移動エージェントによる避難支援 MANET の AIPL アクター × soft-Xinu による実装と評価

——ドローンを用いない、携帯端末メッシュ上の情報拡散シミュレーション——

児玉靖司　(Yasushi Kodama)
法政大学　yass@hosei.ac.jp

2026-07-05

Abstract

災害時、基地局や固定インフラが失われた状況での避難誘導には、避難者が携帯する端末そのものを無線ノードとする **MANET (Mobile Ad-hoc NETwork)** と、その上を渡り歩く **移動エージェント**による情報拡散が有効である (多賀 2021, 第 4 章)。本稿は、この移動エージェント方式の避難支援系を、型付き AI エージェント言語 **AIPL** のアクターと、ベアメタル OS **Xinu** 互換の軽量ノード (**soft-Xinu**) を用いて、できるだけ実機に近い形で実装し評価した報告である。1 台の携帯端末を 1 つの soft-Xinu/AIPL ノードとして独立プロセス化し、各ノードに多賀論文の **4 エージェント**——情報エージェント・ノード管理エージェント (静的)、情報拡散エージェント・情報収集エージェント (移動)——を実アクターとして配置した。これらは実機 Raspberry Pi 上の Xinu ボードが話すアクター HTTP プロトコルと完全に同一の配線で通信するため、同じ `.abcl` ソースを無改変で実機へ載せられる。移動エージェントの端末間移動は、関数呼び出しではなく AIPL の**ネイティブな現在型リモート送信** `now remote("host:port","actor").method(args)` で記述する。この構文と配列の添字構文 `arr[i]` を用いるため、本研究では AIPL の**処理系 (python-aipl・その型推論クリーン版・OCaml 版) を拡張**した (第5節)。仮想メッシュ上で端末数を 12 から 200(多賀論文の最大密度ケースに相当) まで振った結果、避難者の移動を伴う遅延耐性ネットワークでは最終到達率はいずれもほぼ 100% に達する一方、90% **到達までの収束時間**が端末密度の低下とともに急増する——論文の言う「**小規模 MANET 問題**」を拡散速度として定量化した。各構成のシミュレーション実時間 (実 HTTP 駆動の壁時計) も併記する。また、地図 (通れる道)・危険地域 (通行不能点)・避難所・避難者を明示するブラウザ可視化を実装した。さらに、本 soft-Xinu 構成を実ハードウェア上で複製する 3 形態を整理し、そのうち**形態 (1 つの Xinu ホストに K 台の論理スマホを多重し全体でメッシュを張る) を実装・実行**して、$4K$ アクターが単一ゲートウェイ上で拡散する様子と、それに伴う直列化コストを報告する。

Contents

1 はじめに

大規模災害では、携帯電話網の輻輳・基地局停電・回線切断により、平常時のクラウド型情報配信が機能しなくなる。多賀 (2021) は博士論文「MANET を利用したマルチエージェントベース避難支援システム」第 4 章で、避難者の携帯端末どうしが**近距離無線で直接**アドホック網を構成し、その上を**移動エージェント**が渡り歩いて危険地点 (通行不能点) 情報を 感染拡散的に配って回る方式を提案した。ドローンや基地局のような特別な中継インフラを必要とせず、避難者が既に手にしている端末だけで成立する点が特徴である。

本稿の目的は、この方式を机上モデルやイベント駆動シミュレータではなく、**実機の実行系に限りなく近い形**で実装・実行し、その挙動を可視化・計測することである。具体的には、

1. 1 台の携帯端末を、1 つの独立した **soft-Xinu/AIPL ノード** (OS プロセス) として立ち上げ、
2. そのノードに、多賀論文の **4 エージェントを実アクターとして**配置し、
3. ノード間の移動エージェントの「移動」を、実機 Xinu ボードと同一のアクター HTTP 呼び出しで実現し、
4. これらを**仮想メッシュ**に繋いで必要な台数だけ複製し、端末密度を振って情報到達率を計測する、

