分散協調型連携を実現するソフトウェアの細粒度自動分散化手法

岩崎陽平† 河口信夫‡

† 名古屋大学大学院情報科学研究科 ‡ 名古屋大学情報連携基盤センター

1 はじめに

近年，MOTE[1] や nRF24E1[2] など，無線通信機能を搭載した低コストな端末が開発されつつあり，センサ・家電・文房具などの様々な小型機器がネットワークに参加しつつある．これらの機器がネットワーク経由で連携して高度なアプリケーションサービスを実現し，我々の生活を支援することが期待される．しかし，機器間の連携のためのプログラムコードは複数の機器に分散するため，従来のソフトウェア開発手法では，連携ソフトウェアの開発や保守が困難となり，また機器生産時に定められたアプリケーションプロトコルの範囲内でしか連携が行えない．例えば，我々は以前，無線ワンチップマイコン nRF24E1[2] を用いて，機器間連携が可能な図 1 のようなプロトタイプデバイスを試作したが [3]，連携に関する機能が複数の端末に分散しているため，連携ソフトウェアの開発や保守が煩雑となった．

本稿では，機器間の連携ソフトウェアを，プログラムコードの分散を意識せずに機器生産後に容易に開発・導入できるようにするために，連携ソフトウェアの自動分散化手法を提案する．特に，ワンチップマイコンなどの低レベルデバイス上で，端末間が直接通信を行う分散協調型 (P2P 型) の連携機構に着目する．提案手法では，機器間の連携機能を，連携ソフトウェアの分散を意識せず単一のソフトウェアモジュールとして記述する．この連携ソフトウェアを自動的に分散化し，連携に参加する機器にインストールする．

ソフトウェアの自動分散化手法としては，既に J-Orchestra[4]，Addistant[5]，Jacross[6] などが提案されているが，これらの手法は，クラス単位・メソッド単位で分散化を行い，RPC（遠隔手続き呼び出し）により相互通信を行うため，遠隔の機能（メソッド）を呼び出すたびに往復通信が発生し，通信回数が多くなってしまう．また，これらの手法は一般的に JavaVM を対象としたものであり，JavaVM を動作させることが困難な低レベルデバイスへの適用は難しい．

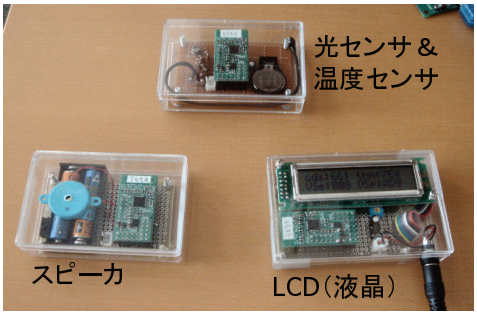

図 1: 直接通信で連携できる低レベルデバイス

提案手法では，プログラムの処理やデータの流れを解析することにより，クラスやメソッドより細かい単位（細粒度）である文（ステートメント）単位での分散化を行うため，通信効率の向上が期待できる．また，低レベルデバイス向けの拡張 C 言語である nesC[7] のモジュールを対象として提案手法を実装し，その実現可能性を示した．

2 集中制御型連携と分散協調型連携

機器間連携のための基盤フレームワークは，主に集中制御型と分散協調型 (P2P 型) に分類される．Universal Plug and Play[8]，Jini[9] などの集中制御型フレームワークでは，機器の機能をネットワークへ公開し，単一の連携ソフトウェアが遠隔の機器を集中制御して連携を実現する．一方，Touch-and-Connect[10]，AMIDEN[11] などの分散協調型フレームワークでは，各機器が連携に特化した連携機能を持ち，機器間が直接通信して連携を実現する．

本研究では，分散協調型フレームワークに着目する．この分散協調型フレームワークは，別途制御ノードを必要としない，特定の連携に特化した通信効率の良いアプリケーションプロトコルを使える，などの利点がある一方，連携に関するプログラムコードが複数端末間に分散し，その開発や保守は容易ではない．

ここで，連携の具体例として，CD プレーヤの音量を，ディスプレイに音量メータとして表示する連携を考える．図 2 は集中制御型の連携を，図 3 は分散協調型の連携を示したものである．集中制御型の連携では，制御ノードを介して，音声データと映像データを送受信している．一方，分散協調型の連携では，音声＋映像データに比べてより軽量な，音量データのみを送受信するアプリケーションプロトコルを用いて連携を行うことができ，また連携時に制御ノードが不要である．しかし，連携に関するプログラムコードが，音声の音量を計算する部分

