平成２１年度　特別研究報告書

MANET 環境における
端末バッテリーに依存した
フラッディング手法

龍谷大学　理工学部　情報メディア学科

学籍番号　　T050495　　麻田 浩二

指導教員　三好 力 教授

概要

近年、無線通信技術の発展により Mobile Ad-hoc Network(MANET)技術が注目されつつある。MANET の特徴として、ネットワークインフラを必要としない、マルチホップ通信でネットワークを構築できるという利点があるが、長時間利活用するにはデータを送信する際の消費電力を考慮しなければならないという問題点がある。本研究では様々なルーティングプロトコルに使用されているフラッディングの送信半径に焦点を置き、端末バッテリーに依存したフラッディングを提案し、シミュレーション実験により本研究の有効性の検証を行った。

シミュレーション実験は既存のフラッディングと提案手法を同じ環境で比較し、フラッディングの成功率、ノードの生存率、平均最短経路ホップ数、平均消費電力量をそれぞれ評価した。結果として、ノード密度に比例してフラッディングの成功率、ノードの生存率、平均消費電力量は既存方法よりも提案手法の有効性が確認できたが、端末バッテリーに依存して送信半径を変更したため、既存方法よりも提案手法の平均最短経路ホップ数は増加した。

目次

第1章 はじめに

1.1 本研究の背景

携帯電話や携帯ゲーム機、ネットブックなどの普及により誰もが常に携帯端末を持ち歩くようになった。電気通信事業者協会の発表によると、日本国内において 2009 年 9 月時点の事業者別携帯電話・PHS 契約数は 1 億 1 千台数を越えている(表 1)。日本の携帯電話・PHS の保有率は 95.6%[1]と高い数値である。

表 1　事業者別携帯電話・PHS 契約数（2009 年 9 月現在）

携帯電話事業者	契約総数
NTT ドコモ	55,186,500
au(KDDI)	31,232,700
Softbank	21,316,900
EMOBILE	1,897,700
合計	109,633,800
PHS 事業者	契約総数
ウィルコム	4.3.34,900

ワンセグ放送等年々新しいサービスが提供され、無線通信の規格 IEEE802.11n や WiMAX、携帯電話の通信規格である LTE などの登場により無線環境でも高速な通信ができるようになってきている。また近年、無線通信技術は携帯端末に限らず、車や自動販売機に搭載されるなど利活用が増えてきている。

このようにモビリティを考えられた製品や技術が今後も続々と開発されていくことが予想される。

現在、一般的な無線ネットワークは無線基地局などのインフラストラクチャが必要とするものが大半を占めている。これに対して無線基地局等のインフラを必要としないネットワークをアドホックネットワークという。アドホックネットワークの中でも多数の移動端末でネットワークを構成しているネットワーク、Mobile Ad-hoc Network（MANET）にも期待が高まっている。MANET は無線通信が可能な端末が集まることによりネットワークを形成することができる。基地局等既存のインフラがない場所でも通信が可能なことから、災害時などのインフラが機能しない場所や状況においての活躍が期待できる。しかし、MANET にもまだまだ問題点や改善点があり、さまざまな研究が行われている。

1.2 本研究の目的

1.1 節のように、MANET には利活用できる範囲が増えてきている。しかし、長時間の使用については通常よりも消費電力を考慮しなければならない。MANET では移動端末を使用してアドホックネットワークを構築することを想定している。実際、移動端末のバッテリー量

は移動端末の保持者によって異なり、バッテリー容量が十分の端末もあれば限りなく少ない端末もある。MANET ではデータを送る際、送信する端末と受信する端末の他に中継する端末も電力を消費する。もちろん、データを送る以前の経路発見のための経路制御パケットでも同じことが言えるが、送信元から宛先までデータを転送するために経路を確立しても、バッテリー量の少ない端末が経路の一部となってしまった場合には、バッテリー切れのためデータが届かないことや、端末保持者が緊急で使用したいときにバッテリーが少なく何もできないということになりかねない。

本論文では、移動端末が MANET を形成し、データ転送を行ってもバッテリー切れにならないような経路探索を行うフラッディング手法を提案する。

1.3　本論文の流れ

本論文は以下の章により構成される。

第 1 章では、本研究の概要について述べる。

第 2 章では、Mobile Ad-hoc Network（MANET）の概要とルーティング、経路探索のためのフラッディングとそのルーティングプロトコルについて解説する。

第 3 章では、提案手法である端末バッテリーに依存したフラッディングについて解説する。

第 4 章では、実験環境と実験内容、及びに実験結果を示し、それに対する考察を行う。

第 5 章では、本論文に関するまとめと今後の課題について述べる。

第2章 Mobile Ad-hoc Network

本章では Mobile Ad-hoc Network（MANET）の概要と関連技術について述べる。

2.1 MANET とは

MANET とは基地局を必要とせず、移動端末を利用してマルチホップ通信を行うネットワークのことを指し示す。従来の無線通信では、インターネットに接続した無線ルータやアクセスポイントに各端末が接続することで、インターネットに接続したり、互いに通信したり、インターネット上にあるサービスを利用したりすることができる。しかし、MANET では基地局などを利用してインターネットに接続することなく、各端末と直接通信し、即席のネットワークを構築する。アドホックネットワークのアドホックとは『その場限りの』という意味である。また、マルチホップ通信とは送信元端末から宛先端末まで直接通信できない場合に、複数の中継端末を通信経路としてデータを伝送することである（図 1）。

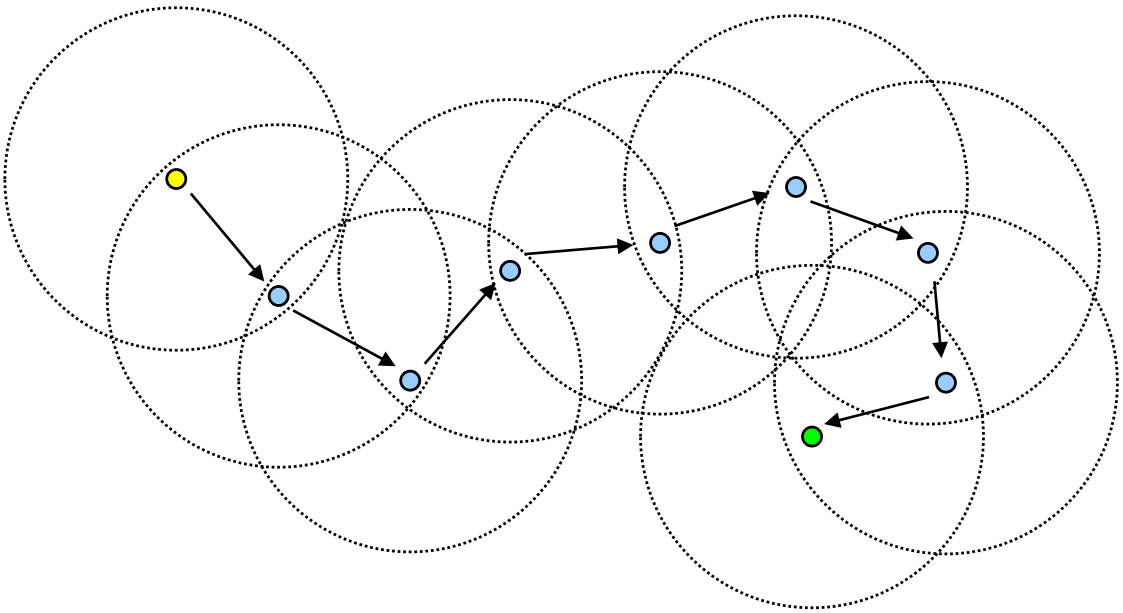


図 1　アドホックネットワークにおける通信の様子

他にも MANET には端末が移動することを前提としており、経路を確立しても、経路になっていた端末が移動してしまうことがあり、もう一度同じ経路が利用できるとは限らない。そのため、ネットワークトポロジの変化に対応できるようになっている（図 2）

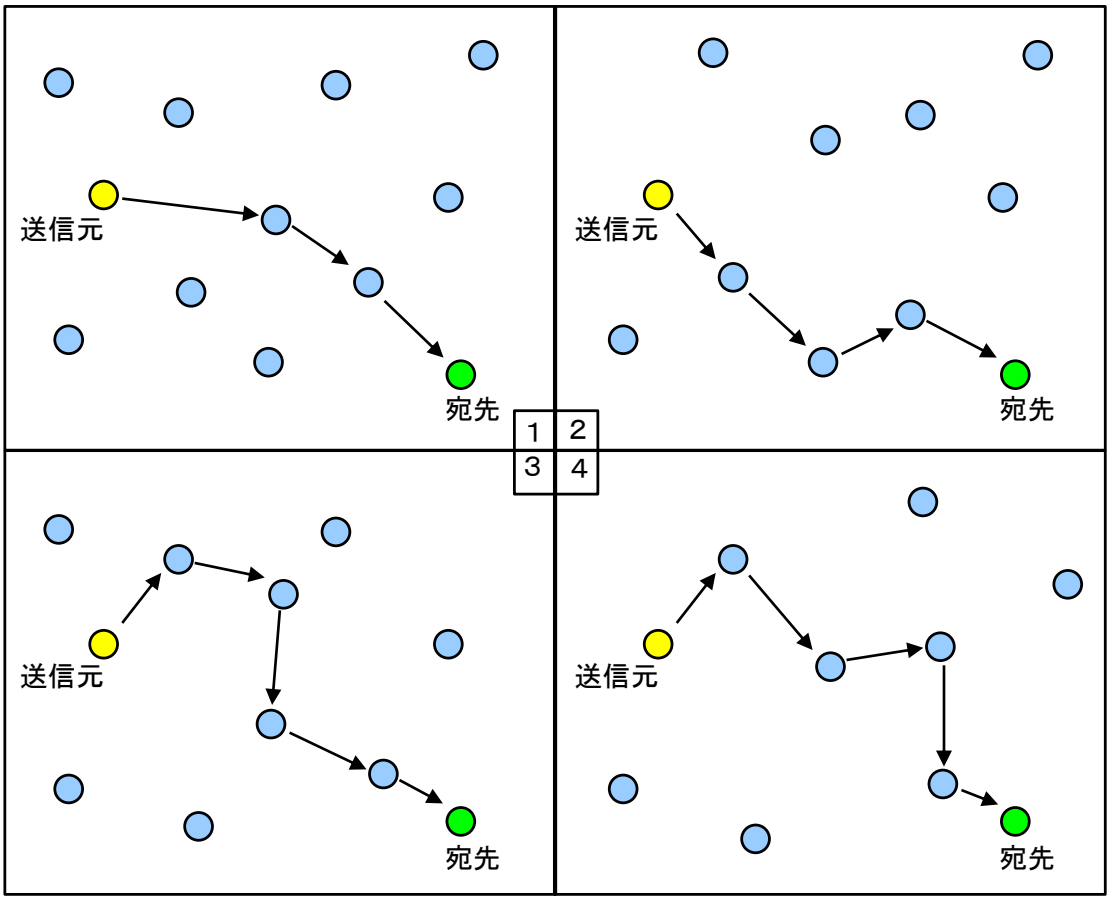


図 2　MANET におけるトポロジの変化に対応した配信経路の例

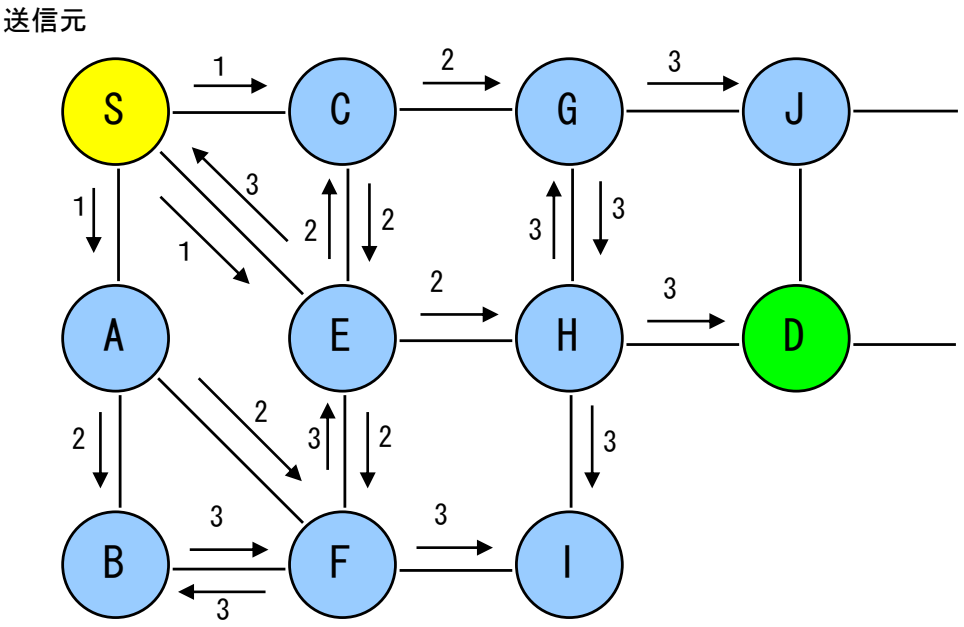
2.2　ルーティングとは

ルーティングあるいはルーティング制御とは、経路制御、すなわちデータパケットを送信元ノードから宛先ノードまで転送し送り届けるまでの中継制御のことを意味する。経路制御の機能は、ISO（International Organization for Standardization）が規定したコンピュータネットワークのプロトコル体系で 7 層から構成される OSI（Open Systems Interconnection）基本参照モデルでは、第 3 層であるネットワーク層に位置付けられる。

