G-03

無線センサネットワークのアプリケーション用 API の設計と実装

Design and Implementation of API for Aplication of Wireless Sensor Network

稲垣彰祐[1] 森 駿介[1] 梅津 高朗[1,2]

Akihiro Inagaki[1] Shunsuke Mori[1] Takaaki Umedu[1,2]

山口 弘純[1,2] 東野 輝夫[1,2]

Hirozumi Yamaguchi[1,2] Teruo Higashino[1,2]

1. まえがき

ワイヤレスセンサネットワーク (WSN) は災害活動や気象観測など様々な用途において利用可能な技術として注目を集めている．WSN では，多くの場合は，有線ネットワークアーキテクチャと比較して，センサ端末の通信および処理性能，電力容量，スケールやデータ処理頻度，対象アプリケーションが大きく異なるため，用途やアーキテクチャに応じたプロトコル設計がなされる [1, 2, 3]．したがってセンサネットワークの設計においては，(i) 自律分散処理を含むアルゴリズムレベルのプロトコル設計，(ii) 隣接ノードの管理やメッセージフォーマットの処理などプログラムレベルの処理を含むアルゴリズム実装，ならびに (iii) シミュレーション実験や実証実験などを通した性能検証，など抽象レベルから実装レベルに至る設計を行う必要がある．

そこで，我々は WSN プロトコルの設計開発から性能評価までを支援する統合開発支援環境 D-sense[4] を提案している．D-sense では既存の様々な WSN プロトコルにおいて頻繁に用いられる処理をモジュール化し (i) でのハイレベルデザインのフェーズに利用できるようにした API を提供することによって (ii) での実装時にかかる時間と労力を軽減する．これにより，設計者はプロトコルの設計に専念することが可能となる．また，実機用コードからシミュレータ用コードへの変換機能，実機から収集した電波受信状況や電力残量をシミュレーションに反映する機能，実行時のデバッグ機能やモニタリングを可能とする実環境を対象とした性能評価支援環境などを提供することで，実機とシミュレーションのシームレスな評価実験環境を実現し，(iii) における課題を解決することを目標としている．

本稿では D-sense の実装支援 API の設計および実装について述べる．D-sense の実装支援 API は，様々なプロトコルで頻繁に用いられる処理やデータ構造をあらかじめ用意しておくことにより，実装の工程を削減し，設計者がプロトコル設計に専念できるようにすることを目標としている．そこで，既存のプロトコルやアプリケーションの処理を参考に，一般的に様々なプロトコルでよく行われる処理を列挙し，API として設計した．特に，複数のノードが関わる一連の処理は実装が煩雑になりやすい．例えば，クエリ処理のような協調処理の場合，クエリを送信する処理，クエリを受信した時に要求に応じてデータを返す処理，さらにクエリやデータを中継するノードの処理のように，多数のノードの動作を全て実装しなければならない．従って，このような一連の処理を API としてあらかじめ用意しておくことにより，簡単なの記述のみで，複数のノード間で行われる具体的なパケット交換を意識しない実装が可能となり，実装の工程の大幅な削減が見込める．

[1]大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

[2]独立行政法人科学技術振興機構, CREST
Japan Science Technology and Agency, CREST

2. 統合開発支援環境 D-Sense の概要

2.1 実装支援

D-sense の提供する主な機能の 1 つが，アルゴリズム設計の支援である．開発者は D-sense の実装支援 API を利用してアルゴリズムレベルでコードを記述し，API トランスレータが API を実装コードに置き換えることで，センサーノード (Mica Mote) 上で実行可能な NesC によるプロトコルの実装を得られる．様々なプロトコルの実装を支援するため，既存の典型的なプロトコルの分析を行い，頻繁に用いられる処理をピックアップして実装支援 API を設計した．この分析は，文献 [5] などを参考に，WSN プロトコルの特徴を考慮して行った．

特に，ルーティングプロトコルは WSN の様々なアプリケーションのベースとなり，必要性が高いため，当初の設計対象とした．例えば，無線範囲内にいるノード ID の取得，指定したノード座標の取得，クラスタ内で最も大きい電力残量を持つノード ID の取得などの処理が API として用意されている．これらの API を利用することで，多くのルーティングプロトコルで実行される動作の実装工程を省略することができると共に，直感的な記述が可能となる．