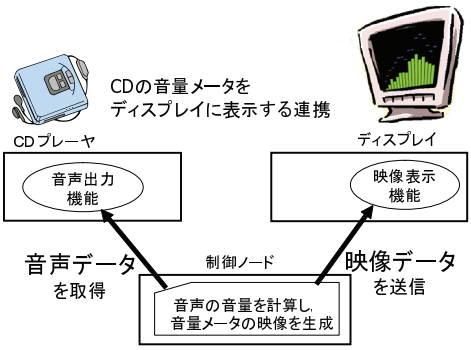

図 2: 集中制御型連携

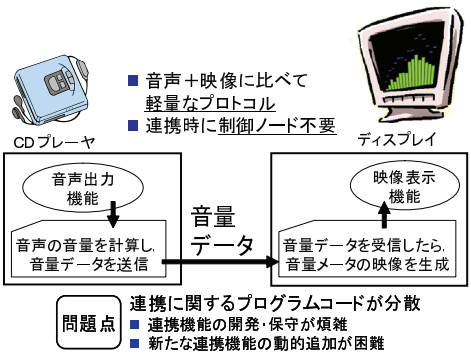

図 3: 分散協調型連携

と，音声メータの映像を生成する部分に分割され，また音量データの送受信処理も記述しなければならず，ソフトウェアの開発や保守が煩雑となる．また，予めアプリケーションプロトコルを定め，連携機能を組み込んで機器を開発するため，機器生産後に連携の種類を動的に追加することが困難となる．

2.1 連携ソフトウェアの自動分散化

このような問題点を解決するために，本手法では，図 4 のように複数のノードに分散する連携機能を抽出し，開発時には単一のソフトウェアモジュールとして扱う．この連携ソフトウェアは，単一端末用のソフトウェアと同じように記述すればよく，容易に開発することが出来る．記述されたソフトウェアは，連携に参加する端末の機能に合わせて連携設定時に自動的に分散化され，各端末へインストールされる．すなわち，本フレームワークでは，集中制御型の連携ソフトウェアを，自動的に分散協調型の連携ソフトウェアに変換することにより，開発時には集中制御型の利点（単一モジュールで連携機能を開発可能）を得られ，実行時には分散協調型の利点（通信効率の良いアプリケーションプロトコルで直接通信可能）を得られる．

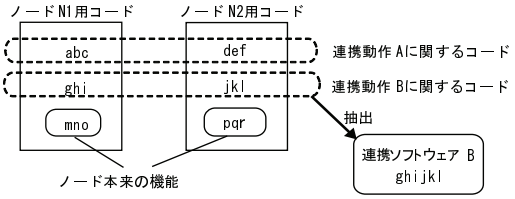

図 4: 複数の端末に分散する連携機構

3 連携ソフトウェアの細粒度自動分散化アルゴリズム

本節では，単一端末の場合と同様の形式で書かれた連携ソフトウェアを，連携に参加する各端末のモジュール構成（どの機能をどの端末上で実行するべきかを示したもの）に従い，自動的に分散化する手法について述べる．文（ステートメント）単位での細粒度の分散化を行うため，文献 [4][5] 等の，RPC(遠隔手続き呼び出し) を用いるクラス単位の分散化手法と比較して，通信効率の良いアプリケーションプロトコルを生成できる．

分散化のアルゴリズムを以下に示す．また，このアルゴリズムを実際のプログラム例に適用した場合を図 5 に示す（図中の番号がアルゴリズムの各手順に対応）．

(1) 入力　分散化の対象となるプログラムを入力する．この際，各端末のモジュール構成（どの機能をどの端末上で実行するべきかを示したもの）も入力する．端末固有の機能の利用は，関数呼び出しとして実現される．

図 5 の (1) の例では，センサ (sensor) より取得した温度 (temperature) の値を画面 (LCD) に表示し，またセンサより取得した電池電圧 (battery voltage) の値が，特定の閾値 (4800) よりも小さい場合は,「LOW BATTERY」というメッセージを画面に表示する，という連携ソフトウェアのプログラムを考える．このプログラムを，センサから値を取得する機能（先頭に Sensor. とある関数）を持つセンサ側端末と，画面に値を出力する機能（先頭に Lcd. とある関数）を持つ画面側端末に分散化する．