MANET では、データ送信要求があった場合に随時通信経路を作成する方式や、あらかじめ通信経路を作成し、データ送信要求があればすぐに伝送が行うことができる方式などがある。これらの詳しい方式については 2.5 節のルーティングプロトコルで解説する。

2.3　フラッディングとは

MANET ではデータを転送する際、端末がどの端末と繋がっているか各端末が知っている必要がある。各端末が通信経路を作成するために行う技術がフラッディング[2]である。これはデータを送るためではなく、経路発見のための経路制御パケットを通信するもので、ブロードキャストで次から次へと転送して経路を発見していくものである。しかし、ノード数の増加に比例してフラッディングによる経路制御パケットが増えてしまうため、経路探索にかかるオーバーヘッドが非常に多くなる。通常は TTL（Time To Live）を設定し、転送数に制限を設けることによって経路制御パケットが必要以上に転送されることを防ぐ。しかし、送信元ノードは宛先ノードまでのホップ数が事前にわからないため、TTL を小さく設定してしまうと宛先ノードまで経路制御パケットが到達しない可能性がある。フラッディングによる通信の様子を図 4 に示す。



*数字は送信元からフラッディングする時に、
ノード間の通信時間を同じと仮定した場合のタイミングを示す

図 4　フラッディングの様子

フラッディングによる各ノードの手順を以下に記す。送信ノード(S)は宛先ノード(D)にデータを送るため通信経路を探索する。最初に S は通信できるノードを探すために経路制御パケットをブロードキャストする。経路制御パケットを受け取ったノードはまず自分が宛先でないか確認する(A,C,E)。自分が宛先でなかった場合、経路制御パケットの TTL を確認する。TTL の値が 0 の場合は経路制御パケットを破棄する。TTL の値が 0 でなかった場合、受け取った経路制御パケットをブロードキャストする(B,C,E,F,G,H)。一度送信した経路制御パケットをもう一度受信した場合は経路制御パケットを破棄する。最初に D に届いた経路を通信経路にするため、通信経路の確定を知らせる経路制御パケットを通ってきた経路に対して返送する（D→H→E→S)。

2.4 主なルーティングプロトコル

MANET において、経路探索や保持といった経路制御やフラッディングに関するメッセージ量の抑制などを考慮する必要がある。このような問題を解決するために IETF の MANET WG で標準化が進められてきた。図 3 のように、その構造的特徴からリアクティブ型、プロアクティブ型、ハイブリッド型などに分類できる。

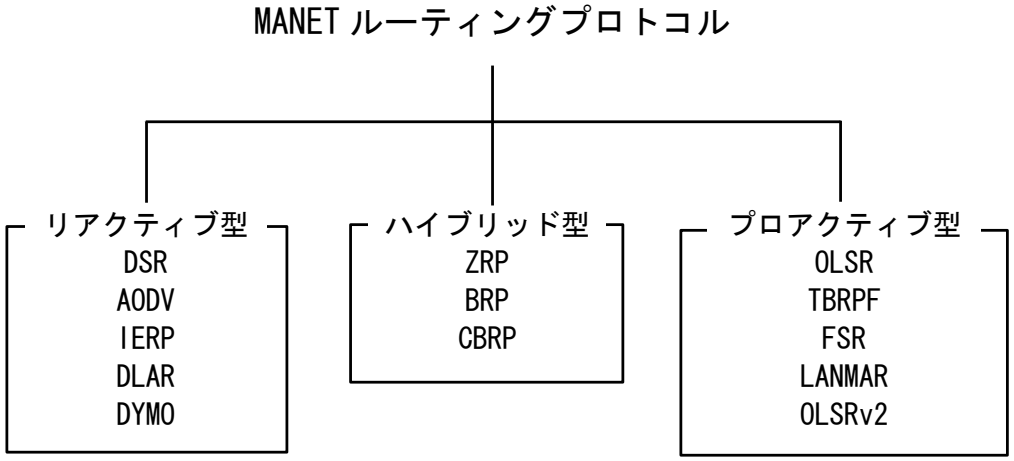


図 3　MANET における代表的なルーティングプロトコル

2.4.1　　リアクティブ型プロトコル

　　リアクティブ型のプロトコルでは、通信要求があった場合に経路探索を行うため、通信要求がない場合には経路制御パケットはまったく送信されない。しかし、複数のノードから短時間のうちに通信要求があると大量のデータパケットと経路制御パケットでネットワークに負荷をかけてしまう。
そのため、経路探索によって確立された通信経路を再利用することによってネットワークの負荷を軽減している。このようなリアクティブ型のプロトコルは各ノードの移動が頻繁に起こるネットワークの場合に有効である。

・ DSR (Dynamic Source Routing)

DSR[3]は MANET の中では最も古くから研究が進められてきたプロトコルの一つである。主な特徴としては、送信元ノードから宛先ノードまでの経路を全てパケットヘッダに付加し、データパケットが宛先ノードまで到達するまでその情報が使用され転送される。ルーティングテーブルを使用せずにこのように転送を行うことをソースルーティングと呼ぶ。宛先ノードまでの通信経路がわからない場合、送信元ノードは宛先ノードまでのネットワークを形成するために経路要求のための経路制御パケット（RREQ：Route Request）を隣接するノードに向けてブロードキャスト的にフラッディングを行う。RREQ パケットを受信したノードは自分が宛先でなければ中継ノードとなるので、RREQ パケットに自分の ID を追加し、隣接するノードに向けて同じくフラッディングを行う。RREQ パケットにはどのノード

を経由してきたかが順に記録されている。RREQ パケットを受信したノードが宛先ノードであった場合は、経路を確立したことを送信元ノードに知らせるために経路応答パケット（RREP：Route Reply）を RREQ パケットに記録されている経路情報を利用してユニキャストで送信元ノードに返答する。宛先ノードは別の経路を経由した複数の RREQ パケットを受信することがあり、その場合は複数の経路の RREP パケットを返すこともできる。

RREP パケットを受信した送信元ノードは RREP パケットの経路情報を用いて宛先ノードまでの経路を知ることができ、複数の RREP パケットの経路情報をキャッシュすることもできる。RREP パケットの到着後、送信元ノードは宛先ノードへデータパケットの転送を開始する。各ノードは RREQ パケットを送信する前に自分自身のキャッシュに宛先ノードへの経路情報があるか確認する。その中に宛先ノードまでの経路を発見するとキャッシュを使うことによって、経路探索の回数を減らすことができ、トラフィックの低減、経路発見の高速化、電池の消費節約を図っている。

・AODV（Ad-hoc On Demand Vector）

AODV[4]の主な特徴は各ノードが次にどのノードに送ればよいかというルーティングテーブルを保持している点である。すなわち、各ノードは次の中継ノードもしくは宛先ノードのどちらかだけを知っており、送信元ノードから宛先ノードまでの経路情報の全容は知らされていない。ルーティングテーブルを作成するために、通信要求を出している送信元ノードから DSR と同じく RREQ パケットを隣接するノードに向けてフラッディングする。中継ノードも同じくフラッディングを行い、宛先までの経路を確立していく。宛先ノードまで RREQ パケットが到達すれば送信元ノードへ RREP パケットを返答するが、複数の経路があった場合、送信元ノードからのホップ数が最も少なかった経路に対してユニキャストで RREP パケットを返答し、中継ノードはそのやりとりの中でルーティングテーブルを作成していく。データパケットの転送はルーティングテーブルによって行い、パケットの中に中継ノードの情報は挿入されない。AODV プロトコルでは RREQ パケットや RREP パケットなどの経路制御パケットのヘッダに送信元ノードと宛先ノードのシーケンス番号を挿入する。シーケンス番号を挿入することによって全経路がループせず、最新の経路情報を保持することを保証する。

また、各ルーティングテーブルには、RREP パケット受信時に前もって迂回路として設定しておくプリカーサノードを保持しており、リンクに障害が起こったときに利用する。プリカーサノードとは前ホップのノード群を保存しておくことで、ノードの移動や電源が切られてしまうなど経路が使用できなくなった場合に、プリカーサノードとして保存してあるノードに対してルーティングテーブルを書き換えるための経路制御パケット（RERR：Route Error）を転送する。RERR パケットを受信したノードは該当経路情報を無効とし、自身のプリカーサノードに対して RERR パケットを送信する。

2.4.2　プロアクティブ型プロトコル

プロアクティブ型のプロトコルは事前に通信経路を確立するため、各ノードにおいて定期的に経路情報を交換する。その際に、いかにして経路情報の量を抑えるかが重要な点である。このようなプロアクティブ型のプロトコルは各ノードの移動が少ないネットワークに対して有効である。また、事前に通信経路を確立しているため、通信要求があればすぐにデータパケットを送信できるという利点もある。

・OLSR (Optimized Link State Routing)

OLSR[5]の主な特徴は、マルチポイントリレー（MPR）というノードの集合を使用してフラッディングを効率的に行っている。従来のフラッディングは全てのノードが必ずパケットを一度だけ中継するのに対して、MPR 集合を利用したフラッディングでは、必要最小限のノードだけがパケットを中継する。MPR 集合とはフラッディングに必要最小限の数の再送信ノードの集まりである。パケットを受信したノードはすべて、直ちにそのパケットを次の隣接ノード群へ再送信するが、そのノードを再送信ノードと呼ぶ。図 4 に MPR 集合による再送信ノードの削減の様子を示す。MPR の選択にあたっては、まず各ノードが隣接ノードの Hello メッセージの交換により、2 ホップ先までの経路情報を収集する。図 4（1）では楕円で囲んだ五つのノードが再送信ノードであるが、この通信では無駄が生じており、図 4（2）のように再送信ノードは三つで済む。この三つのノードが MPR 集合である。

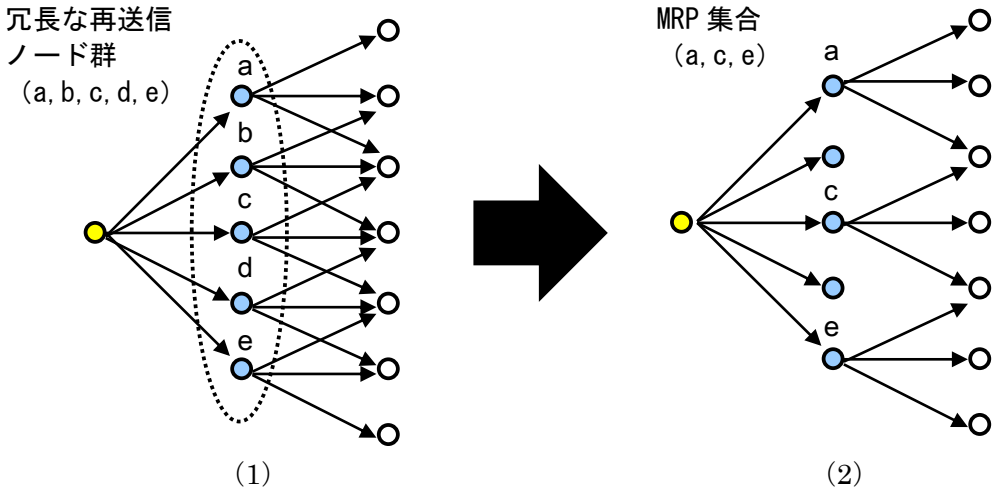


図 4　MPR 集合による再送信ノードの削減

実際にこの MPR を利用したフラッディングは、経路表の作成に利用されている。MPR として選択されたノードは、自分がどのノードの MPR であるかというリストをフラッディングする。あるノードが選択している MPR 集合がわかるということは、そのノードへの経路の 1 つ手前のノードがわかるということである。さらにその 1 つ手前のノードの MPR 集合もわかるため、最終的に各ノードはネットワーク全体のトポロジ構成を認識することが可能となる。実際にデータパケットを転送する際には、この経路表に従って送信される。