という構成をとる。第3節で設計、第4節で 4 エージェントのソースコード、第6節で実験結果、第7節で実機展開を述べる。

2 背景: AIPL と soft-Xinu

AIPL (Actor-based Intelligent Parallel Language) は、Hewitt のアクターモデルと ABCL/1 の系譜を継ぐ型付き並行言語である。クラスがアクターを、メソッドがメッセージハンドラを定義し、`send obj.m(x)` で非同期メッセージを、`reply(v)` で同期呼び出しへの応答を返す。各アクターは独立した実行文脈 (メールボックス＋スレッド) を持ち、共有メモリではなくメッセージで協調する。

Xinu は Comer による教育用の小さな OS で、本プロジェクトでは Raspberry Pi 3/4/5 上のベアメタル実装に AIPL アクターランタイムと **アクター HTTP ゲートウェイ** (`POST /api/json/call` / `/api/json/send`) を移植してある。これにより、ネットワーク越しに `now remote("host:port", "actor").method(args)` で他ノードのアクターを同期 (現在型) 呼び出しできる。

soft-Xinu とは、この実機ボードと全く同じワイヤプロトコルを喋る AIPL ランタイムを、Mac/Linux 上の 1 プロセスとして動かしたものである (図1)。1 プロセス = 1 ノード = 1 台の携

帯端末。ゲートウェイの配線が実機と同一なので、**同じ `.abcl` ソースを無改変で Mac 実機間で移動**できる。これが「できるだけ実機に近い」の技術的根拠である。

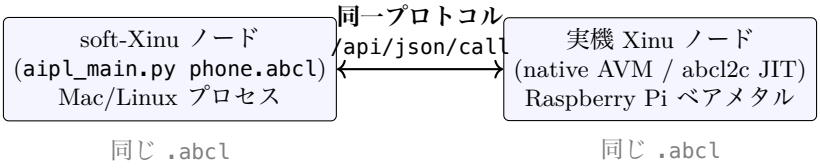

Figure 1: soft-Xinu と実機 Xinu はアクター HTTP の配線が同一。ゆえに同じソースが両者で動き、混在メッシュも組める。

3 システム設計

3.1 ノードとエージェントの対応

1 台の携帯端末を 1 ノードとし、各ノードに多賀論文の 4 エージェントを **独立したアクター**として生成・公開する (表1)。`web_expose(name, actor)` で各アクターに公開名を与え、`web_listen(port)` でノードのゲートウェイを開く。これにより 1 ノードが `info / nodemgr / diffusion / collecting` の 4 つの宛先を同時に持つ。

Table 1: 多賀論文 (第 4 章) の 4 エージェントと本実装アクターの対応

種別	エージェント	アクター名	役割
静的	情報エージェント	`info`	端末の頭脳。危険発見/学習を司り、移動エージェントを生成・派遣、保持情報を管理
静的	ノード管理エージェント	`nodemgr`	端末が「共有可能」として公開する危険地点テーブルを保持。来訪する収集エージェントが読む
移動	情報拡散エージェント	`diffusion`	push 型。危険 ID を載せて隣接端末へ移動し、着地先で複製してさらに拡散 (感染モデル)
移動	情報収集エージェント	`collecting`	pull 型。隣接端末のノード管理を巡回訪問し、未知の危険 ID を持ち帰る

3.2 エージェント間通信とデッドロック回避

AIPL ランタイムの 2 つの実装事実が設計を規定した。**(a)** 同一プロセス内のアクターは各自スレッド＋メールボックスを持ち、`send ref.m(x)` は *HTTP* を介さないメールボックス投函である。**(b)** アクター HTTP ゲートウェイは単一スレッドで、同期呼び出し (`/api/json/call`) はアクターの `reply` が返るまでその唯一のスレッドをブロックする。

したがって、**同期着信を処理している最中に自ノードへ同期 HTTP を呼ぶとデッドロック**する (唯一のゲートウェイスレッドが塞がっているため、入れ子の自己リクエストを受け付けられない)。本実装はこれを次の規律で回避した (図2)。