(2) 制御フローグラフの作成　入力されたプログラムより制御フローグラフ [12] を作成する．制御フローグラフは，複数の文（ステートメント）をまとめたブロックと，ブロック間の処理の流れを表すものである．この際，1 つの文の中に複数の関数呼び出しを含むものは，一時変数を用いて複数の文に分解する．

(3) 各文の実行端末を決定　各文を実行する端末を決定する．特定の端末の機能（関数）を使う文は，その端末上で実行する．そうでないものは，何らかの方法によって実行する端末を決定する必要がある．ここでは，直前に出現した文と同じ端末とした．

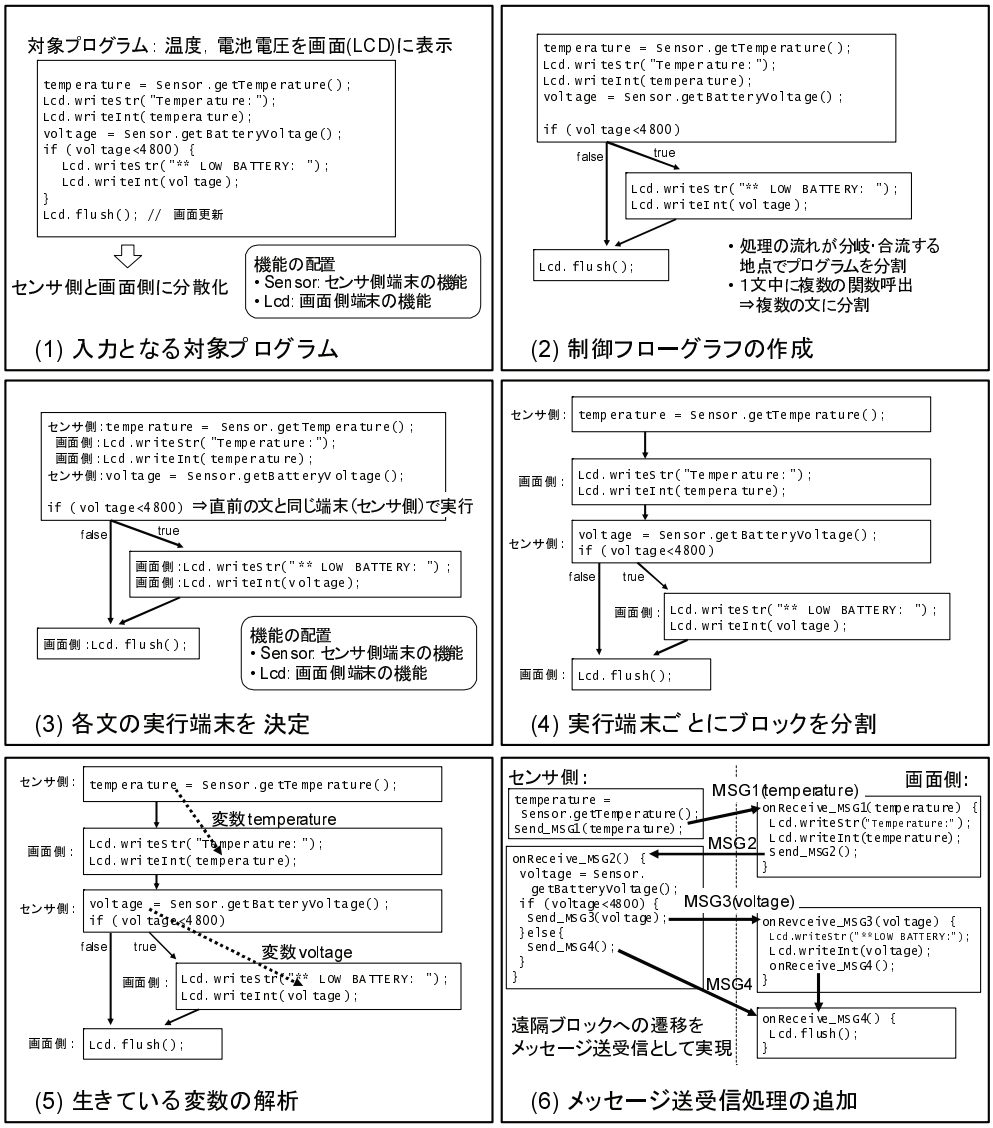

図 5: 自動分散化アルゴリズムの適用例

(4) 実行端末ごとにブロックを分割　各文を実行する端末が切り替わる点で，ブロックを分割する．これにより，各ブロックは単一の端末で実行される文のみを含むため，そのブロックを実行する端末が決まる．