・TBRPF (Topology Broadcast Based on Reverse-Path Forwarding)

TBRPF[6]の主な特徴はネットワークトポロジの変化による各ノードのルーティングテーブルの後進に要する処理を最小限に抑えるため、積極的に安定したリンクの選択やトポロジの差分情報を利用している。経路などの情報をすべて定期的に更新するのではなく、何

が追加され何が削除されたかなどの必要最小限の情報を利用する。この仕組みによって、やりとりすべきメッセージの量を小さくすることができ、比較的短い期間に周期的に経路情報を送受信することができる。このため、ノードの移動などによる急激なトポロジの変化にもプロアクティブ型の中では比較的強いプロトコルになっている。

2.4.3 ハイブリッド型プロトコル

リアクティブ型の経路制御プロトコルとプロアクティブ型の経路制御プロトコルの両方の特性を持つものが、ハイブリッド型の経路制御プロトコルとなる。そのため、2 つのプロトコルの利点と欠点を有することになる。ハイブリッド型の代表的なプロトコルである ZRP[7]では、近隣ノードとの通信経路は、プロアクティブ型の動作により定期的に更新するが、数ホップ先のノードとの経路については、通信を行いたい場合に限りリアクティブ型の動作により、通信経路を確保する。

このように MANET には様々なプロトコルが提案されており、これらを比べるとそれぞれに利点と欠点を持っている。

2.5 問題点

2.4 節で述べたルーティングプロトコルを含め、2.3 節で示したフラッディングにはいくつか問題がある。

一つはノード数が多く、TTL の値が大きければブロードキャストされる経路制御パケットは増加し、通信帯域を圧迫してしまうことである。このため単純なフラッディングにおけるパケット数を削減するため、様々な方式が提案されている[2]。

方式1. 中継するノードを確率的に決定する
方式2. ノード関の距離を測定して遠くのノードが中継する
方式3. 各ノードの位置情報などを基にして、重複領域が少なくなるノードを中継する
方式4. クラスタ（ノードの集合）を構成するプロトコルにより、クラスタ単位に通信する為のクラスタヘッドを用いる
方式5. 第 2 層の送信キューで送信待ちをしているパケットに対しても、重複パケットなのチェックを行うことで、効率を向上させる

他の問題点として、セキュリティに関することがある。MANET ではデータを送信する際、ノードを中継して宛先まで届く。この時、データはキャッシュとして経路となったノードに保存されている。宛先までデータが届いても、キャッシュを消去する命令は出ていない為、配信されてもキャッシュとして保持されている。ノード保持者は意識せずに MANET の経路になっている。

また、MANET ではデータ通信時に消費する電力を考慮しなければならない。何故ならノード数が増えることによって、フラッディングによる経路制御パケットが増加する。それと MANET では通信経路が確立されたとき、送信元ノードと宛先ノードを直接つなぐことは少ないと考えられる。2.1 節で述べたように、MANET はマルチホップ通信で送信元ノードと宛先ノードをつなぐため、中継ノードを使用することを想定してある。このとき、ノード保持者は自分の持っているノードが中継ノードとして利用されていることを知らない。そのた

め、知らない間に中継ノードとしてデータ通信を行い、電力を使用し、ノード保持者が使用したいときに自分の持っているノードのバッテリー量が少なくなっていることが考えられる。これは通信経路になりうる全てのノードに対して言えることである。これに加え、2.3 節で述べたフラッディングは送信できるノードを探索するため、経路制御パケットをブロードキャストするが、送信半径は一律である。一律であるということはノードのバッテリー量が減少しても同じフラッディングを行う。そのため、通信回数が増加すれば中継したノードがバッテリー切れを起こしてしまい、MANET 環境における中継ノードの減少はデータを届けにくくなってしまう恐れがある。

第3章 端末バッテリーに依存したフラッディング手法

本章では 2.5 節で述べた問題点のうち消費電力に焦点を置き、フラッディング段階での電力消費を低減する手法を提案する。

3.1 概要

ここで提案する手法は既存のフラッディングを拡張したものを指す。既存のフラッディングとは 2.3 節で示したものである。既存のフラッディングはブロードキャストを行う際に送信できる距離は常に出力最大で行っており、送信半径は一定である。端末バッテリーに依存したフラッディングとはフラッディングを行う際に、各ノードからブロードキャストで隣接するノードに向けて送信される経路制御パケットの送信半径を、端末のバッテリー量によって変更する手法である。この手法は既存のフラッディングと比べ、通信要求の増加とともに減少していく中継ノードの生存率を軽減できると考えられる。しかし、送信半径を変えることによってホップ数が増加することが予想される。

また、この手法はフラッディングを利用するルーティングプロトコル全てに適用できると考えられる。

3.2 提案手法 1

提案手法 1 とは既存のフラッディングを元に、端末バッテリーに依存して送信半径を二段階に変更する。変更点は端末のバッテリー量が 50%以上であれば既存のフラッディングと同じ送信半径でフラッディングを行う。端末のバッテリー量が 50%未満になれば既存のフラッディングの送信半径も 50%でフラッディングを行う。アルゴリズムについては 3.4.2 節で示す。

3.3 提案手法 2

提案手法 2 とは既存方法を四段階に拡張した形となっており、変更点は端末のバッテリー量が 75%以上であれば既存のフラッディングと同じ送信半径でフラッディングを行う。端末のバッテリー量が 50%以上 75%未満になれば、既存のフラッディングの送信半径も 75%でフラッディングを行う。端末のバッテリー量が 25%以上 50%未満になれば、既存のフラッディングの送信半径も 50%でフラッディングを行う。端末のバッテリー量が 25%未満になれば、既存のフラッディングの送信半径も 25%でフラッディングを行う。アルゴリズムについては 3.4.3 節で示す。

3.4 アルゴリズム詳細

以下にシミュレーション実験を行った際に用いたアルゴリズムを説明する。

3.4.1 既存のフラッディング

既存のフラッディングについて以下にアルゴリズムを示す。

＜開始＞
1 初期化
1.1 ノード数、TTL 値、バッテリー量、送信半径を設定
2 全てのノードをランダムに配置
3 送信元ノードと宛先ノードを決定
4 フラッディング
4.1 送信元ノードは送信半径内にいるノードに対してパケットを送信
4.2 TTL が 0 になった場合、フラッディングを終了
4.3 一度送信したパケットを受信した場合、パケットを破棄
4.4 パケットを受信したノードは、自分が宛先でなかった場合、受信したパケットをそのパケットを送信したノード以外に送信半径内にいるノードに対して送信
4.5 TTL を 1 減らす
4.6 4.2 に戻る
5 最短経路になっているノードのバッテリー量から送信半径の 2 乗分をそれぞれ消費させる
6 各種データの取得
＜終了＞

3.4.2 提案手法 1

提案手法 1 について以下にアルゴリズムを示す。

＜開始＞
1 初期化
1.1 ノード数、TTL 値、バッテリー量、送信半径を設定
2 全てのノードをランダムに配置
3 送信元ノードと宛先ノードを決定
4 フラッディング
4.1 送信元ノードは送信半径内にいるノードに対してパケットを送信
この時、送信元ノードのバッテリー量が 50%以上の場合は既存のフラッディングと同じ通信半径で行う。バッテリー量が 50%以下だった場合、送信半径も 50%とする。
4.2
4.3 TTL が 0 になった場合、フラッディングを終了
4.4 一度送信したパケットを受信した場合、パケットを破棄
4.5 パケットを受信したノードは、自分が宛先でなかった場合、受信したパケットをそのパケットを送信したノード以外に送信半径内にいるノードに対して送信

この時もノードのバッテリー量が 50%以上の場合は既存のフラッディングと同じ通信半径で行う。バッテリー量が 50%以下だった場合、送信半径も 50%とする。
4.6
4.7 TTL を 1 減らす
4.8 4.2 に戻る
5 最短経路になっているノードのバッテリー量から送信半径の 2 乗分をそれぞれ消費させる
6 各種データの取得
＜終了＞

3.4.1 提案手法 2

提案手法 2 について以下にアルゴリズムを示す。

＜開始＞
1 初期化
1.1 ノード数、TTL 値、バッテリー量、送信半径を設定
2 全てのノードをランダムに配置
3 送信元ノードと宛先ノードを決定
4 フラッディング
4.1 送信元ノードは送信半径内にいるノードに対してパケットを送信
この時、送信元ノードのバッテリー量が 75%以上の場合は既存のフラッディングと同じ通信半径で行う。バッテリー量が 50%以上 75%以下だった場合、送信半径も 75%とする。バッテリー量が 25%以上 50%以下だった場合、送信半径も 50%とする。バッテリー量が 25%以下だった場合、送信半径も 25%とする。
4.2
4.3 TTL が 0 になった場合、フラッディングを終了
4.4 一度送信したパケットを受信した場合、パケットを破棄
4.5 パケットを受信したノードは、自分が宛先でなかった場合、受信したパケットをそのパケットを送信したノード以外に送信半径内にいるノードに対して送信
この時、ノードのバッテリー量が 75%以上の場合は既存のフラッディングと同じ通信半径で行う。バッテリー量が 50%以上 75%以下だった場合、送信半径も 75%とする。バッテリー量が 25%以上 50%以下だった場合、送信半径も 50%とする。バッテリー量が 25%以下だった場合、送信半径も 25%とする。
4.6 TTL を 1 減らす
4.7 4.2 に戻る
5 最短経路になっているノードのバッテリー量から送信半径の 2 乗分をそれぞれ消費させる
6 各種データの取得
＜終了＞

第4章 実験及び評価

本章では、本手法の有効性を実証するためシミュレーション実験を行う。比較、評価を行う指示として、成功率、ノード生存率、平均最短経路ホップ数、平均消費電力量の 4 項目を使用する。

4.1 実験概要

本論文では、既存のフラッディングと提案手法をそれぞれ比較、評価を行うために C 言語での実装を行った。評価内容としては成功率、ノード生存率、平均消費電力、平均最短経路ホップ数をそれぞれ集計し、フラッディング回数毎の平均により比較を行う。シミュレーションを行うためのパラメータ設定は 4.2 節で解説する。ここで述べているノード生存率とは送信電力消費量以上のバッテリー量があるノードを指し、既存のフラッディングでは 400 以上、提案手法 1 では 100 以上、提案手法 2 では 25 以上のバッテリー量の端末数とする。

4.2 実験環境

実験環境を図 5 に示す。実験環境は障害物のない空間とし、100×100 の空間に 100 から 900 数のノードをそれぞれランダムに配置させている。各ノードは送信半径 20 以内に存在するノードと直接通信することができ、マルチホップ通信を用いて MANET を形成する。送信元ノードと宛先ノードはランダムに決定し、経路探索にはフラッディングを行う。シミュレータで 1000 回のフラッディングを行い、回数毎の成功率、ノード生存率、平均最短経路ホップ数、平均消費電力量を比較、評価する。詳しいシミュレーションパラメータは表 4 にまとめる。

表 4　シミュレーションパラメータ

モデルサイズ	100×100
ノード数	100, 200, 300, 400, 500, 600 , 700, 800, 900
通信半径	20
初期バッテリー量	1000
フラッディング回数	1000
試行回数	10000
送信電力消費量	通信半径の二乗