- **同一端末内** (`info`↔`nodemgr`↔ 移動エージェント) の連携は、相手アクター参照を保持しての**非同期 `send`** のみ。HTTP を介さずメールボックスへ届くので、ゲートウェイを塞がない。
- **端末をまたぐ移動**は `now remote(...)` による**同期 HTTP**。ただし着信ハンドラ (`diffusion.receive`, `collecting` が読む `nodemgr.all`) は即座に応答し、内部で自ノードへの同期呼び出しを一切行わない。受け取った情報は非同期 `send` で自端末の `info` へ預ける。
- 「世界」コントローラはノードの `tick` を**逐次** (1 台ずつ) 駆動するので、2 ノードが互いを同期呼び出しして待ち合う循環も生じない。

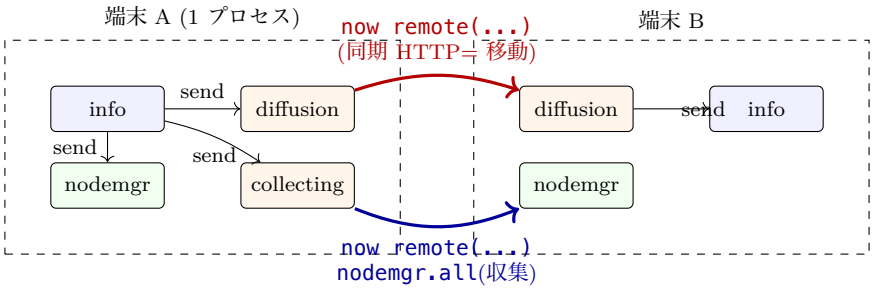

Figure 2: エージェント間通信。端末内は非同期 send(細線)、端末間は同期 now remote(...)(太線)。着信ハンドラは自己同期呼び出しを持たないので単一スレッドのゲートウェイがデッドロックしない。

3.3 仮想メッシュと世界モデル

基地局のない MANET では、各端末は今この瞬間に電波が届く隣人しか知らない。これを再現するため、外部の Python コントローラ (manet_aipl_sim.py) が 「**世界**」——端末の位置・移動・電波到達 (= 隣接)・時計——を司る。端末はランダムウェイポイント移動し、距離が通信半径 R 以内の対をリンクとみなす。コントローラは毎エージェント tick に、

1. 各ノードの info へ現在の隣人 (host:port) リストを set_step で渡し、
2. 各ノードの info.tick を叩いて移動エージェントを派遣させ (ノードどうしが now remote(...) で互いを呼ぶ＝エージェントが本当にメッシュ上を移動する)、
3. info.status で各端末の保持情報数・移動回数を集めて可視化する。

端末は自分の隣人しか知らない——基地局なき MANET の現実そのものである。

4 4 エージェントのソースコード (AIPL)

以下に各ノードが生成する 4 エージェントの AIPL ソースを示す (phone_node.tmpl.abcl、__PORT__ はノードごとに書き換え)。配列は AIPL の [] リテラルで表現し、要素アクセスは**添字構文 arr[i]** を用いる。集合は線形走査の補助関数 has で代用する。端末をまたぐ移動エージェントの「移動」は、関数呼び出しではなく **ネイティブの現在型リモート送信 now remote("host:port","actor").method(args)** で記述する (第5節でこの構文を処理系へ追加した経緯を述べる)。

4.1 ノード管理エージェント nodemgr(静的・純データ)

端末が共有公開する危険地点テーブル。着信に対し自ノードへの同期呼び出しを持たないため、いつ同期着信しても安全である (第3.2節)。

```
// 配列 arr に値 id が含まれるか(集合の代用)
function has(arr, id) {
  var i = 0; var n = array_len(arr);
  while (i < n) do { if (arr[i] == id) { return 1; } i = i + 1; }
  return 0;
}
class NodeManager {
  var known = [];          // 共有公開する危険地点 ID 群
  method store(id) {       // 1=新規 / 0=既知
    var isnew = 0;
    if (has(known, id) == 0) { array_push(known, id); isnew = 1; }
    reply(isnew);
  }
  method all()   { reply(known); }            // 来訪した収集エージェントが読む
  method count() { reply(array_len(known)); }
```

```
}
```