(5) 生きている変数の解析　ブロック間で受け渡すべきデータを調べるために，生きている変数 [12] の解析を行う．生きている変数とは，ブロック入り口での値が今後使用される可能性のある変数のことである．

(6) メッセージ送受信処理の追加　遠隔ブロックへの遷移を，メッセージの送受信処理として実現する．メッセージの種類は遷移先ブロックごとに作成し，移動元端末と移動先端末で変数の値を同期させるために，生きている変数を引数として渡す．以上により分散化が完了する．

図 5 の (6) の例では，メッセージの送受信は MSG_1, MSG_2, MSG_3 または MSG_4, の 3 回のみであり，遠隔の関数を呼び出すごとに往復通信が発生する RPC を利用する場合と比較して，通信回数を削減できていることが分かる．

3.1 nesC のモジュールを対象とした自動分散化手法の実装

低レベルデバイス向けに拡張された C 言語である nesC[7] のモジュールを，自動分散化の対象となる連携ソフトウェアの記述言語，および分散化の結果出力されるターゲットプログラムの記述言語とし，提案手法の実装を行った．実装は，Java2 SE 5.0，および nesC のパーサである TinyDT[13] を用いて行った．

3 節に挙げた分割前の連携ソフトウェアの例を，nesC[7] のモジュールとして記述したものを図 7 に示す．MySensor, TimerManage, Timer インタフェースは，センサ端

```
※メッセージ構造体
typedef struct {
   uint16_t msg_typeid; // メッセージ種別
   [メッセージ引数の宣言]
      : (引数を列挙)
} MsgType_[メッセージ名];
   : (メッセージを列挙)

typedef struct { uint16_t msg_typeid; } MsgBaseType; // 基底構造体
enum { MsgTypeId_[メッセージ名]=[ユニーク値], … (メッセージを列挙) }; // メッセージ種別

※メッセージ送信関数
void sendMsg_[メッセージ名]() {
   MsgType_[メッセージ名] *msg=(MsgType_[メッセージ名] *)msg_data.data;
   msg->msg_typeid=MsgTypeId_[メッセージ名];
   msg->[メッセージ引数名]=[引き渡す変数名];
      : (引数を列挙)
   call SendMsg.send([送信先アドレス], sizeof([メッセージ名]_t), &msg_data);
}

※メッセージ受信ハンドラ
void onReceiveMsg_[メッセージ名](MsgBaseType *bmsg) {
   MsgType_[メッセージ名] *msg=(MsgType_[メッセージ名] *)bmsg;
   [引き渡す変数名]=msg->[メッセージ引数名];
      : (引数を列挙)
   post [継続するタスク名]();
}
   : (メッセージを列挙)

event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr m) {
   // dispatcher
   MsgBaseType *msg=(MsgBaseType *)m->data;
   switch(msg->msg_typeid) {
      case MsgTypeId_[メッセージ名]: onReceiveMsg_[メッセージ名](msg); break;
         : (メッセージを列挙)
   }
   return m;
}
```

図 6: メッセージ送受信処理のテンプレート

```
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() { ~ }
     :

}
```

複数の端末の機能を混在して宣言

変数宣言

分割対象のタスク（複数の端末の機能を混在して使用）

図 7: 分散化前の nesC によるモジュール (一部省略)

```
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() { ~ }

      :
}
```

センサ端末側の機能を宣言

変数宣言

分割されたタスク

メッセージ送受信処理

図 8: 分散化後のセンサ端末側モジュール (一部省略)

```
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;
   }

      :
}
```

画面端末側の機能を宣言

変数宣言

分割されたタスク

メッセージ送受信処理

図 9: 分散化後の画面端末側モジュール (一部省略)