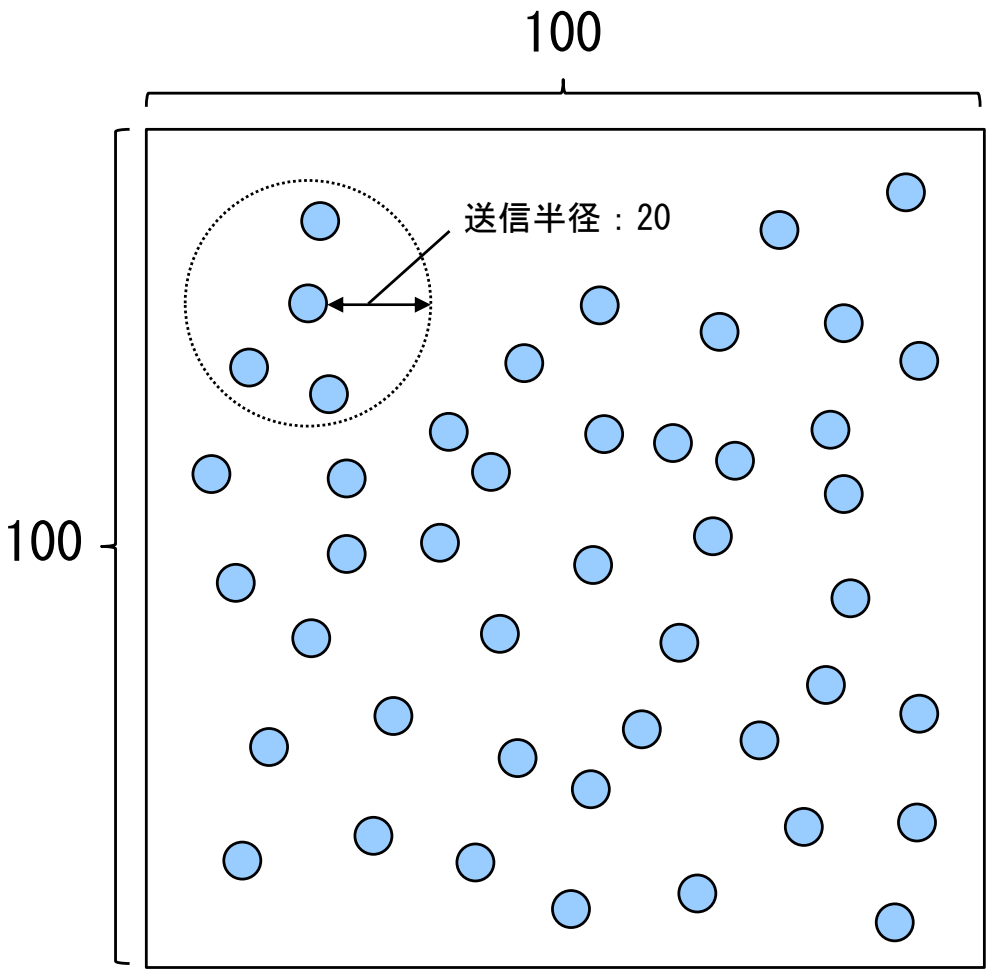


図 5　実験環境

4.3　実験結果

実験結果を図 6 から図 41 に示す。図 6 から図 14 までが成功率、図 15 から図 23 までがノード生存率、図 24 から図 32 までが平均最短経路ホップ数、図 33 から図 41 までが平均電力消費量を表している。ノード数によってそれぞれグラフ化しており、縦軸については各比較項目を、横軸についてはフラッディング回数を示している。

4.3.1　成功率の考察

図 6～14 の縦軸は成功率、横軸はフラッディング回数を示す。図 6 では既存方法と提案手法での差はそれほどないが、図 7 から既存方法と提案手法に変化が見られた。提案手法 2 に関しては、ノード数が増えるとフラッディング回数が増えても高い成功率を維持し続けることができた。提案手法 1 は既存方法と似た曲線を描いており、成功率 50%までは既存方法よりも少し高い値を示している。しかし、図 9 あたりから見られる既存方法よりも成功率が一時的に下回るのは、送信半径が小さくなったときに宛先まで経路を確立しにくいためであると考えられる。

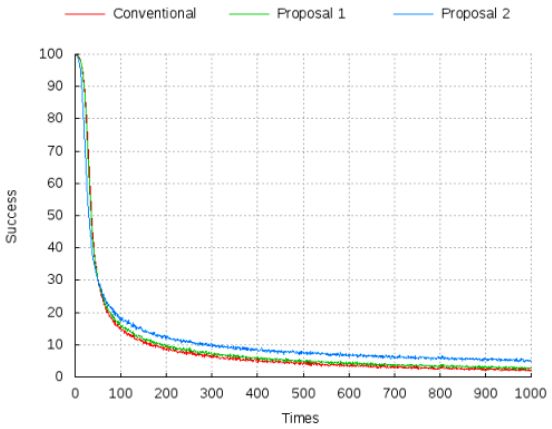


図 6　成功率(ノード数:100)

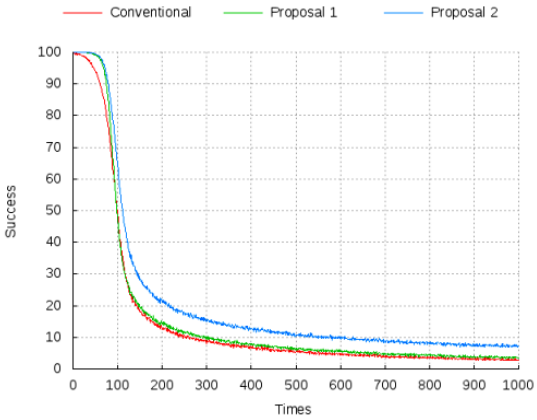


図 7　成功率(ノード数:200)

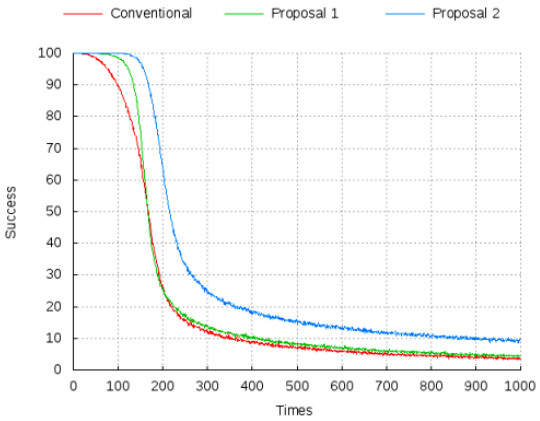


図 8　成功率(ノード数:300)

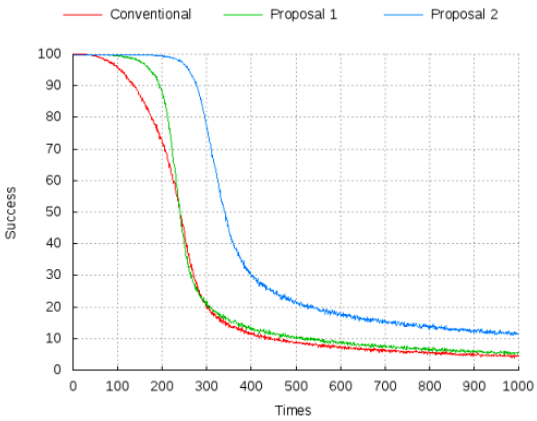


図 9　成功率(ノード数:400)

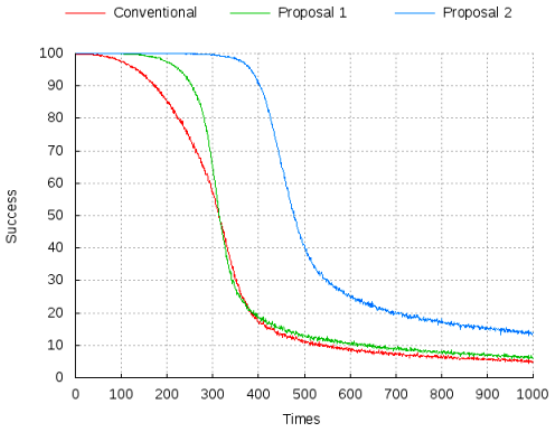


図 10　成功率(ノード数:500)

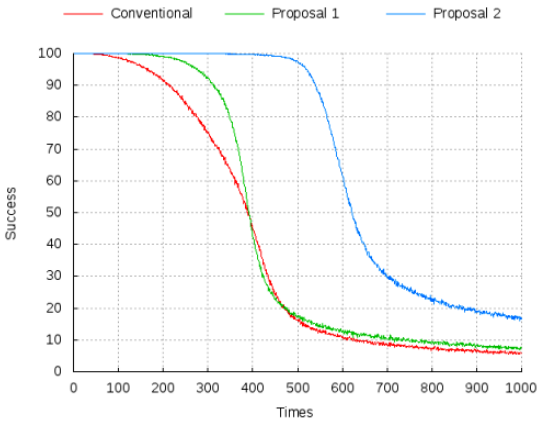


図 11　成功率(ノード数:600)

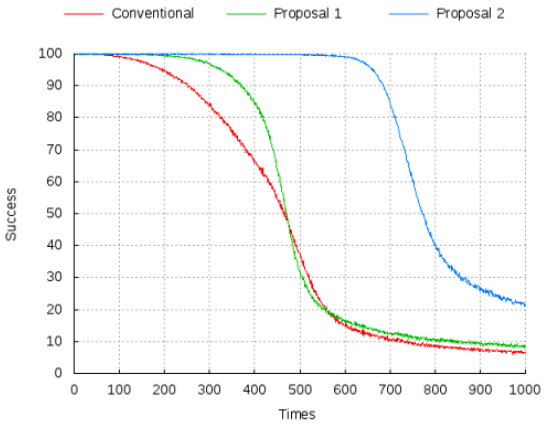


図 12　成功率(ノード数:700)

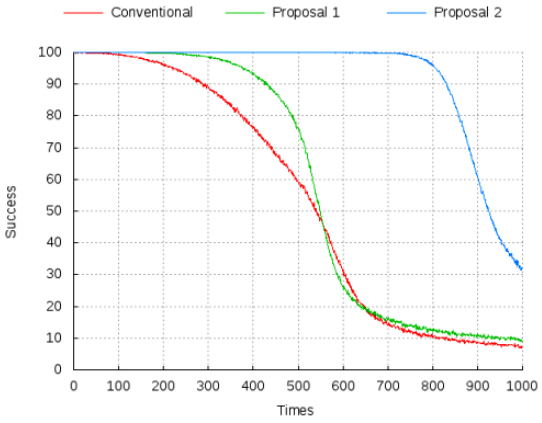


図 13　成功率(ノード数:800)

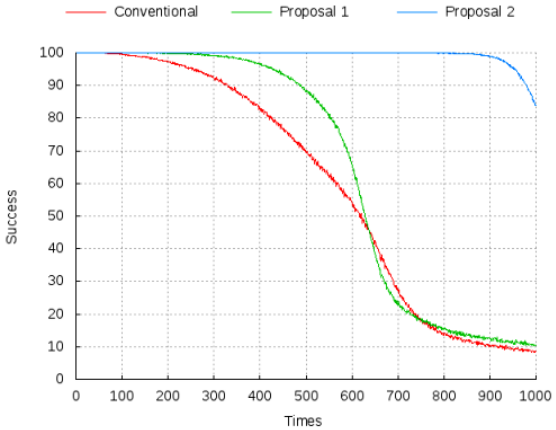


図 14　成功率(ノード数:900)

4.3.2　ノード生存率の考察

図 15～23 の縦軸はノード生存率、横軸はフラッディング回数を示す。既存方法と提案手法とで、フラッディングが行えるバッテリー量を持っているかの基準が異なるため、図 15 から図 23 までのような差の開いたグラフとなっている。既存方法はノード数が 100 を切ったあたりから緩やかな曲線になるが、提案手法 1 ではある地点から緩やかな曲線になっているのは、バッテリー量が 50%以下になったため送信半径も 50%になり、ノード生存率の減少が抑えられていると考えられる。提案手法 2 ではほぼ全てのノードがフラッディングを行える状態であり、通信頻度が増加しても有効であることがわかった。

