Sensor.getBatteryVoltage();

if (voltage<4800)
```

```
L-node: Lcd.writeStr("** LOW BATTERY: ");
L-node: Lcd.writeInt(voltage);
```

```
L-node: Lcd.flush();
```

```
S-node: temp1 = Sensor.getTemperature();
L-node: Lcd.writeInt(temp1);
S-node: voltage = Sensor.getBatteryVoltage();
        if (voltage<4800)
```

```
L-node: Lcd.writeStr("** LOW BATTERY: ");
        Lcd.writeInt(voltage);
L-node: Lcd.flush();
```

```
S-node: temp1 = Sensor.getTemperature();
L-node: Lcd.writeInt(temp1);
S-node: voltage = Sensor.getBatteryVoltage();
        if (voltage<4800)
```

```
L-node: Lcd.writeStr("** LOW BATTERY: ");
        Lcd.writeInt(voltage);
L-node: Lcd.flush();
```

Sensor-node:

```
temp1 =
 Sensor.getTemperature();
Send_MSG1(temp1);
```

```
onReceive_MSG2() {
 voltage = Sensor.
   getBatteryVoltage();
 if (voltage<4800) {
  Send_MSG3(voltage);
 }else{
  Send_MSG4();
 }
}
```

LCD-node:

```
onReceive_MSG1(temp1) {
 Lcd.writeInt(temp1);
 Send_MSG2();
}
```

```
onRevceive_MSG3(voltage) {
 Lcd.writeStr("**LOW BATTERY:");
 Lcd.writeInt(voltage);
 onReceive_MSG4();
}
```

```
onReceive_MSG4() {
 Lcd.flush();
}
```

MSG1(temp1)
MSG2
MSG3(voltage)
MSG4
Realize remote jump as message passing
(6) Generate Message Communications

Figure 3. Applying Example of The Software Decentralization Algorithm

(3) Select Execution Node of Each Statement: The algorithm selects a execution node of each statement (placement of statements). The statement that contains a call for a node-specific function will be executed on the node where the function exists. This is determined according to the inputted functions placement. The placement of the other statements is a challenging problem. In the current implementation, they will be executed on the same node as the last statement.

Figure 3 (3) shows the example placement, where the S-node means the Sensor-node, and the L-node means the LCD-node.

(4) Split Blocks into Execution Node: The algorithm splits a block (sequence of statements) at the point where the execution node of a statement is changed. As a result, each block contains only statements executed on a single node, which means the execution node of the block is determined (placement of blocks).

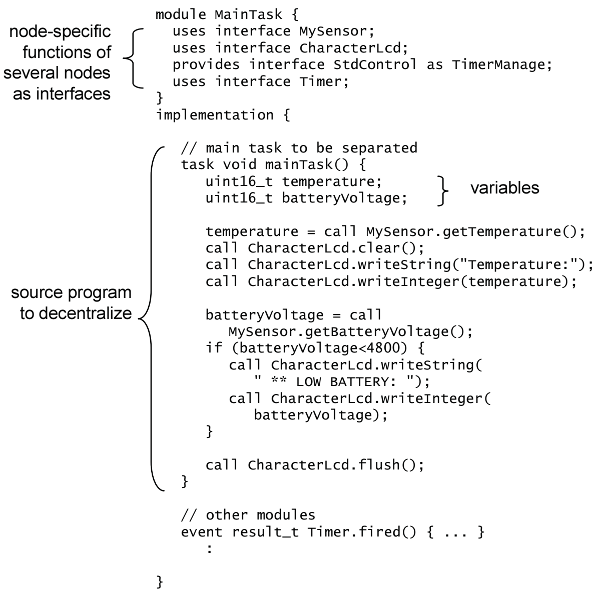
(5) Data-flow Analysis for Determining Live Variables: The algorithm analyzes the data that should be passed between the blocks by data-flow analysis. Here, we use the *live variables*[12] as the data that should be passed. The live variables mean that whose value at the block's entrance will be used later.

(6) Generate Message Communications: The algorithm implements a remote jump (control-flow to a remote node) as a message passing. The message type is created corresponding to the block. The live variables of the block are passed as the message's arguments, in order to synchronize the value of these variables.

The software decentralization is completed by the steps mentioned above. In the example at the Figure 3 (6), the number of times of message passing is three, which are MSG_1, MSG_2, and MSG_3 or MSG_4. This result shows that the proposed algorithm reduced the number of times of message passing compared with the RPC, because two messages (a request and a reply) will be passed for each function call in the RPC.

4 Implementation for nesC Language

We have implemented a prototype system of our proposed method to exemplify its feasibility. We employ the nesC[4] as a source and target program language for the decentralization, because we think generated software should be executed on low-profile devices. The nesC is the extended C language, which targets low-profile devices such as MOTE[6], and employs the component model aiming at