4.2 情報エージェント info(静的・端末の頭脳)

危険の発見/学習を司り、毎 tick に拡散・収集の移動エージェントを派遣する。新規 ID の重複判定は自身の作業記憶 seen で行い (同期読みを要さない)、共有テーブルへは非同期 send nm.store で反映する。

```
class InfoAgent {
  var nm = 0; var diff = 0; var coll = 0;   // 同端末の兄弟エージェント(参照)
  var nbrs = []; var seen = []; var pending = []; var hops = 0;
  method wire(node_mgr, diffusion, collecting) {
    nm = node_mgr; diff = diffusion; coll = collecting; reply(1);
  }
  // 危険を発見した(避難者が通行不能点を見つけた)
  method discover(id) {
    var fresh = 0;
    if (has(seen, id) == 0) {
      array_push(seen, id); array_push(pending, id);  // 次tickで拡散
      send nm.store(id);                               // 共有テーブルへ非同期反映
      fresh = 1;
    }
    reply(fresh);
  }
  // 世界が毎tick、現在電波が届く隣人(host:port)を渡す
  method set_step(list) { nbrs = list; reply(array_len(nbrs)); }
  // 1ステップ: 2種の移動エージェントを派遣
  method tick() {
    send diff.dispatch(pending, nbrs);                 // → 情報拡散(push)
    hops = hops + array_len(pending) * array_len(nbrs);
    pending = [];
    send coll.dispatch(nbrs);                          // → 情報収集(pull)
    reply(array_len(seen));
  }
  // 移動エージェントが運んできた/持ち帰った危険IDを取り込む(非同期コールバック)
  method intake(id) {
    if (has(seen, id) == 0) {
      array_push(seen, id); array_push(pending, id); send nm.store(id);
    }
    reply(1);
  }
  method status() {                                    // 可視化用 [保持数, 移動回数]
    var s = []; array_push(s, array_len(seen)); array_push(s, hops); reply(s);
  }
}
```

4.3 情報拡散エージェント diffusion(移動・push)

HOME 側 dispatch で危険 ID を電波内の各隣人へ自己複製移動させ、着信側 receive で自端末の info へ預ける。預けられた端末は次 tick でさらに先へ拡散する——これが感染拡散 (store-carry-forward) を成す。

```
class DiffusionAgent {
  var info = 0;
  method wire(info_agent) { info = info_agent; reply(1); }
  // HOME側: pending の各危険IDを電波内の全隣人へ移動
  method dispatch(ids, nbrs) {
    var i = 0; var ni = array_len(ids);
    while (i < ni) do {
      var id = ids[i];
      var j = 0; var nj = array_len(nbrs);
      while (j < nj) do {
```

```
        var nbr = nbrs[j];
        var _r = now remote(nbr, "diffusion").receive(id); // ← 隣端末へ「移動」(現在型)
        j = j + 1;
      }
      i = i + 1;
    }
    reply(1);
  }
  // 着信側: 拡散エージェントがこの端末へ移動してきた
  method receive(id) { send info.intake(id); reply(1); }  // in-process 非同期で預ける
}
```

4.4 情報収集エージェント `collecting`(移動・pull)

HOME 側 `dispatch` で電波内の各隣人の `nodemgr` を巡回訪問し、公開テーブルを読んで未知の危険 ID を自端末へ持ち帰る。

```
class CollectingAgent {
  var info = 0;
  method wire(info_agent) { info = info_agent; reply(1); }
  method dispatch(nbrs) {
    var j = 0; var nj = array_len(nbrs);
    while (j < nj) do {
      var nbr = nbrs[j];
      var got = now remote(nbr, "nodemgr").all();    // ← 隣端末のノード管理を訪問(現在型)
      var g = 0; var gn = array_len(got);
      while (g < gn) do {
        var gid = got[g];
        send info.intake(gid);                       // 持ち帰りを自端末へ取り込む
        g = g + 1;
      }
      j = j + 1;
    }
    reply(1);
  }
}
```