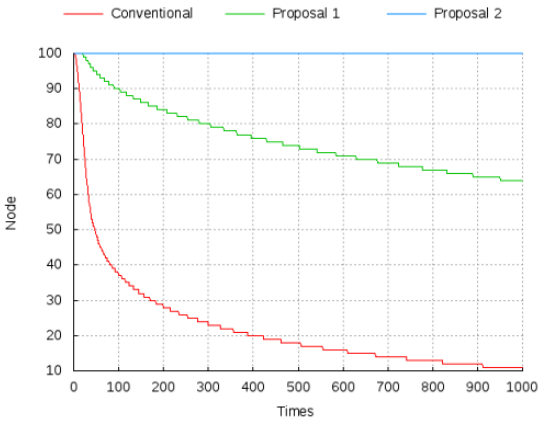


図 15　ノード生存率(ノード数:100)

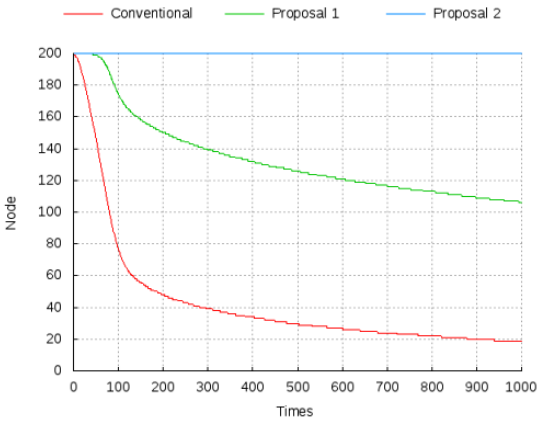


図 16　ノード生存率(ノード数:200)

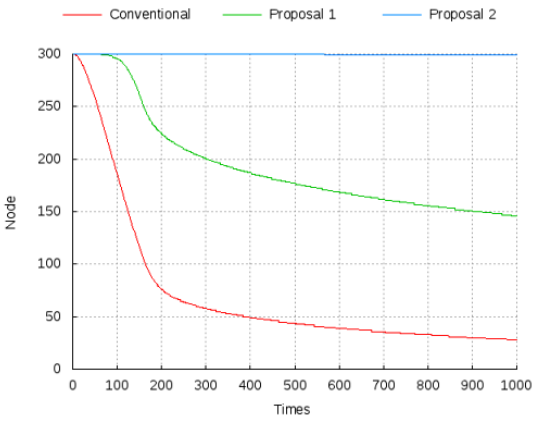


図 17　ノード生存率(ノード数:300)

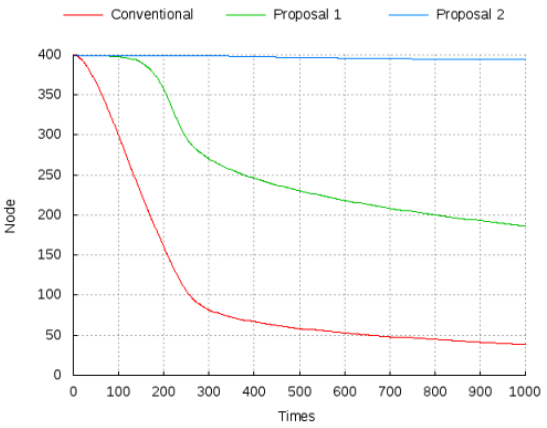


図 18　ノード生存率(ノード数:400)

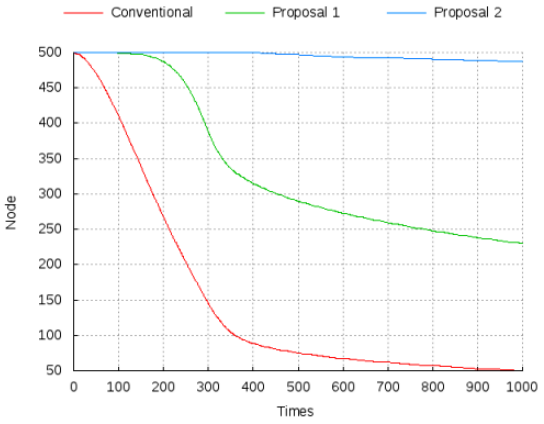


図 19　ノード生存率(ノード数:500)

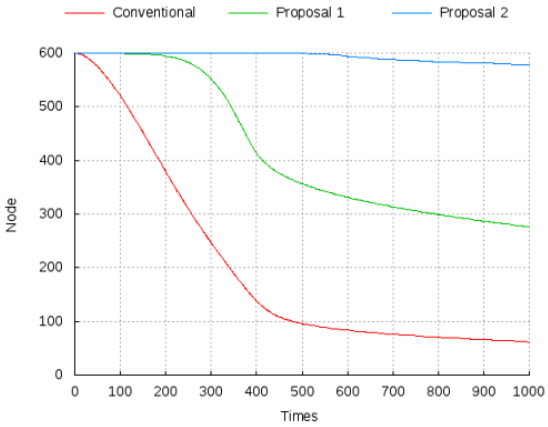


図 20　ノード生存率(ノード数:600)

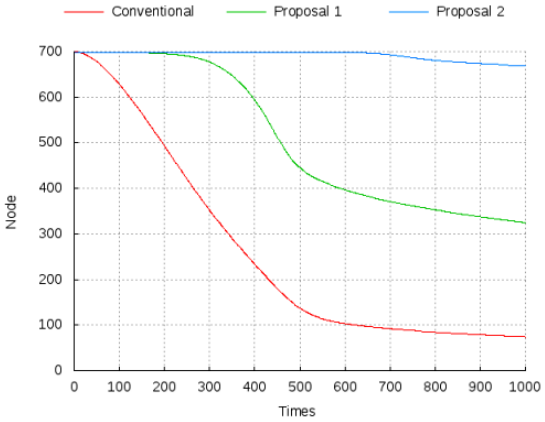


図 21　ノード生存率(ノード数:700)

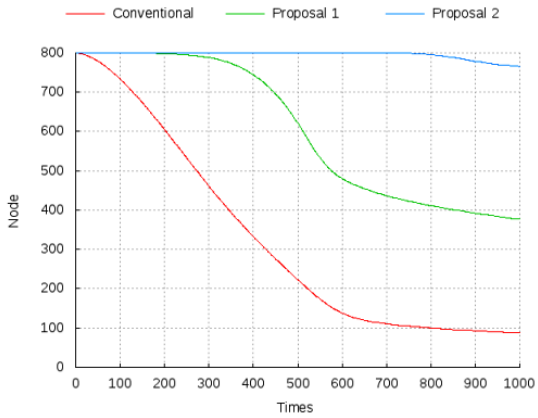


図 22　ノード生存率(ノード数:800)

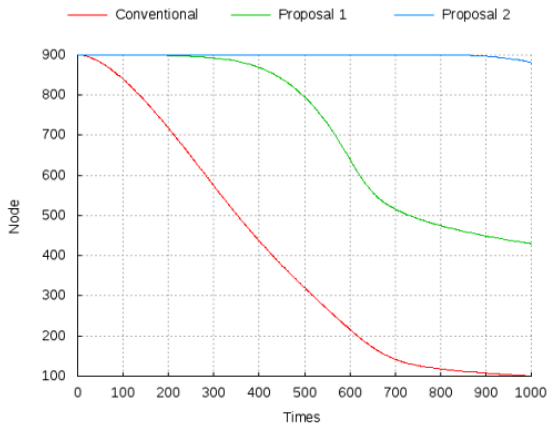


図 23　ノード生存率(ノード数:900)

4.3.3　平均最短経路ホップ数の考察

図 24～32 の縦軸は平均最短経路ホップ数、横軸はフラッディング回数を示す。予想通り、提案手法のホップ数は既存方法よりも高い値となった。ノード数の増加とともに、平均最短経路ホップ数も増えている。特に提案手法 2 は成功率が 60%あたりのとき一番ホップ数が多くなっていた。また、提案手法 2 で図 28 から見られる最大値より少し前に一度盛り上がっているのは、バッテリー量が減り、送信半径が変化することで起こったものと考えられる。

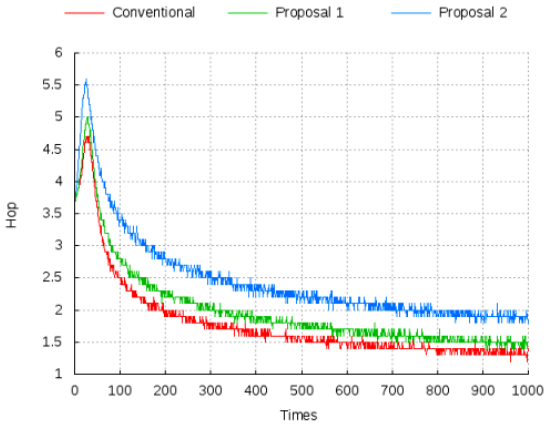


図 24　平均最短経路ホップ数(ノード数:100)

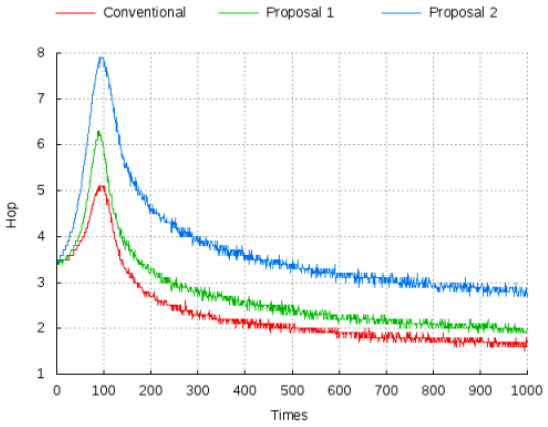


図 25　平均最短経路ホップ数(ノード数:200)

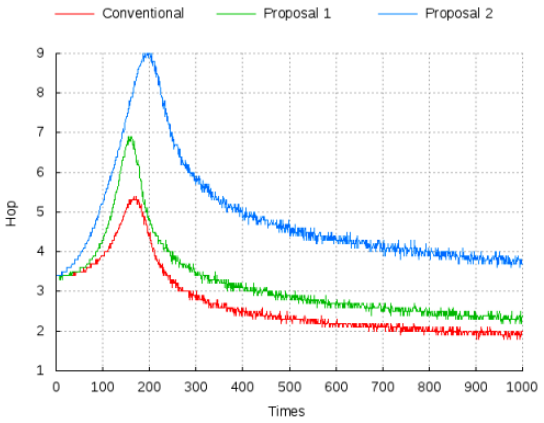


図 26　平均最短経路ホップ数(ノード数:300)

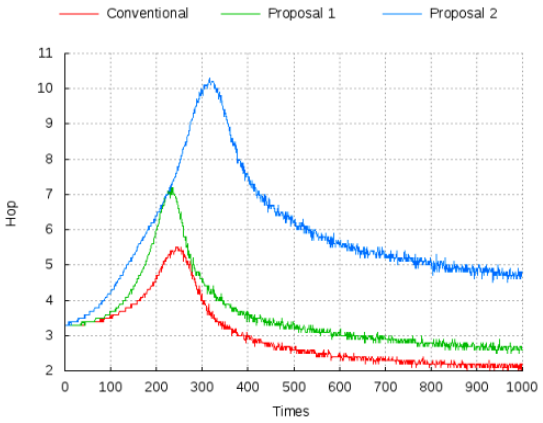


図 27　平均最短経路ホップ数(ノード数:400)

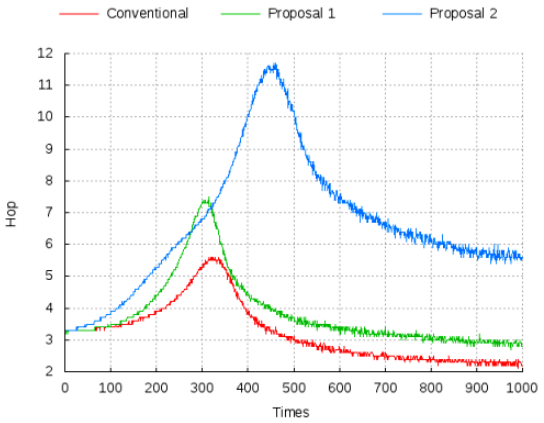


図 28 平均最短経路ホップ数(ノード数:500)