```
module MainTask {
  uses interface MySensor;
  uses interface CharacterLcd;
  provides interface StdControl as TimerManage;
  uses interface Timer;
}
implementation {

  // main task to be separated
  task void mainTask() {
     uint16_t temperature;
     uint16_t batteryVoltage;

     temperature = call MySensor.getTemperature();
     call CharacterLcd.clear();
     call CharacterLcd.writeString("Temperature:");
     call CharacterLcd.writeInteger(temperature);

     batteryVoltage = call
        MySensor.getBatteryVoltage();
     if (batteryVoltage<4800) {
        call CharacterLcd.writeString(
           " ** LOW BATTERY: ");
        call CharacterLcd.writeInteger(
           batteryVoltage);
     }

     call CharacterLcd.flush();
  }

  // other modules
  event result_t Timer.fired() { ... }
     :

}
```

node-specific functions of several nodes as interfaces
variables
source program to decentralize

Figure 4. Source Program (digest)

a static optimization. Note that our decentralization algorithm itself will not be executed on the low-profile devices, thus we have implemented the algorithm on Java 2 SE 5.0, with the nesC parser TinyDT[8].

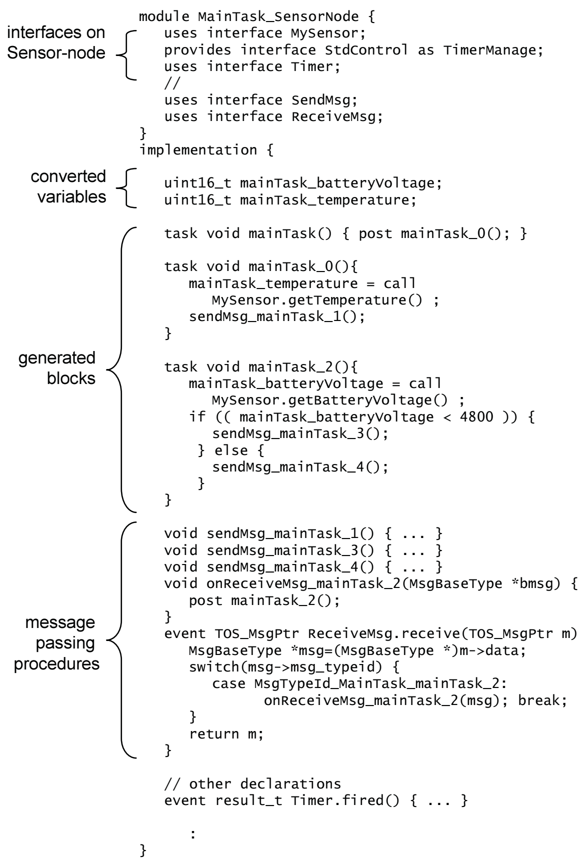
First, we targeted the TinyOS[4] platform, which is the standard API libraries for the nesC. An example source program is shown in Figure 4, which is the same behavior as the example in Section 3 and is decentralized into Sensor-node and LCD-node. The generated programs are shown in Figure 5 (Sensor-node) and Figure 6 (LCD-node).

Node-specific functions are represented as *interfaces* of nesC. Additionally, a user inputs which node each interface exists. In the above example, MySensor, Timer, and TimerManage interfaces are on the Sensor-node, and CharacterLcd interface is on the LCD-node. (See Section 4.1)

A source program to decentralize should be written as a *task function* in a nesC module. This is because the task function is executed asynchronously at a system idle period, which means the call stack is empty. This is convenient so that the system keep the local variables while executing the decentralized programs.

In generated programs, the local variables of the source function are converted to module variables. Each generated block (sequence of statements) is converted as a new task function. Message sending and receiving process also is inserted, which use SendMsg and ReceiveMsg interfaces in the TinyOS API.

Table 1 shows the size of the source and generated programs of the example above. Note that this shows line and byte counts of a nesC module file (or sum of two generated nesC module files) without any comments, blank lines, and


interfaces on Sensor-node
converted variables
generated blocks
message passing procedures