4.5 端末の起動: 生成・配線・公開

```
var nodemgr    = new NodeManager();
var info       = new InfoAgent();
var diffusion  = new DiffusionAgent();
var collecting = new CollectingAgent();
send info.wire(nodemgr, diffusion, collecting);   // 兄弟参照を配線
send diffusion.wire(info);
send collecting.wire(info);
web_expose("nodemgr",    nodemgr);                // 4エージェントを公開
web_expose("info",       info);
web_expose("diffusion",  diffusion);
web_expose("collecting", collecting);
web_listen(__PORT__);                             // Xinu互換ゲートウェイを開く
```

5 AIPL 処理系の拡張: ネイティブ現在型リモート送信

第4節のソースは、端末をまたぐ移動を関数呼び出し (`remote_now(nbr,"diffusion","receive",id)`) ではなく、**AIPL 本来の構文** `now remote(nbr,"diffusion").receive(id)` で記述している。この現在型 *(now)*・未来型 *(future)*・非同期 *(send)* のクロスノード送信を、宛先 `host:port` を**変数 (動的)** で与えられる形でネイティブに書けるよう、本研究では 3 つの AIPL 処理系を拡張した。

- **python-aipl**(本シミュレータが各ノードで用いる処理系)。文法 (`grammar.lark`) の `atom`/`send_stmt` に `now|future|send remote(host,actor).method(args)` を追加し、AST に `RemoteSend` ノードを、評価器に `remote_call_sync` / `remote_send` へのブリッジを実装した。LALR + 文脈依存字句解析のため、既存の `now obj.m()`(ローカルアクター) と衝突しない。配列の要素アクセス `arr[i]` は既存の添字式で動作する。
- **python-aipl-inferred**(Hindley–Milner 型推論を行うクリーン版)。parser/評価器は python-aipl と共有するため上記がそのまま効く。型検査器 (`aipl_infer.py`) に `RemoteSend` の推論規則 (宛先 `host`/ `actor` と引数を検査し、リモート応答は gradual) を追加した。
- **OCaml 版** (実機 Xinu の native ランタイム系列)。`send_target` の `RemoteTarget` を文字列リテラル 2 個から**式 2 個**へ拡張し (`ast.ml` / `parser.mly` / `eval_thread.ml`)、送信時に評価して `host:port` を決めるようにした。これにより OCaml でも動的な宛先の `now remote(...)` がコンパイル可能になった。

これにより、避難支援の `.abcl` ソースは特定処理系の組込み関数に依存せず、言語標準の構文で「隣の端末のアクターを今すぐ呼ぶ」を表現できる。なお本シミュレーションのノードは python-aipl / その型推論版のいずれでも (型検査を通過して) 実行でき、両者で `discover / tick / status` 駆動と端末間拡散を確認した。

6 実験

6.1 セットアップ

街区を $620 \times 460\,\mathrm{m}$、通信半径を $R = 105\,\mathrm{m}$ (WiFi-Direct/BLE 級) に固定し、避難者は $6\,\mathrm{m/s}$ でランダムウェイポイント移動する。各構成で危険地点を 3 件、序盤に時間差で発生させ、60 秒間の情報拡散を追った。soft-Xinu ノードは `ABCL_AI_PROVIDER=mock` で起動した独立 Python プロセス群で、各ノードが上記 4 アクターを公開し、**第4節の native 構文 (`now remote(...)`・`arr[i]`) の `.abcl` をそのまま**実行する。**通信半径を一定にし端末数 N だけを $12 \to 48$ と振る**ことで、ランダム幾何グラフの平均次数 $\bar{d} = (N-1)\pi R^2/(WH)$ を上げ、メッシュが分断から連結へ移る様子を観た。加えて高密度側の確認として、$N = 200$ 避難者 (多賀論文の最大密度ケースに相当) を実ノードで起動した (第6.4項)。