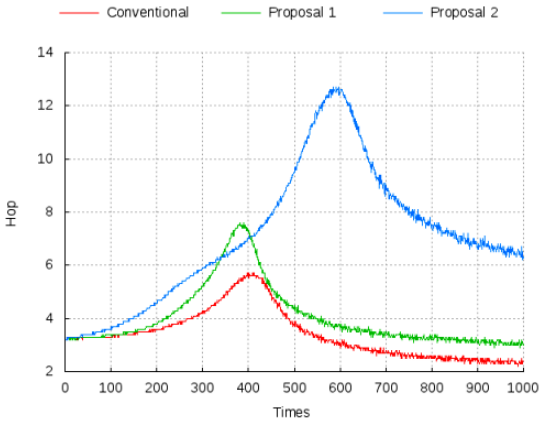


図 29 平均最短経路ホップ数(ノード数:600)

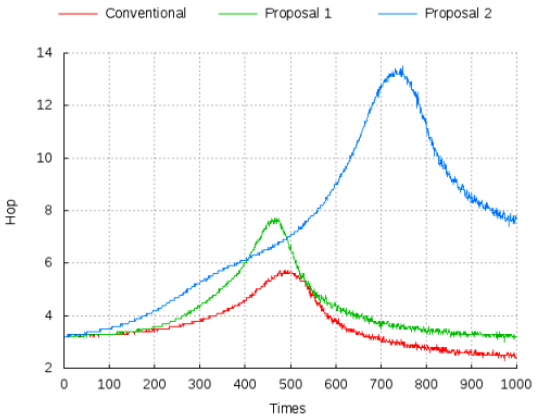


図 30 平均最短経路ホップ数(ノード数:700)

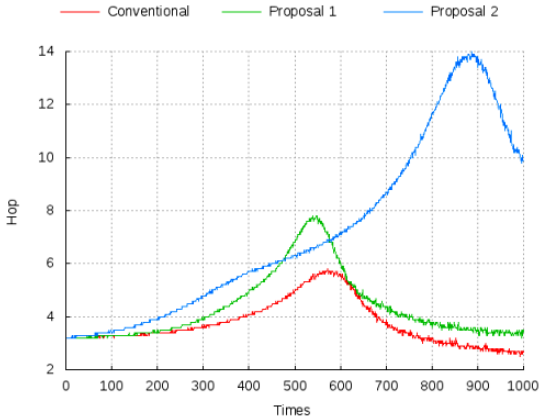


図 31 平均最短経路ホップ数(ノード数:800)

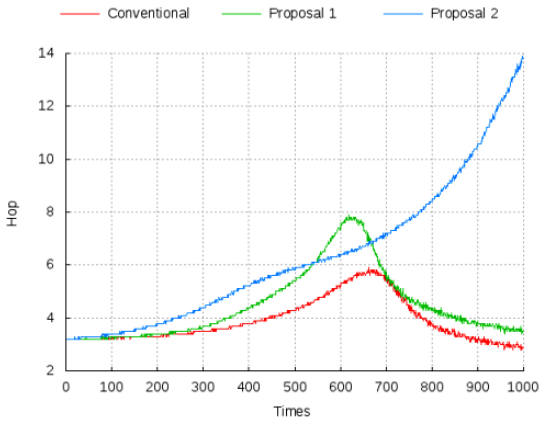


図 32 平均最短経路ホップ数(ノード数:900)

4.3.4　平均消費電力量の考察

図 33～41 の縦軸は平均消費電力量、横軸はフラッディング回数を示す。図 33 から図 41 を見てわかる通り、既存方法よりも提案手法のほうが消費電力量は少なくなっている。既存方法については平均最短経路ホップ数に比例して平均消費電力も増加しており、平均最短経路ホップ数が一番多いとき消費電力量も最大値を示している。しかし、提案手法 1 について最初は消費電力量が少しずつ増加しているが、平均最短経路ホップ数が一番多いときを境に消費電力量は大きく低下している。提案手法 2 では平均最短経路ホップ数が一番多いとき消費電力量も少し増加するが、最大値ではなくフラッディング回数に比例して減少し続けている。

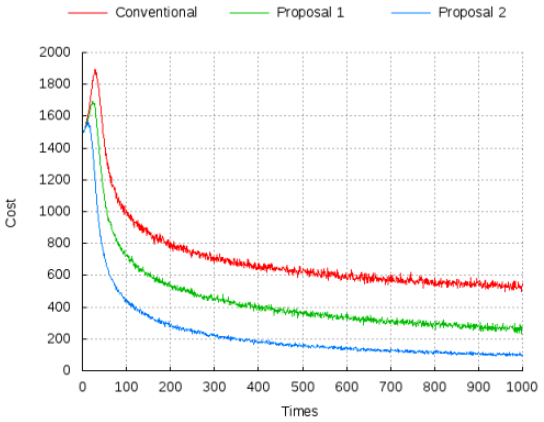


図 33　平均消費電力量(ノード数:100)

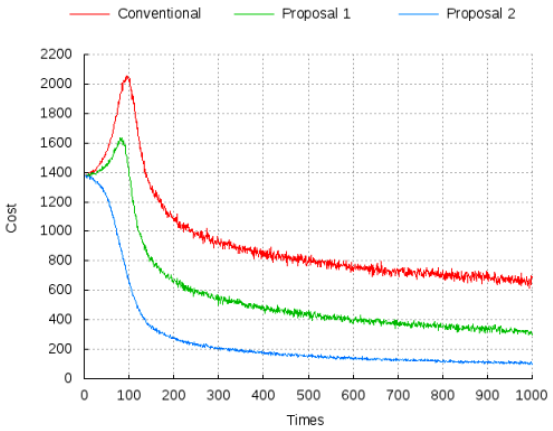


図 34　平均消費電力量(ノード数:200)

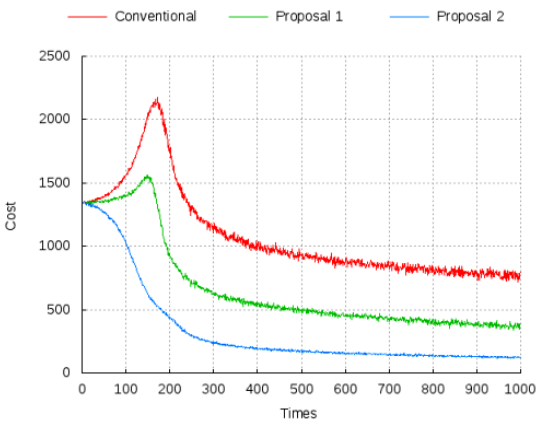


図 35　平均消費電力量(ノード数:300)

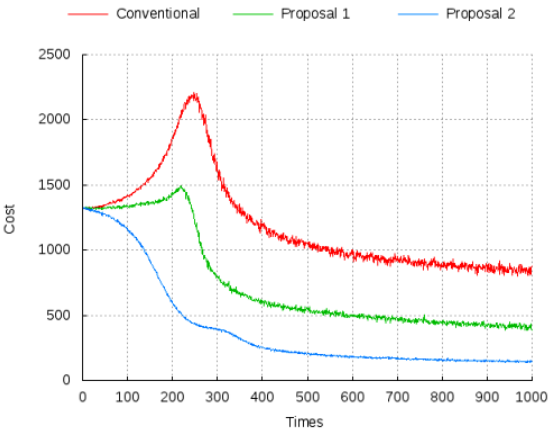


図 36　平均消費電力量(ノード数:400)

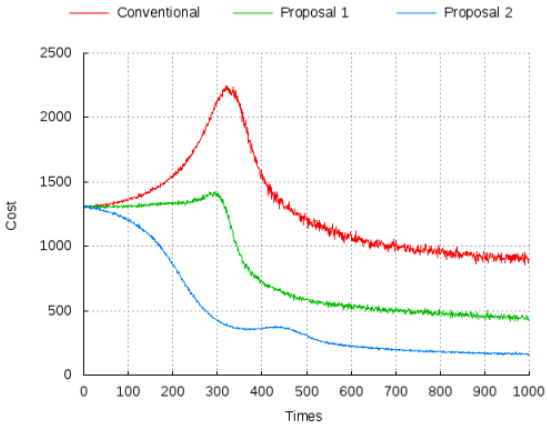


図 37 平均消費電力量(ノード数:500)

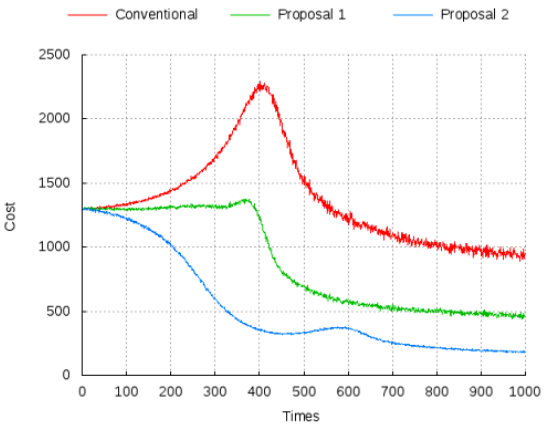


図 38 平均消費電力量(ノード数:600)

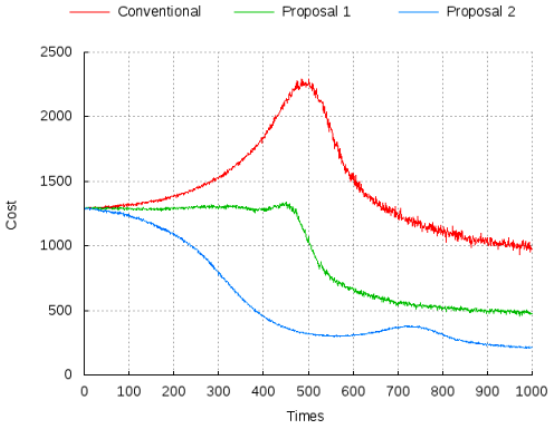


図 39 平均消費電力量(ノード数:700)

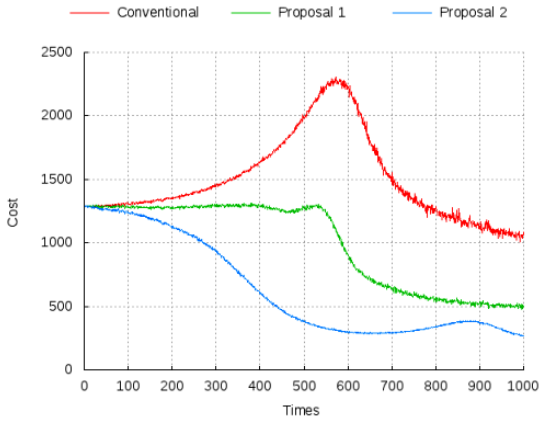


図 40 平均消費電力量(ノード数:800)

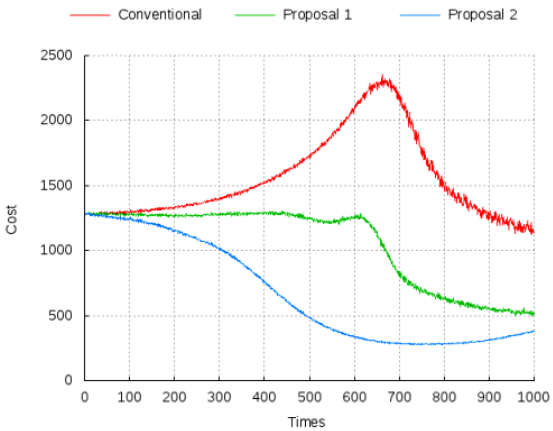


図 41 平均消費電力量(ノード数:900)

第5章　おわりに

5.1　本研究のまとめ

本論文では、既存のフラッディングを改良し、通信頻度が増加してもより効率よくフラッディングを行うことができる端末バッテリーに依存したフラッディング手法を提案した。

実験結果から、提案手法 1 について平均最短経路ホップ数は増加してしまうが、ノードの生存率、平均消費電力量については既存方法よりも有効性は確認できた。しかし成功率は既存方法とあまり差がでなかった。

提案手法 2 については提案手法 1 と同じく平均最短経路ホップ数は増加してしまうが、成功率、ノード生存率、平均消費電力量は既存方法よりも有効性があると確認できた。

フラッディングは MANET に欠かせない技術である。本研究でフラッディングの改良を行い、既存のフラッディングよりも提案手法の有効性を確認できた。

5.2　今後の課題

今後は、データ転送している最中に端末の移動による再配信も考慮してフラッディングを行う。また、障害物も考慮したフラッディング手法を構想する。

謝辞

本研究を進めるにあたり、指導教官である龍谷大学理工学部情報メディア学科の三好力教授には常日頃から研究についての貴重なアドバイスを頂いたばかりでなく、生活面においても大変配慮していただき厚く御礼申し上げます。
また、研究室の皆様にも公私にわたり様々なアドバイスをしていただき多くのことを学ばせていただきました。ここに深く感謝を申し上げます。
最後にこれまで支えてくれた家族、友人たちに心より感謝申し上げます。

参考文献

[1]. 総務省,平成 20 年通信利用動向調査報告書(世帯編),(2009).