```
module MainTask_SensorNode {
   uses interface MySensor;
   provides interface StdControl as TimerManage;
   uses interface Timer;
   //
   uses interface SendMsg;
   uses interface ReceiveMsg;
}
implementation {

   uint16_t mainTask_batteryVoltage;
   uint16_t mainTask_temperature;

   task void mainTask() { post mainTask_0(); }

   task void mainTask_0(){
      mainTask_temperature = call
         MySensor.getTemperature() ;
      sendMsg_mainTask_1();
   }

   task void mainTask_2(){
      mainTask_batteryVoltage = call
         MySensor.getBatteryVoltage() ;
      if (( mainTask_batteryVoltage < 4800 )) {
         sendMsg_mainTask_3();
       } else {
         sendMsg_mainTask_4();
       }
   }

   void sendMsg_mainTask_1() { ... }
   void sendMsg_mainTask_3() { ... }
   void sendMsg_mainTask_4() { ... }
   void onReceiveMsg_mainTask_2(MsgBaseType *bmsg) {
      post mainTask_2();
   }
   event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr m) {
      MsgBaseType *msg=(MsgBaseType *)m->data;
      switch(msg->msg_typeid) {
         case MsgTypeId_MainTask_mainTask_2:
               onReceiveMsg_mainTask_2(msg); break;
      }
      return m;
   }

   // other declarations
   event result_t Timer.fired() { ... }

      :
}
```

Figure 5. Generated Program for Sensor-node (digest)

Table 1. Size of the Source and Generated Programs

	Line Counts	Bytes
Source Program	35	913
Generated Programs	146	4531

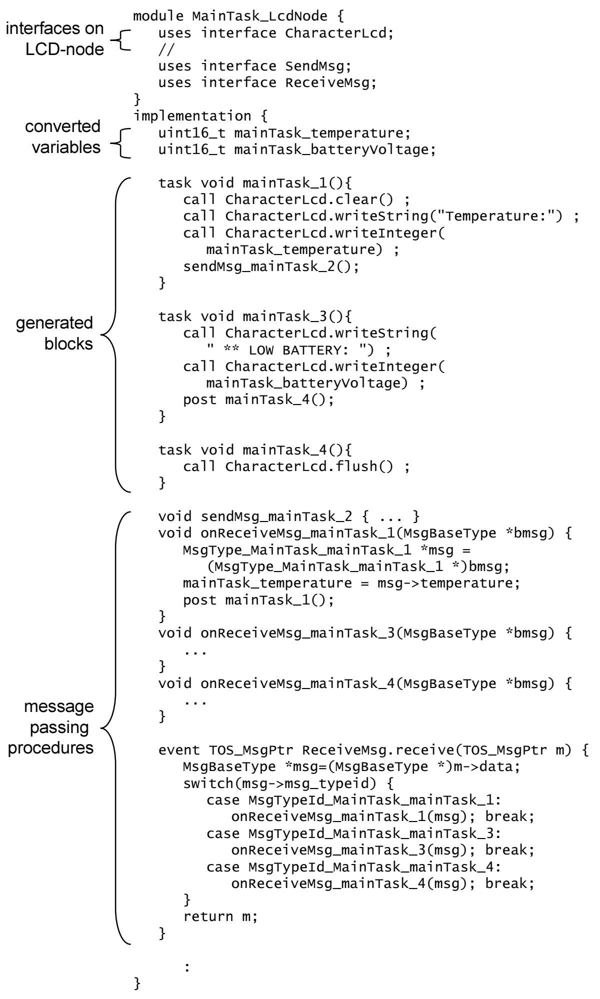
redundant white spaces. This result shows that implementation of distributed software needs large programs and our proposed method successfully reduces the program size that we have to write.

We have confirmed the execution of the generated programs on the EmTOS[5], which is a TinyOS emulator on Linux.

4.1 Decentralization of Configuration File

Our prototype system also supports decentralization of a nesC configuration file[4]. A nesC configuration file represents constituent components (modules) of application software, and links between the components, as shown in Figure 7 (1). The user also inputs component placement information written in JavaScript, which represents which node each component will be placed in generated distributed software, as shown in Figure 7 (2).

Our prototype system infers function placement (See section 3) from these information, and automatically generates nesC configuration files for generated distributed software. The execution node of a function (interface) is where its connected component exists. Component diagrams of source and generated configuration files are shown in Figure 8. Note that MainForTimer and MainForLcd components manage application initialization, and GenericComm component manages RF communication.


interfaces on LCD-node
converted variables
generated blocks
message passing procedures