2.2 シミュレーションと実環境実験のシームレスな連携による実験支援

D-sense はシミュレーションと実環境実験との間の連携を行い，性能評価を支援するための機能の提供を行っている．連携機能として，(1) 2 者間でのコード共有，(2) アニメータによる実環境実験結果の可視化，及び (3) 実環境のログを元にしたシミュレーションのパラメータ設定，が挙げられる．

D-sence を構成する API トランスレータにより生成された NesC のコードはそのまま無線通信を行う端末 Mote 上で実行可能である上，NesC トランスレータによって QualNet シミュレータ用の C++コードを生成することも可能である．可視化機能は，実環境のセンサノード群の動作や状態を QualNet アニメータで表示することができる．

また，パラメータ設定機能は，実環境で得たノードの位置情報や受信メッセージの電波強度のログを元に，実環境での設定や電波状況を継承したシミュレーション環境へのシームレスな移行を実現する．これにより，小規模ネットワークでは実環境で評価し，中規模から大規模ネットワークでは同様

の設定でシミュレーションにより評価するといったことが容易に行える．

2.3 プロトコルのデバッグおよび管理支援

分散環境におけるセンサノード上のプログラムのデバッグは多くの困難を伴う．D-sense はそれを軽減するために，ノードの情報の取得やデバッグの簡易化を可能とするためのデバッグ支援を行う．開発者は，どのノードでどのような条件が成立した場合にセンサネットワーク全体でプログラム実行を停止したいか，また停止後にはどのノードがどのような処理を行うべきかといったデバッグシナリオを記述することでより容易にデバッグを行うことができる．この機能を利用することで，モニタリングや保守などをより容易に運用できる．

3. 実装支援 API および API トランスレータ の設計と実装

本稿では D-sense へと組み込む協調処理 API の設計と実装を行ったため，それについて述べる．特に複数のノードが関わってパケット交換を行う必要のある処理は，ノードの役割ごとに他ノードとの連携を充分考慮して動作を規定し，一連の処理を正しく動作させなければならないため，特定ノードまでのホップ数の取得などの簡単に思えるような処理でも実装には手間がかかり，コードは複雑になりやすい．そのため，こういった処理を API としてあらかじめ用意しておくことは大変効果があると考えられる．

3.1 協調処理 API

ルーティングプロトコルにおいて，例えば GPSR [1]，SPEED [2] など，位置情報を用いる様々な手法が考案されているが，各ノードがあらかじめ自身の位置情報を保持しておくことは，場合によっては困難である．そのため，ノード自身が他ノードとの通信により得ることのできる情報から自身の位置を推定する位置推定プロトコルは必要性が高く，極めて重要である．そこで以下では D-sence の API のうち，位置推定を対象として設計した API 群を例として説明を進める．

位置推定などの協調処理アプリケーションにおいて共通して行われている一般的な処理を考え，メッセージ管理を行う API，データ送信を行う API，ネットワーク管理を行う API の 3 つのレベルに分けて API を設計した．さらに，これらの API を用いて，位置推定などのアプリケーション中のハイレベルな処理を API として設計した．設計した主な協調処理 API を表 1 に示す．

表 1 の get_oneHopDistance() は 1 ホップあたりの平均距離を計算する API であり，協調処理 API である get_hop() を用いて設計を行った．計算方法については，既存の位置推定手法 DV-Hop[6] で行われる方法を用いている．座標が既知である基地局 (以下，ランドマーク) が他のランドマークまでのホップ数，および距離を取得し，一定数のランドマークからの情報を取得後，各ランドマークまでのホップ数の総和と距離の総和から 1 ホップあたりの距離を算出する．各ランドマークは自身を含めた全ランドマークのノード ID，および座標を知っているものとすると，上記の計算を行うためには他のランドマークまでのホップ数を取得する必要があるが，この処理を内部的に協調処理 APIget_hop() を用いて行っている．この API は計算結果を第一引数として指定されている distance に格納し，その後，ユーザによって定義される event の内容を実行する．

表 1: 協調処理 API の一部

メッセージ管理 API
受信したメッセージを保存してから一定時間経過後に破棄 -> store_msg(msg,time) 指定されたメッセージを取得 -> get_msg(msg,seq)
データ送信 API
指定されたホップ数分だけブロードキャスト -> khop_broadcast(k,msg,event) 指定領域へデータを送信 -> sendTo_area(area,data,event)
ネットワーク管理 API
隣接テーブルを取得 -> get_neighborTable(table,event) 指定ノードまでのホップ数を取得 -> get_hop(ID,hop,event) 1 ホップあたりの平均距離を計算 -> get_oneHopDistance(distance,event)