[2]. 阪田 史郎, ユビキタス技術 センサネットワーク, オーム社, (2006).

[3]. D. Johnson, D.A. Maltz and J. Broch, The Dynamic Source Routing Protocol for Mobile Ad hoc Networks, Mobile Ad-hoc Network (MANET) Working Group, IETF,(1998).

[4]. C.E. Perkins and E.M. Royer, Ad hoc on-Demand Distance Vector(AODV) Routing, Mobile Ad-hoc Network (MANET) Working Group, IETF, (1998).

[5]. T. Clausen and P. Jacquet, Optimized Link State Routing Protocol (OLSR), Mobile Ad-hoc Network (MANET) Working Group, IETF, (1998).

[6]. R. Ogier, F.L.Templin, NOKIA, and M.G.Lewis. Topology Dissemination Based on Reverse-Path Forwarding(TBRPF). RFC3684, (2004).

[7]. T. Clausen, The Optimized Link-State Routing Protocol version 2, draft-clausenmanet-olsrv2-00.txt, IETF, (2005).

[8]. Z.J. Haas and M.R. Pearlman, The Zone Routing Protocol (ZRP) for Ad hoc Networks, Mobile Ad-hoc Network (MANET) Working Group, IETF, (1998).

参考資料

1. ソースコード

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define SECOND 10000//繰り返し回数
#define TIMES 1000//時系列
#define POWER 1000//端末バッテリー
#define MAX 900//ノード数
#define SQU 100//縦×横範囲
#define DIS_0 20//既存通信半径
#define DIS_1 15//提案手法 2:ノード電力量 75-50
#define DIS_2 10//提案手法 1:バッテリーが 50%未満の通信範囲//提案手法 2:ノード電力量 50-25
#define DIS_3 5//提案手法 2:ノード電力量 25-0
#define TTL 20//Time To Live

/*パラメータの設定*/
struct Global{
int ttl0;
int ttl1;
int ttl2;
}Global;

/*ノードの点在位置、バッテリー量、一つ前のノードを保存、送信半径ノード*/
struct Node{
int num;
int x;
int y;
int power_0;
int before_0[MAX];
int flag_0;
int send_0[MAX];

int power_1;
int before_1[MAX];
int flag_1;
int send_1[MAX];

int power_2;
int before_2[MAX];
int flag_2;
int send_2[MAX];
}node[MAX];

/*端末間の距離*/
struct distance{
double d;
}dis[MAX][MAX];

/*経路情報を保存*/
struct list{
int next_node_0[MAX];
int Second_0;

int next_node_1[MAX];
int Second_1;

int next_node_2[MAX];
int Second_2;
}list[TTL];


/*データ取得用*/
struct Data_0{
double Success;
double Energy;
double a_cost;
int i;
}Data_0;

struct Data_1{
double Success;
double Energy;
double a_cost;
int i;
}Data_1;

struct Data_2{
double Success;
double Energy;
double a_cost;
int i;
}Data_2;

struct Average_0{
double success;
int s_flag;
double hop;
int h_flag;
double node;
double av_cost;
int avc_flag;
}Average_0[TIMES];

struct Average_1{
double success;
int s_flag;
double hop;
int h_flag;
double node;
double av_cost;
int avc_flag;
```

```
}Average_1[TIMES];

struct Average_2{
double success;
int s_flag;
double hop;
int h_flag;
double node;
double av_cost;
int avc_flag;
}Average_2[TIMES];

/***ノードの位置情報の作成***/
void Node_loc(int in,int des)
{
  int i,j,k;
  for(i=0;i<MAX;i++){
    node[i].x = rand() % SQU;
    node[i].y = rand() % SQU;
    node[i].power_0 = node[i].power_1 =
node[i].power_2 = POWER;
    for(j=0;j<i;j++){
      if(node[i].x == node[j].x && node[i].y ==
node[j].y)  i--;
    }
  }
}


/***二点間の距離***/
void broadcast_0(int in,int des,int ttl,int A)
{
  int i=0,j=0,k=0,xdis,ydis;

  /**保持電力がなければ破棄**/
  if(node[in].flag_0 == 0){
    for(i=0;i<MAX;i++){

      /**二点間の距離を構造体に格納**/
      xdis = (node[in].x - node[i].x) * (node[in].x
- node[i].x);
      ydis = (node[in].y - node[i].y) * (node[in].y
- node[i].y);
      dis[in][i].d = sqrt(xdis + ydis);

      /**二点間の距離が送信可能範囲内なら以下を実
行**/
      if(dis[in][i].d > 0 && dis[in][i].d <= DIS_0 &&
node[in].power_0 > pow(DIS_0,2)){
        if(i == des){
          Data_0.Success++;

          /*最短経路を格納*/
          if(Global.ttl0 >= ttl)  Global.ttl0 = ttl;
        }
        node[in].send_0[k] = i;
        k++;

        /*ttl 値の単位で重複なく list にまとめる*/
        if(A == 0){
          list[ttl+1].next_node_0[Data_0.i] = i;
          Data_0.i++;
        }
        if(A > 0){
          j = 0;
          while(1){
            if(j == Data_0.i){
              list[ttl+1].next_node_0[Data_0.i] =
i;
              Data_0.i++;
              break;
            }
            if(list[ttl+1].next_node_0[j] == i)
break;
            j++;
          }
        }
        node[i].before_0[ttl] = in;
      }
    }
    node[in].flag_0 = 1;
    node[in].send_0[k] = EOF;
  }
}

void broadcast_1(int in,int des,int ttl,int A)
{
  int i=0,j=0,k=0;
  double DIS=DIS_0;

  /**保持電力がなければ破棄**/
  if(node[in].flag_1 == 0){
    for(i=0;i<MAX;i++){

      /**二点間の距離が送信可能範囲内なら以下を実
行**/
      if(node[in].power_1 >= POWER/2)  DIS=DIS_0;
      else  DIS=DIS_2;
      if(dis[in][i].d > 0 && dis[in][i].d <= DIS &&
node[in].power_1 > pow(DIS,2)){
        if(i == des){
          Data_1.Success++;

          /*最短経路を格納*/
          if(Global.ttl1 >= ttl)  Global.ttl1 = ttl;
        }
        node[in].send_1[k] = i;
        k++;

        /*ttl 値の単位で重複なく list にまとめる*/
        if(A == 0){
          list[ttl+1].next_node_1[Data_1.i] = i;
          Data_1.i++;
        }
        if(A > 0){
          j = 0;
          while(1){
            if(j == Data_1.i){
              list[ttl+1].next_node_1[Data_1.i] =
i;
              Data_1.i++;
              break;
            }
            if(list[ttl+1].next_node_1[j] == i)
break;
            j++;
```

```
          }
        }
        node[i].before_1[ttl] = in;
      }
    }
    node[in].flag_1 = 1;
    node[in].send_1[k] = EOF;
  }
}

void broadcast_2(int in,int des,int ttl,int A)
{
  int i=0,j=0,k=0;
  double DIS=DIS_0;

  /**保持電力がなければ破棄**/
  if(node[in].flag_2 == 0){
    for(i=0;i<MAX;i++){

      /**二点間の距離が送信可能範囲内なら以下を実
行**/
      if(node[in].power_2 >= POWER*0.75)
DIS=DIS_0;
      else if(node[in].power_2 < POWER*0.75 &&
node[in].power_2 >= POWER*0.5)  DIS=DIS_1;
      else if(node[in].power_2 < POWER*0.5 &&
node[in].power_2 >= POWER*0.25)  DIS=DIS_2;
      else  DIS=DIS_3;
      if(dis[in][i].d > 0 && dis[in][i].d <= DIS &&
node[in].power_2 > pow(DIS,2)){
        if(i == des){
          Data_2.Success++;

          /*最短経路を格納*/
          if(Global.ttl2 >= ttl)  Global.ttl2 = ttl;
        }
        node[in].send_2[k] = i;
        k++;

        /*ttl 値の単位で重複なく list にまとめる*/
        if(A == 0){
          list[ttl+1].next_node_2[Data_2.i] = i;
          Data_2.i++;
        }
        if(A > 0){
          j = 0;
          while(1){
            if(j == Data_2.i){
              list[ttl+1].next_node_2[Data_2.i] =
i;
              Data_2.i++;
              break;
            }
            if(list[ttl+1].next_node_2[j] == i)
break;
            j++;
          }
        }
        node[i].before_2[ttl] = in;
      }
    }
    node[in].flag_2 = 1;
    node[in].send_2[k] = EOF;
  }
}

/*フラッディング処理*/
void flooding_0(int in,int des,int ttl)
{
  int i;
  Data_0.i = 0;
  if(ttl > 1)  list[ttl].Second_0 -= 1;
  for(i=0;i<=list[ttl].Second_0;i++){
    if(ttl > 1)  in = list[ttl].next_node_0[i];
    broadcast_0(in,des,ttl,i);
  }
  list[ttl+1].Second_0 = Data_0.i;
}

void flooding_1(int in,int des,int ttl)
{
  int i;
  Data_1.i = 0;
  if(ttl > 1)  list[ttl].Second_1 -= 1;
  for(i=0;i<=list[ttl].Second_1;i++){
    if(ttl > 1)  in = list[ttl].next_node_1[i];
    broadcast_1(in,des,ttl,i);
  }
  list[ttl+1].Second_1 = Data_1.i;
}

void flooding_2(int in,int des,int ttl)
{
  int i;
  Data_2.i = 0;
  if(ttl > 1)  list[ttl].Second_2 -= 1;
  for(i=0;i<=list[ttl].Second_2;i++){
    if(ttl > 1)  in = list[ttl].next_node_2[i];
    broadcast_2(in,des,ttl,i);
  }
  list[ttl+1].Second_2 = Data_2.i;
}


/*最短経路のノード電力量を消費*/
void node_cost_0(int des,int ttl,int k)
{
  int i=0,j;
  int
a=0,b,shortest_route_0=0,sou_route_0[MAX],des_rout
e_0[MAX];
  double cost_0=0;

  if(Data_0.Success == 0 && Global.ttl0 == TTL)
Global.ttl0 = 0;

  /*最短経路検索*/
  if(Global.ttl0 != 0 || Data_0.Success > 0){
    des_route_0[0] = des;
    des_route_0[1] = shortest_route_0 =
node[des].before_0[Global.ttl0];
    b=2;
    for(a=Global.ttl0-1;a>0;a--){
      des_route_0[b] =
node[shortest_route_0].before_0[a];
      shortest_route_0 = des_route_0[b];
```

```
      b++;
    }

    j = Global.ttl0;
    for(i=0;i<=Global.ttl0;i++){
      if(j < 0)  break;
      sou_route_0[i] = des_route_0[j];
      //printf("%d → ",sou_route_0[i]);
      j--;
    }

    /*最短経路のノード電力量を消費*/
    for(i=0;i<Global.ttl0;i++){
      cost_0 = pow(DIS_0,2);
      Data_0.a_cost += cost_0;
      node[sou_route_0[i]].power_0 =
node[sou_route_0[i]].power_0 - pow(DIS_0,2);
    }
  }
}

void node_cost_1(int des,int ttl,int k)
{
  int i=0,j;
  int
a=0,b,shortest_route_1=0,sou_route_1[MAX],des_rout
e_1[MAX];
  double cost_1=0;

  if(Data_1.Success == 0 && Global.ttl1 == TTL)
Global.ttl1 = 0;