```
module MainTask_LcdNode {
   uses interface CharacterLcd;
   //
   uses interface SendMsg;
   uses interface ReceiveMsg;
}
implementation {
   uint16_t mainTask_temperature;
   uint16_t mainTask_batteryVoltage;

   task void mainTask_1(){
      call CharacterLcd.clear() ;
      call CharacterLcd.writeString("Temperature:") ;
      call CharacterLcd.writeInteger(
         mainTask_temperature) ;
      sendMsg_mainTask_2();
   }

   task void mainTask_3(){
      call CharacterLcd.writeString(
         " ** LOW BATTERY: ") ;
      call CharacterLcd.writeInteger(
         mainTask_batteryVoltage) ;
      post mainTask_4();
   }

   task void mainTask_4(){
      call CharacterLcd.flush() ;
   }

   void sendMsg_mainTask_2 { ... }
   void onReceiveMsg_mainTask_1(MsgBaseType *bmsg) {
      MsgType_MainTask_mainTask_1 *msg =
         (MsgType_MainTask_mainTask_1 *)bmsg;
      mainTask_temperature = msg->temperature;
      post mainTask_1();
   }
   void onReceiveMsg_mainTask_3(MsgBaseType *bmsg) {
      ...
   }
   void onReceiveMsg_mainTask_4(MsgBaseType *bmsg) {
      ...
   }

   event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr m) {
      MsgBaseType *msg=(MsgBaseType *)m->data;
      switch(msg->msg_typeid) {
         case MsgTypeId_MainTask_mainTask_1:
            onReceiveMsg_mainTask_1(msg); break;
         case MsgTypeId_MainTask_mainTask_3:
            onReceiveMsg_mainTask_3(msg); break;
         case MsgTypeId_MainTask_mainTask_4:
            onReceiveMsg_mainTask_4(msg); break;
      }
      return m;
   }

      :
}
```

Figure 6. Generated Program for LCD-node (digest)

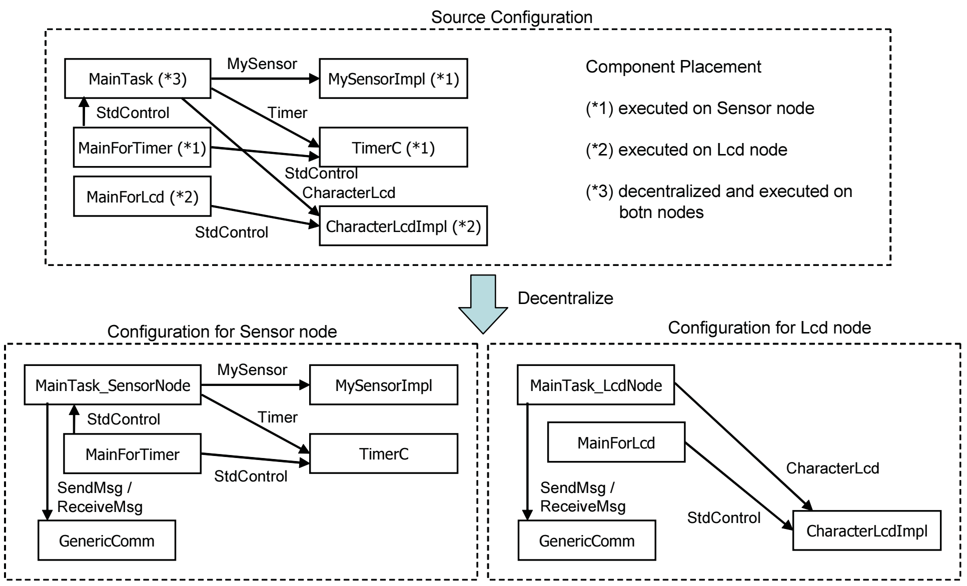


Figure 8. Component Diagrams

(1) nesC Configuration File:

```
configuration AppC {
} implementation {
   components MainTask; // to be separated
   components MySensorImpl, TimerC, CharacterLcdImpl;
   components Main as MainForTimer, Main as MainForLcd;

   MainTask.Timer -> TimerC.Timer[unique("Timer")];
   MainTask.MySensor -> MySensorImpl.MySensor;
   MainTask.CharacterLcd -> CharacterLcdImpl.CharacterLcd;

   MainForTimer.StdControl -> MainTask.TimerManage;
   MainForTimer.StdControl -> TimerC.StdControl;
   MainForLcd.StdControl -> CharacterLcdImpl.StdControl;
}
```

(2) Component Placement Information :

```
places.separateModules.add("MainTask");

p=new Place("SensorNode","0x01");
   p.components.add("TimerC");
   p.components.add("MySensorImpl");
   p.components.add("MainForTimer");
places.add(p);

p=new Place("LcdNode","0x02");
   p.components.add("CharacterLcdImpl");
   p.components.add("MainForLcd");
places.add(p);
```

Figure 7. Configuration File

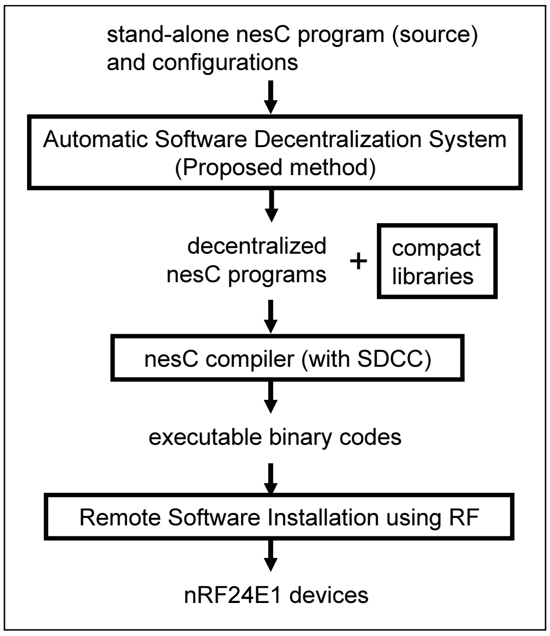

stand-alone nesC program (source) and configurations
Automatic Software Decentralization System (Proposed method)
decentralized nesC programs
+
compact libraries
nesC compiler (with SDCC)
executable binary codes
Remote Software Installation using RF
nRF24E1 devices

Figure 9. Automatic Deployment Flow

4.2 The nRF24E1 Device

Then, we have applied our proposed method to real low-profile hardware devices. We targeted the nRF24E1[14] device, which is Intel 8051 compatible one-chip microcomputer with RF communication function. This device is small and low-cost (a chip itself without circuit board costs 5 dollars), but very low-profile (8bits CPU, 4K bytes memory including code and data).

There is a project that implements TinyOS for 8051 platform[17]. However, we think the standard TinyOS API is large for this low-profile device, thus we developed more compact API and libraries for the nRF24E1 device, including Timer, RF communication, AD converter, and so on.

For easy software deployment, we also develop a software remote installation protocol via RF communication.

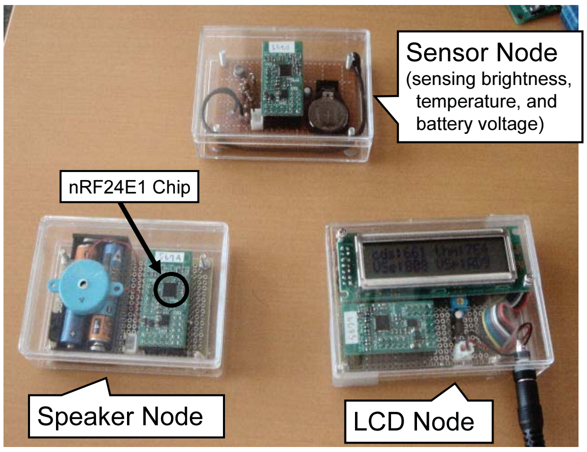


Figure 10. Prototype Devices

Figure 9 shows the data flow of our system. First, a stand-alone source program written in nesC is decentralized into distributed programs by our proposed method, then they are compiled to executable binary codes by the nesC compiler with SDCC[13] as a back-end, finally they are remotely installed into the target devices via RF communication.

We have developed some application hardware devices shown in Figure 10. The example device collaboration is the morning call service. The Sensor-node continuously senses the outdoor brightness. The LCD-node shows the brightness degree. And the Speaker-node beeps when the brightness exceeds the threshold, which means sunrise in the morning. All you have to do is just writing a single stand-alone program which implements the scenario. The system automatically generates the distributed software which does not require the control node, and installs them into the target devices. This shows the proposed method successfully improves the ease of development.

5 Conclusion

In this paper, we have proposed an automatic software decentralization method, which convert a stand-alone software program into distributed software programs. In order to execute the generated software on the low-profile devices such as sensor network nodes and one-chip microcomputers, we employ the programming language nesC[4] for a decentralization target. We also have implemented applications with real hardware devices to exemplify the usefulness of our proposed method.