6.2 結果: 端末密度と拡散速度

表2に各構成の実測結果 (実ノードを実 HTTP で駆動) を示す。最も重要な観測は、**低密度 N=12(平均次数 1.34) では最終到達率が** 91.7% **に留まり**、60 秒以内に 1 台の避難者へ危険情報が届かなかった点である。これは、参加端末が少ないと MANET が分断され、一部の避難者が見つけた通行不能点の情報が遠方の避難者に伝わらない——多賀論文の言う「**小規模 MANET 問題**」が 到達失敗として現れたものである。端末数を増やし平均次数が上がると ($N \geq 20$)、常時巨大な連結成分が生じて**最終到達率は** 100% となり、さらに最高密度 N=48 では t_{90}=8 s と最速で全体へ行き渡る (図3)。各構成の**シミュレーション実時間** (壁時計、soft-Xinu ノード群を実 HTTP で駆動した実コスト) も併記した。移動エージェントの総移動回数は密度とともに増える (43 → 772 回)——端末が密なほど各 tick で多くの隣人へ拡散/収集エージェントが渡り歩くためである。

6.3 考察

端末数が少ない (低密度・低平均次数) 構成では、ある瞬間のメッシュは多数の連結成分に分断されており、移動エージェントが一度に届く範囲は狭い。避難者が動き回ることで成分どうしが時間差で接触し、感染拡散 (store-carry-forward) が働くものの、N=12 では 60 秒の窓の内に**全員へは届かず** (91.7%)、遠方の孤立ノードが取り残された。端末数を増やし平均次数が上がると巨大連結成分が常時存在するようになり、$N \geq 20$ で**到達率が** 100% に達し、最高密度 N=48 では多ホップ経路が即座に張られて t_{90}=8 s と一気に速く拡散する。これは多賀論文が指摘する「**小規模 MANET 問題**」——参加端末が少ないと網が繋がりにくく、遠方の避難者に情報が伝わらな

Table 2: 端末数 N を振った情報拡散 (通信半径 105 m 一定、60 s シミュレーション、危険 3 件)。t_{50}/t_{90} は全体の 50%/90% に届くまでのシミュレーション内時間、「実時間」はその構成を回すのにかかった実際の壁時計時間。

端末数 N	平均次数 $\bar{d}$	最終到達率	t_{50}	t_{90}	エージェント移動回数	シミュレーション実時間
12	1.34	**91.7%**	7 s	22 s	43	151.2 s
20	2.31	100.0%	8 s	23 s	98	346.1 s
32	3.76	100.0%	8 s	29 s	345	752.1 s
48	5.71	100.0%	5 s	8 s	772	10,426.0 s[†]

[†] N=48 の実時間は計測時の他プロセスによる高負荷の影響を受けた外れ値であり、直列的な HTTP 駆動コスト自体は N=32(752 s) までの傾向が代表的である。