  /*最短経路検索*/
  if(Global.ttl1 != 0 || Data_1.Success > 0){
    des_route_1[0] = des;
    des_route_1[1] = shortest_route_1 =
node[des].before_1[Global.ttl1];
    b=2;
    for(a=Global.ttl1-1;a>0;a--){
      des_route_1[b] =
node[shortest_route_1].before_1[a];
      shortest_route_1 = des_route_1[b];
      b++;
    }

    j = Global.ttl1;
    for(i=0;i<=Global.ttl1;i++){
      if(j < 0)  break;
      sou_route_1[i] = des_route_1[j];
      //printf("%d → ",sou_route_1[i]);
      j--;
    }

    /*最短経路のノード電力量を消費*/
    for(i=0;i<Global.ttl1;i++){
      if(node[sou_route_1[i+1]].power_1 >=
POWER*0.5){
        node[sou_route_1[i+1]].power_1 =
node[sou_route_1[i+1]].power_1 - (DIS_0*DIS_0);
        cost_1 = pow(DIS_0,2);
      }else{
        node[sou_route_1[i+1]].power_1 =
node[sou_route_1[i+1]].power_1 - (DIS_2*DIS_2);
        cost_1 = pow(DIS_2,2);
      }
      Data_1.a_cost += cost_1;
    }
  }
}

void node_cost_2(int des,int ttl,int k)
{
  int i=0,j;
  int
a=0,b,shortest_route_2=0,sou_route_2[MAX],des_rout
e_2[MAX];
  double cost_2=0;

  if(Data_2.Success == 0 && Global.ttl2 == TTL)
Global.ttl2 = 0;

  /*最短経路検索*/
  if(Global.ttl2 != 0 || Data_2.Success > 0){
    des_route_2[0] = des;
    des_route_2[1] = shortest_route_2 =
node[des].before_2[Global.ttl2];
    b=2;
    for(a=Global.ttl2-1;a>0;a--){
      des_route_2[b] =
node[shortest_route_2].before_2[a];
      shortest_route_2 = des_route_2[b];
      b++;
    }

    j = Global.ttl2;
    for(i=0;i<=Global.ttl2;i++){
      if(j < 0)  break;
      sou_route_2[i] = des_route_2[j];
      //printf("%d → ",sou_route_2[i]);
      j--;
    }

    /*最短経路のノード電力量を消費*/
    for(i=0;i<Global.ttl2;i++){
      if(node[sou_route_2[i+1]].power_2 >=
POWER*0.75){
        node[sou_route_2[i+1]].power_2 =
node[sou_route_2[i+1]].power_2 - (DIS_0*DIS_0);
        cost_2 = pow(DIS_0,2);
      }else if(node[sou_route_2[i+1]].power_2 <
POWER*0.75 && node[sou_route_2[i+1]].power_2 >=
POWER*0.5){
        node[sou_route_2[i+1]].power_2 =
node[sou_route_2[i+1]].power_2 - (DIS_1*DIS_1);
        cost_2 = pow(DIS_1,2);
      }else if(node[sou_route_2[i+1]].power_2 <
POWER*0.5 && node[sou_route_2[i+1]].power_2 >=
POWER*0.25){
        node[sou_route_2[i+1]].power_2 =
node[sou_route_2[i+1]].power_2 - (DIS_2*DIS_2);
        cost_2 = pow(DIS_2,2);
      }else{
        node[sou_route_2[i+1]].power_2 =
node[sou_route_2[i+1]].power_2 - (DIS_3*DIS_3);
        cost_2 = pow(DIS_3,2);
      }
```

```
      Data_2.a_cost += cost_2;
    }
  }
}
/*データ取得*/
void Get_data0(int k)
{
  int i;

  /*残りノード数*/
  for(i=0;i<MAX;i++){
    if(node[i].power_0 >= DIS_0*DIS_0)
Data_0.Energy++;
  }

  /*成功率*/
  if(Data_0.Success >= 1)  Average_0[k].success++;
  /*ホップ数*/
  Average_0[k].hop += Global.ttl0;
  if(Global.ttl0 != 0)  Average_0[k].h_flag++;
  /*ノード生存数*/
  Average_0[k].node += Data_0.Energy;
  /*消費電力量*/
  Average_0[k].av_cost += Data_0.a_cost;
  if(Data_0.a_cost != 0)  Average_0[k].avc_flag++;
}

void Get_data1(int k)
{
  int i;

  /*残りノード数*/
  for(i=0;i<MAX;i++){
    if(node[i].power_1 >= DIS_2*DIS_2)
Data_1.Energy++;
  }

  /*成功率集計*/
  if(Data_1.Success >= 1)  Average_1[k].success++;
  /*ホップ数*/
  Average_1[k].hop += Global.ttl1;
  if(Global.ttl1 != 0)  Average_1[k].h_flag++;
  /*ノード生存数*/
  Average_1[k].node += Data_1.Energy;
  /*消費電力量*/
  Average_1[k].av_cost += Data_1.a_cost;
  if(Data_1.a_cost != 0)  Average_1[k].avc_flag++;
}

void Get_data2(int k)
{
  int i;

  /*残りノード数*/
  for(i=0;i<MAX;i++){
    if(node[i].power_2 >= DIS_3*DIS_3)
Data_2.Energy++;
  }

  /*成功率集計*/
  if(Data_2.Success >= 1)  Average_2[k].success++;
  /*ホップ数*/
  Average_2[k].hop += Global.ttl2;
  if(Global.ttl2 != 0)  Average_2[k].h_flag++;
  /*ノード生存数*/
  Average_2[k].node += Data_2.Energy;
  /*消費電力量*/
  Average_2[k].av_cost += Data_2.a_cost;
  if(Data_2.a_cost != 0)  Average_2[k].avc_flag++;
}


int main(void)
{
  FILE *co,*pr1,*pr2;

  if ((co = fopen("Conventional.txt", "w+")) == NULL)
{
    printf("file open error!!¥n");
    exit(1);
  }

  if ((pr1 = fopen("Proposal-1.txt", "w+")) == NULL)
{
    printf("file open error!!¥n");
    exit(1);
  }

  if ((pr2 = fopen("Proposal-2.txt", "w+")) == NULL)
{
    printf("file open error!!¥n");
    exit(1);
  }

  fclose(co);
  fclose(pr1);
  fclose(pr2);

  if ((co = fopen("Conventional.txt", "a+")) == NULL)
{
    printf("file open error!!¥n");
    exit(1);
  }

  if ((pr1 = fopen("Proposal-1.txt", "a+")) == NULL)
{
    printf("file open error!!¥n");
    exit(1);
  }

  if ((pr2 = fopen("Proposal-2.txt", "a+")) == NULL)
{
    printf("file open error!!¥n");
    exit(1);
  }


  int i,j,k,l,S_node,D_node,ttl;

  srand((unsigned int)time(NULL));

  for(i=0;i<TIMES;i++){
    Average_0[i].success = 0;
    Average_0[i].s_flag = 0;
    Average_0[i].hop = 0;
    Average_0[i].h_flag = 0;
```

```
  Average_0[i].node = 0;
  Average_0[i].av_cost = 0;
  Average_0[i].avc_flag = 0;

  Average_1[i].success = 0;
  Average_1[i].s_flag = 0;
  Average_1[i].hop = 0;
  Average_1[i].h_flag = 0;
  Average_1[i].node = 0;
  Average_1[i].av_cost = 0;
  Average_1[i].avc_flag = 0;

  Average_2[i].success = 0;
  Average_2[i].s_flag = 0;
  Average_2[i].hop = 0;
  Average_2[i].h_flag = 0;
  Average_2[i].node = 0;
  Average_2[i].av_cost = 0;
  Average_2[i].avc_flag = 0;
}

for(l=0;l<SECOND;l++){
  printf("%d¥n",l);

  /***ノードのランダム配置***/
  Node_loc(S_node,D_node);

  for(k=0;k<TIMES;k++){

    /***初期化***/
    ttl=1;
    S_node=EOF;
    D_node=EOF;

    Data_0.Success = 0;
    Data_0.Energy = 0;
    Data_0.a_cost = 0;
    Global.ttl0 = TTL;

    Data_1.Success = 0;
    Data_1.Energy = 0;
    Data_1.a_cost = 0;
    Global.ttl1 = TTL;

    Data_2.Success = 0;
    Data_2.Energy = 0;
    Data_2.a_cost = 0;
    Global.ttl2 = TTL;

    for(i=0;i<MAX;i++){
      node[i].num = i;
      node[i].flag_0 = 0;
      list[i].Second_0 = 0;

      node[i].flag_1 = 0;
      list[i].Second_1 = 0;

      node[i].flag_2 = 0;
      list[i].Second_2 = 0;
    }

    for(i=0;i<MAX;i++){
      for(j=0;j<MAX;j++){
        node[i].before_0[j] = EOF;
        node[i].before_1[j] = EOF;
        node[i].before_2[j] = EOF;
      }
    }

    /***送信元・宛先ノードの決定***/
    do{
      S_node = rand() % MAX;
      D_node = rand() % MAX;
    }while(S_node == D_node);

    /***繰り返し***/
    for(ttl=1;ttl<=TTL;ttl++){
      flooding_0(S_node,D_node,ttl);
      flooding_1(S_node,D_node,ttl);
      flooding_2(S_node,D_node,ttl);
    }
    node_cost_0(D_node,ttl,k);
    node_cost_1(D_node,ttl,k);
    node_cost_2(D_node,ttl,k);

    Get_data0(k);
    Get_data1(k);
    Get_data2(k);
  }
}

for(k=0;k<TIMES;k++){
  if(k==0){
    fprintf(co,"#Conventional");
    fprintf(co,"¥n#No.  Success  Hop  Node
Cost¥n");
    fprintf(pr1,"#Proposal 1");
    fprintf(pr1,"¥n#No.  Success  Hop  Node
Cost¥n");
    fprintf(pr2,"#Proposal 2");
    fprintf(pr2,"¥n#No.  Success  Hop  Node
Cost¥n");
  }

  fprintf(co,"%-2d  ",k+1);
  fprintf(co,"%-7.1f
",(Average_0[k].success/SECOND)*100);

  if(Average_0[k].hop != 0 &&
Average_0[k].h_flag != 0)
    fprintf(co,"%-7.1f
",Average_0[k].hop/Average_0[k].h_flag);
  else  fprintf(co,"%-7.1f  ",0.0);

  fprintf(co,"%-7.0f
",Average_0[k].node/SECOND);

  if(Average_0[k].av_cost != 0 &&
Average_0[k].avc_flag != 0)

fprintf(co,"%-f¥n",Average_0[k].av_cost/Average_0[
k].avc_flag);
  else  fprintf(co,"%-f¥n",0.0);

  fprintf(pr1,"%-2d  ",k+1);
  fprintf(pr1,"%-7.1f
```

```
",(Average_1[k].success/SECOND)*100);

    if(Average_1[k].hop != 0 &&
Average_1[k].h_flag != 0)
      fprintf(pr1,"%-7.1f
",Average_1[k].hop/Average_1[k].h_flag);
    else  fprintf(pr1,"%-7.1f  ",0.0);

    fprintf(pr1,"%-7.0f
",Average_1[k].node/SECOND);

    if(Average_1[k].av_cost != 0 &&
Average_1[k].avc_flag != 0)

fprintf(pr1,"%-f¥n",Average_1[k].av_cost/Average_1
[k].avc_flag);
    else  fprintf(pr1,"%-f¥n",0.0);

    fprintf(pr2,"%-2d  ",k+1);
    fprintf(pr2,"%-7.1f
",(Average_2[k].success/SECOND)*100);

    if(Average_2[k].hop != 0 &&
Average_2[k].h_flag != 0)
      fprintf(pr2,"%-7.1f
",Average_2[k].hop/Average_2[k].h_flag);
    else  fprintf(pr2,"%-7.1f  ",0.0);

    fprintf(pr2,"%-7.0f
",Average_2[k].node/SECOND);

    if(Average_2[k].av_cost != 0 &&
Average_2[k].avc_flag != 0)

fprintf(pr2,"%-f¥n",Average_2[k].av_cost/Average_2
[k].avc_flag);
    else  fprintf(pr2,"%-f¥n",0.0);
  }
  fclose(co);
  fclose(pr1);
  fclose(pr2);
  return 0;
}
```