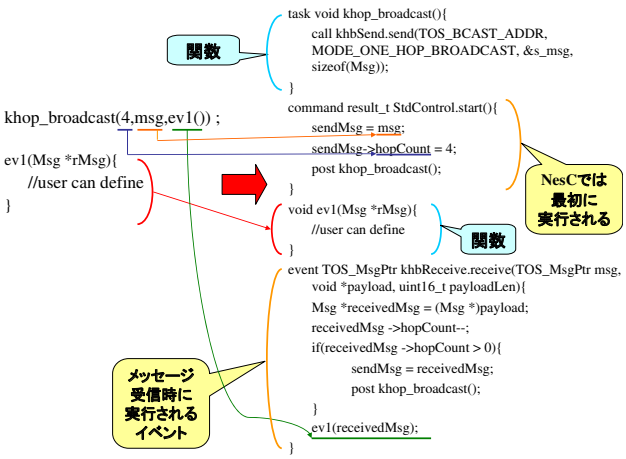

図 1: トランスレータの変換概略図

以上のようにそれぞれのレベルにおいて一般的な処理を API として用意し，さらにそれを用いてハイレベルな API も用意しておくことで，協調処理を行う様々なプロトコルやアプリケーションの実装工程を削減できる．

3.2 API トランスレータ

D-sense の API トランスレータは前述のような API で記述された部分を検出し，あらかじめ用意されているコードを調整して置き換えることで，全て NesC で記述されたプログラムを生成する．

トランスレータの API の変換概略図を図 1 に示す．図 1 では指定されたホップ数分だけブロードキャストを行う API である khop_broadcast() の場合の変換概略図を示している．

変換前の khop_broadcast() は引数が 3 つあり，左から順にブロードキャストするホップ数”4”，送信するメッセージ”msg”，メッセージを受信した時に実行される関数”ev1()”，となっている．ここには，メッセージ中のデータを特定の変数に格納するなどの処理内容を，別途ユーザが定義した関数の関数名として指定する．受信メッセージを使った処理を記述することを想定しているため，受信メッセージを引数とする関数を指定する必要がある．ただし，指定されたホップ数分のブロー

ドキャストを行うための判定処理や転送処理は API にて行っているため ev1() には記述する必要がなく，メッセージを受け取った後に各ノードで行う処理のみを記述すればよい．

変換後については，NesC の記述方法に従って StdControl.start() から khop_broadcast() が実行され，指定されたメッセージがブロードキャストされる．NesC ではメッセージを受信した場合には記述例末尾に記述されているような event 関数が実行される．そこで，この関数内に変換前の khop_broadcast() に渡したメッセージが受信された場合の処理を追加する．最初に残りホップ数の値を減らし，0 でなければ転送を行っている．その後に，ユーザによって指定されていた ev1() を実行する．

また，NesC においてはメッセージ受信時にどのイベントが実行されるかはインターフェースと呼ばれる識別子によって決まる．記述例においては変換後の上部の”khbSend”，および末尾の event の”khbReceive”がインターフェースであり，これらが外部のインターフェース定義ファイルで接続されているため，”khbSend.send()”で送信したメッセージの受信時には”khbReceive”が実行される．トランスレータはこの外部ファイルの記述も行う．

4. API を用いた実装例

既存の位置推定手法である DV-Hop[6] を例として API の利用イメージを説明する．DV-Hop は，各ノードが，ランドマークとの距離をホップ数から推定し，自身の座標を求めるプロトコルである．ランドマークの処理を D-sense の実装支援 API を用いて実装した例を以下に示す．

```
typedef struct sendMsg{
    uint16_t srcID;
    double distance;
    uint16_t hopCount;
}sendMsg;

double oneHop_distance;

command result_t StdControl.start(){
    get_oneHopDistance(oneHop_distance,ev1());
}

ev1(){
    sendMsg *sMsg;
    sMsg->srcID = TOS_LOCAL_ADDRESS;
    sMsg->distance = oneHop_distance;
    khop_broadcast(6,sMsg,ev2());
}

ev2(sendMsg *rMsg){
}
```