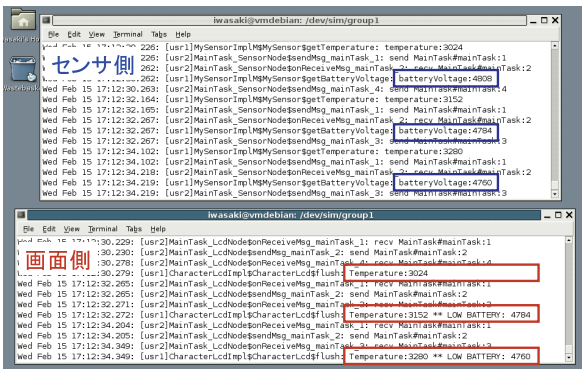

図 10: エミュレータ上での動作確認

末側の機能であり，CharacterLcd インタフェースは，画面端末側の機能である．分散化の対象となるプログラムは，タスク mainTask() であり，両端末の機能を混在させた形で自然に記述できる．各インタフェースがどの端末の機能であるか（各端末のモジュール構成）を入力として与える．

nesC のモジュールを構成する要素としては，変数，非同期に呼び出されるプライベート関数である task 関数，他モジュールから利用される関数である command 関数および event 関数がある．変数は基本的に全端末のモジュールで共有する．ただし，各端末で全く使用しない変数は除くことができる．また，スタックフレームを保存する機能は想定しないため，ローカル変数はモジュール変数へと変換する．task 関数は，3 節で提案した自動分散化手法に基づいて，細粒度に (文単位で) 各端末へと分散化される．分割された各ブロックがそれぞれ新しい task 関数として定義され，図 6 のようなコードテンプレートを元に，対応するメッセージ送受信処理が組み込まれる．command 関数および event 関数は，他モジュールからの同期呼び出しであり，スタックフレームを保存する機能無しに制御の流れを分割することが困難なため，細粒度での分散化は想定せず，単に関数全体をインタフェースが属する端末側のモジュールへと移動する．

図 7 に示す分割前のモジュールを，センサ端末側モジュールと画面端末側モジュールに分散化したものを，それぞれ図 8，図 9 に示す．分割後のこれらのモジュールは，TinyOS エミュレータである EmTOS[14] 上で，実際に連携動作することを確認した（図 10）．

4 遠隔端末へのソフトウェアのインストール

本手法では，実際に連携動作を行う低レベルデバイスであるシンプルノード（無線通信機能を持つ 8bit マイコン程度を想定）と，連携の設定やソフトウェアのビルドを行うための高機能なスーパーノード（PC，PDA，携帯電話などを想定）の 2 種類の端末を用いる．スーパーノードは設定時のみに必要であり，実際の連携動作時には必要ない．スーパーノード上では連携ソフトウェアの自動分散化機構が動作しており，ユーザが希望する連携構成に基づき，必要な連携ソフトウェアを統合して自動分散化（ノードごとに分割しアプリケーションプロトコルを生成）し，各ノード用のプログラムを生成する．連携構成が変わる度に，各ノードのプログラムコードを再コンパイルするため，静的な最適化を用いたコンパクトなコードを生成でき，低レベルデバイスに適する．

図 11 は，自動分散化により生成された各端末ごとのプログラムを，遠隔の端末（シンプルノード）にインストールする手順の詳細を示す．ここでは，2 台の端末 N1,N2 で実行する例を示しているが，1 台でも，3 台以上でもかまわない．まず，プログラムをコンパイルし，実行する端末向けのメモリイメージを生成する．実行する端末（シンプルノード）には，あらかじめ遠隔でメモリを読み書き出来る機能がインストールされている．生成されたメモリイメージを，この機能を使って実行する端末に遠隔で書き込むことにより，プログラムのインストールを行う．また，実行する端末のメモリの一部を遠隔で読み出し，書き込むメモリイメージに反映させることにより，実行する端末上の現在のデータ（変数の値等）を保存したまま，プログラムを更新することができる．メモリ上のどのアドレスの内容をどのアドレスに書き込めばよいかは，コンパイル時に生成されたメモリマップ（各データがメモリ上のどのアドレスに配置されているかを示した情報）などに基いて決定する．

5 まとめ

本稿では，プログラムコードの分散を意識せずに連携ソフトウェアを容易に開発，導入できるようにするために，連携ソフトウェアの自動的分散化手法を提案した．提案手法では，プログラムの処理やデータの流れを解析することにより，文（ステートメント）単位での細粒度の分散化を行うため，通信効率の向上が期待できる．本手法を用いることにより従来ノードごとに別々に開発していたソフトウェアを，連携動作単位で開発することができ，連携機能の容易な開発が可能となる．