There are a number of areas that need further investigation.

- The usefulness of our proposed method should be evaluated quantitatively with more practical applications.
- Generated distributed programs should be optimized by using more advanced data-flow analysis and parallelism between statements obtained by such as Program Dependency Graph[3].
- Currently generated software is deployed from a rich device to low-profile devices. Maté[9] realize dynamic software deployment directly between sensor-nodes by using tiny virtual machines. We are also planning dynamic software migration between low-profile devices by implementing custom hardware microprocessors on FPGAs, which may be more low-profile and cost-effective than using virtual machines.
- Extend source language specification in order to describe collaboration toward an array of multiple sensor nodes easily.
- Software decentralization has many choices, ambiguity and trade-offs. For example, code optimization policies, which nodes an application should be executed, and which application can be executed on devices here. It is desirable that the system infers best default choice, or offers several choices to users.
- Exception handling should be considered in the circumstances such as a communication error.

Acknowledgement

This research is partly supported by "Strategic Information and Communications R&D Promotion Programme (SCOPE)" from Ministry of Internal Affairs and Communications, Japan.

References

[1] Anoto Group AB. Anoto Digital Pen and Paper. `http://www.anoto.com/`.

[2] U. Bischoff and G. Kortuem. Programming the ubiquitous network: A top-down approach. In *System Support for Ubiquitous Computing Workshop (UbiSys 2006)*, 2006.

[3] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[4] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of Programming Language Design and Implementation (PLDI)*, 2003.

[5] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer. A System for Simulation, Emulation, and Deployment of Heterogeneous Sensor Networks. In *ACM Conference on Embedded Networked Sensor Systems (SenSys 2004)*, 2004.

[6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 93–104, 2000.

[7] Y. Iwasaki, N. Kawaguchi, and Y. Inagaki. Touch-and-Connect: A connection request framework for ad-hoc networks and the pervasive computing environment. In *First IEEE Annual Conference on Pervasive Computing and Communications (PerCom 2003)*, pages 20–29, March 2003.

[8] Janos Sallai et al. TinyDT - TinyOS Plugin for the Eclipse Platform. `http://www.tinydt.net/`.

[9] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, 2002.

[10] Microsoft Corporation. Universal Plug and Play Device Architecture. `http://www.upnp.org/resources/`.

[11] M. Minoh and T. Kamae. Networked Appliances and Their Peer to Peer Architecture AMIDEN. *IEEE Communications Magazine*, 39(10):80–84, 2001.

[12] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[13] Sandeep Dutta et al. SDCC - Small Device C Compiler. `http://sdcc.sourceforge.net/`.

[14] N. Semiconductor. nRF24E1 2.4 GHz Radio Transceiver with Microcontroller. `http://www.nvlsi.no/`.

[15] M. Tatsubori, S. Chiba, and K. Itano. Addistant: An Aspect-Oriented Distributed Programming Helper (in Japanese). *IPSJ Transactions on Programming*, 43(SIG03):17–25, March 2002.

[16] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. In *European Conference on Object-Oriented Programming (ECOOP)*, June 2002.

[17] TinyOS 8051 Working Group. `http://www.tinyos.net/scoop/special/working_group_tinyos_8051`.

[18] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, 1999.