DV-Hop におけるランドマークの処理は，他の複数のランドマークの位置，およびそのランドマークまでのホップ数から 1 ホップあたりの平均距離を見積もり，それを他の通常のノードへ送信する，というものである．1 ホップあたりの平均距離の見積もりを get_oneHopDistance()，他のノードへ送信を khop_broadcast() という API を用いて実装している．

NesC においては，最初に StdControl.start() が実行される．その中で用いられている get_oneHopDistance() は 3.1 節で述べた協調処理 API で，get_hop() を用いて 1 ホップあたりの平均距離を計算して返す．

ev1() では送信メッセージの内容の作成と，送信を行っている．khop_broadcast() は指定ホップ数分だけメッセージの

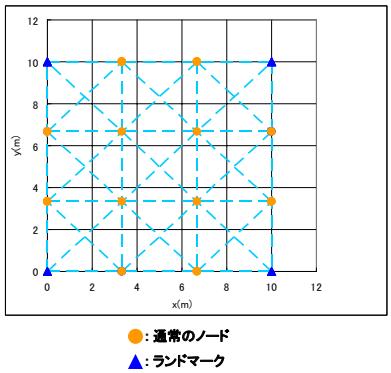

図 2: ノード配置およびノード間の想定接続関係

ブロードキャストを行う．この API を使う場合には，ユーザはメッセージのフィールドには uint16_t hopCount; という項目を含めなければならない．hopCount は API により自動的に更新されるため，ユーザがホップ数を格納しておく必要はない．ここでは 6 ホップ以内のノードへと位置推定に必要な情報を送信している．ev2() は，khop_broadcast() が送信したメッセージを受信した時に実行される．各ランドマークは自身の位置がわかっているため位置の推定を行う必要がなく，判定，転送処理は API が行っているため，受信メッセージに対して何も行う必要がない．そのため，ev2() 内では何も実行しない．一方，それ以外のノードでは，十分な数のメッセージを受け取った場合に，メッセージに記録された distance とホップ数から，自身の推定位置を計算してユーザに提示する等の処理を記述すれば良い．

5. 動作検証

API を利用した実装の正当性を示すため，API による実装を行った DV-Hop を実環境上のセンサノードを用いて動作させた．ここではノードの数に限りがあったため，16 ノードを図 2 に示すように配置して実験を行った．16 ノード中ランドマークは 4 ノードとした．推定を行うノードが 3 つのランドマークで構成される三角形の外部にあると結果の座標が不正確になりやすいため，ランドマークは正方領域の四隅に配置した．残りのノードは図 2 のように均一に配置した．

上記の環境において，各ノードはランドマークから定期的に送信される情報を基に位置推定を行った．そして，図 2 で示すような，想定通りのノード間の接続関係であった場合の各ノードの DV-Hop による推定位置をあらかじめ計算しておき，実験結果の推定位置をそれと比較することで評価を行った．また，実験は D-sense のログ回収機能などを用いることでスムーズに行うことができた．

図 3 は DV-Hop による実験で得られた推定位置と理論上の推定位置と実際のノード配置を示している．理論上の推定位置と実験結果の推定位置は線で繋いでいる．また，表 2 は実験結果の推定位置，理論上の推定位置，実験結果の推定位置と理論上の推定位置の誤差，を示している．表 3 は実験時に各ノードが計算に使用したランドマークとそこまでのホップ数を示している．

あらかじめ計算しておいた理論上の推定位置は理想的な条件で計算を行ったため，図 3 に示すように，4 つのランドマークで構成される正方形の中心に近い 4 つのノードは実際の配置と一致するなど，いずれのノードも実際のノード位置のす