今後の課題としては，以下のようなものがあげられる．

- まずは簡単なプログラムを対象にエミュレータ上での動作確認を行ったが，実デバイス上で実用的なアプリケーションを対象とした分散化を行い，本手法の有用性を評価する
- アプリケーションプロトコルのメッセージの引数として生きている変数 [12] を用いたが，何度も同じ端末に同じ値を送信するのは無駄であるため，本手法に特化した，より通信効率の良いデータフロー解析手法を検討する
- プログラムの文間の依存関係を解析する Program Dependency Graph[15] などの手法を応用し，分散化を行う際に文の実行順序を並び替え，端末間の通信回数をより少なくするような最適化を検討する
- 通信エラー等が発生した場合の対策（例外処理）を検討する必要がある

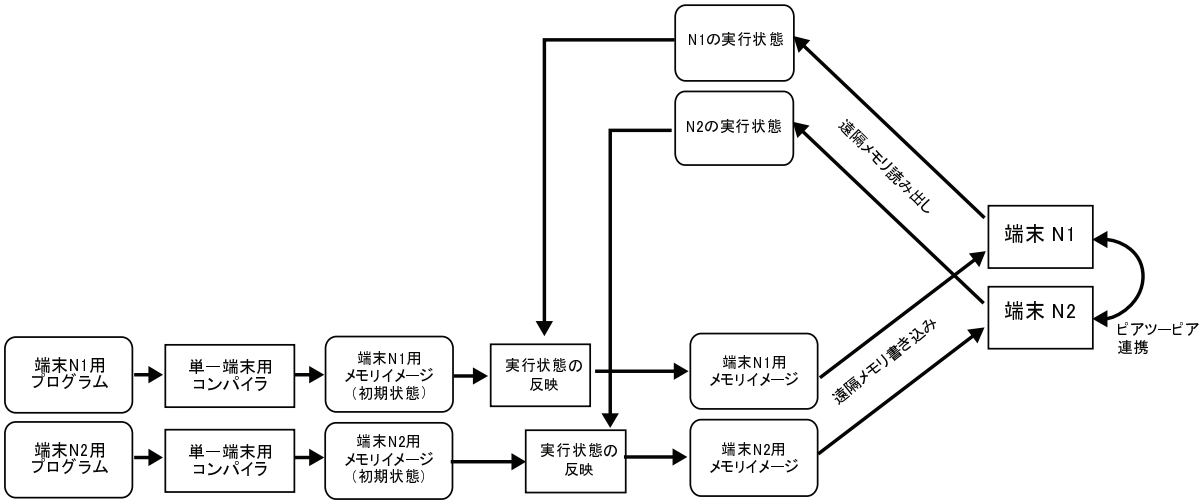

図 11: 遠隔端末のソフトウェアの更新

参考文献

[1] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pp. 93–104, 2000.

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

[3] 岩崎陽平, 河口信夫. 低レベル実行環境向けの P2P 連携機構の動的構成. 第 4 回 SPA サマーワークショップ（SPA-SUMMER 2005）, August 2005.

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

[5] 立堀道昭, 千葉滋, 板野肯三. Addistant：アスペクト指向の分散プログラミング支援ツール. 情報処理学会 研究会論文誌 Programming, Vol. 43, No. SIG03, pp. 17–25, March 2002.

[6] 須永豊. 既存 Java プログラムの機能分散化の支援を行うスクリプト言語 Jacross. 第 3 回 SPA サマーワークショップ（SPA-SUMMER 2004）, 2004.

[7] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of Programming Language Design and Implementation (PLDI)*, 2003.

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

[9] Jim Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, Vol. 42, No. 7, pp. 76–82, 1999.

[10] 岩崎陽平, 河口信夫, 稲垣康善. Touch-and-Connect: ユビキタス環境における接続指示フレームワーク. 情報処理学会論文誌, Vol. 45, No. 12, pp. 2642–2654, March 2004.

[11] Michihiko Minoh and Tak Kamae. Networked Appliances and Their Peer to Peer Architecture AMIDEN. *IEEE Communications Magazine*, Vol. 39, No. 10, pp. 80–84, 2001.

[12] 中田育男. コンパイラの構成と最適化. 朝倉書店, 1999.

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

[14] Lewis Girod, Thanos Stathopoulos, Nithya Ramanathan, Jeremy Elson, Deborah Estrin, Eric Osterweil, and Tom Schoellhammer. A System for Simulation, Emulation, and Deployment of Heterogeneous Sensor Networks. In *ACM Conference on Embedded Networked Sensor Systems (SenSys 2004)*, 2004.

[15] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 3, pp. 319–349, July 1987.