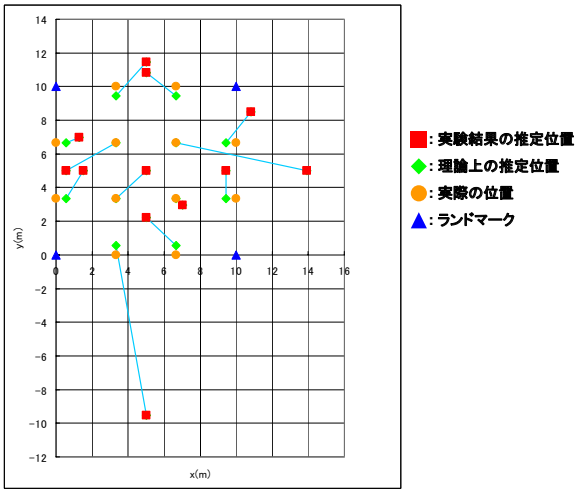

図 3: 実験結果

表 2: 実験結果

ノード ID	実験結果の推定位置	理論上の推定位置	実験結果と理論値の誤差
1	(5, -9.52)	(3.33, 0.56)	10.22
2	(5, 2.22)	(6.67, 0.56)	2.36
3	(1.5, 5)	(0.56, 3.33)	1.92
4	(5, 5)	(3.33, 3.33)	2.36
5	(7.04, 2.96)	(6.67, 3.33)	0.53
6	(9.44, 5)	(9.44, 3.33)	1.67
7	(1.3, 6.98)	(0.56, 6.67)	0.8
8	(0.56, 5)	(3.33, 6.67)	3.24
9	(13.92, 5)	(6.67, 6.67)	7.44
10	(10.83, 8.5)	(9.44, 6.67)	2.29
11	(5, 11.48)	(3.33, 9.44)	2.63
12	(5, 10.83)	(6.67, 9.44)	2.17
平均値			3.13

表 3: 実験時に各ノードが計算に用いたランドマーク情報 (LM はランドマーク)

ノード ID	LM1	hop	LM2	hop	LM3	hop
1	L1	1	L3	5	L4	5
2	L2	2	L3	3	L4	3
3	L1	1	L2	2	L4	2
4	L1	2	L3	2	L2	2
5	L2	2	L4	3	L1	3
6	L2	1	L4	1	L3	3
7	L3	1	L1	3	L4	4
8	L1	1	L4	3	L2	3
9	L4	1	L1	4	L3	4
10	L4	1	L2	2	L1	3
11	L4	1	L4	1	L1	3
12	L4	2	L1	3	L2	3

ぐ近くの座標となった．しかし，図 3，表 2 で示すように実験結果の推定位置と理論上の推定位置の誤差はやや大きい値となった．原因として考えられるのは，実環境であるため，無線通信距離が不安定であり，かつ各端末ごとに差があるということである．表 3 に示すようにランドマークから 5 ホップで受信したデータを用いていたり，通信できないはずのノードと通信していたりするなど想定外の通信がなされている例が多くあることが分かった．

通信が不安定であるためにホップ数が想定外の値となっている場合も多いが，それらのいわゆるはずれ値を除けば，各ノードでの計算そのものは正しく行われており，一連の処理は問題なく行われていると言える．

6. まとめと今後の課題

本稿ではマルチホップ通信を行う WSN のプロトコル及びアプリケーション開発を支援するための API を設計し，その実装を行った．API の設計対象として，複数ノード間協調処理を取り上げ，さらに API を用いた記述を NesC のみを用いた記述に変換するトランスレータの設計についても述べた．また，位置推定アプリケーションの 1 つを API を用いて実装を行い，実環境実験により実装の評価を行うことで，実装支援 API の利便性・正当性を示した．

今後はさらに設計対象を拡張し，実装支援 API を用いてより多くの WSN プロトコルの設計開発を支援することを目指していく．

参考文献

[1] B. Karp et al.: GPSR: Greedy Perimeter Stateless Routing for Wireless Networks, *Proc. of MobiCom*, pp. 149–160 (2000).

[2] T. He et al.: SPEED: A Stateless Protocol for Real-Time Communication in Sensor Networks, *Proc. of ICDCS*, pp. 46–55 (2003).

[3] D. Braginsky et al.: Rumor routing algorthim for sensor networks, *Proc. of WSNA*, pp. 22–31 (2002).

[4] 森 駿介, 梅津 高朗, 廣森 聡仁, 山口 弘純, 東野輝夫：ワイヤレスセンサネットワークの設計開発支援環境 D-sense, 情報処理学会論文誌, Vol. 50, No. 10 (採録決定).

[5] J. N. Al-Karaki et al.: Routing techniques in wireless sensor networks: a survey, *IEEE Trans. on Wireless Communications*, Vol. 11, No. 6, pp. 6–28 (2004).

[6] D. Niculescu and B. Nath: DV Based Positioning in Ad Hoc Networks, *Journal of Telecommunication Systems*, Vol. 22, pp. 267–280 (2003).