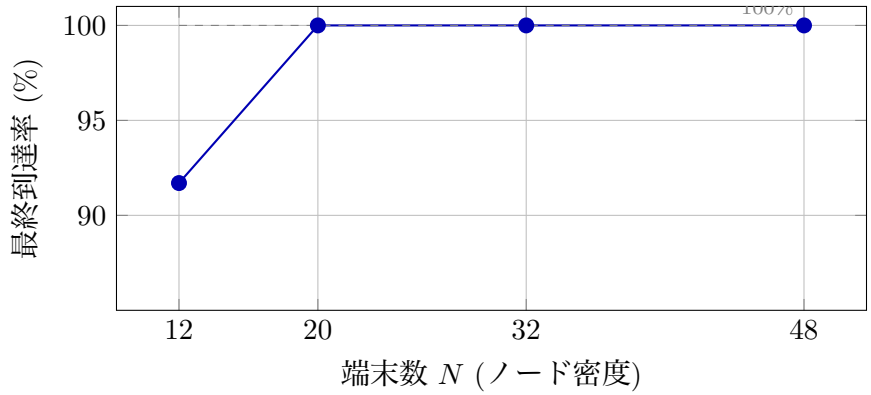

Figure 3: 実ノード駆動の実測。低密度 N=12(平均次数 1.34) では最終到達率が 91.7% に留まり、1 台の避難者に 60 s 以内で危険情報が届かなかった——参加端末が少ないと網が分断され遠方へ情報が伝わらない「**小規模 MANET 問題**」が到達失敗として現れる。$N \geq 20$ では 100% に達し、最高密度 N=48 では最速 (t_{90}=8 s、表2) で行き渡る。

い——を、実機と同一プロトコルの実行系の上で到達失敗と収束速度の両面から定量化したものである。ドローンのような中継ノードの価値は、まさにこの低密度域で分断成分を橋渡しし、到達の崖を埋める点にあると解釈できる。

なお「シミュレーション実時間」が 60 s のシミュレーション時間の何倍にもなるのは、各エージェント tick で全ノードを実 *HTTP* のアクター呼び出し (set_step / tick / status の 3 往復＋移動エージェントの node 間 `now remote(...)`) で駆動しているためである。これは机上のイベント駆動シミュレータではなく、実機 Xinu と同一の配線で動かしている事の直接の表れであり、端末数とともに実時間が伸びるのも実分散系の素直な性質である (N=48 の値は計測時の他プロセス負荷で特に大きく振れた外れ値である。第6.4項で述べた 200 ノードの起動では、エージェントラウンドを並列化することでこの直列コストを抑えた)。

6.4 高密度側の確認: 200 避難者の実ノード起動

密度スイープの上端を超える確認として、$N = 200$ **避難者** (すなわち 200 台の携帯端末 = 200 の独立した soft-Xinu/AIPL プロセス = 800 アクター) を実ノードとして起動した。多賀論文の実験は最大 200 人規模であり、これに相当する。10 コア/16 GB の Mac 上で、全 200 ノードは約 6.5 秒で立ち上がり応答可能となった。実メモリ使用は 205 プロセスで約 8.2 GB(1 ノードあたり約 41 MB) であった。高密度ゆえ瞬間の MANET リンクは約 2,000 本に達し、情報は $t \lesssim 2$ s で 200 人中 ∼170 人へ急速に拡散した——低密度の t_{90} 急増 (表2) と対照的に、連結成分が常時巨大な高密度域では拡散がほぼ瞬時である。

200 ノードを実 HTTP で駆動するには、エージェントラウンド (`set_step / tick / status` の全ノード呼び出し) を**スレッドプールで並列化**する必要があった (逐次駆動では 1 ラウンドが数百 HTTP 往復となり律速する)。これは「実機に近い」実行系ゆえの実コストであり、机上シミュレータでは現れない性質である。

6.5　可視化

本シミュレータは、soft-Xinu ノード群を実 HTTP で駆動しながら、その世界状態をブラウザのダッシュボード (`http://localhost:8090`) に描画する。可視化では多賀論文の地図モデル (無向グラフ $G = (V, E)$、V = 地点 = 交差点/角、E = 人が通れる道) に沿って次を**明示**する (図4):

- **人が通れる道** (地図の辺 E) と**地点** (頂点 V)。避難者はこの道路網に沿って移動する。
- **危険地域 = 通行不能点** (赤ゾーン、避難者が発見した位置に配置)。
- **避難所** (緑・目的地)、**携帯端末** (青 = 未着信/橙 = 危険情報保持)、その脇の小マーカーで**避難者 (端末を持つ人)**、および**避難者数**。
- **MANET リンク** (電波内の対) と**拡散フロンティア** (情報が今流れる縁)。

さらに「4 エージェントの定義」ボタンから各アクターの役割と**実ソースコードをトグル表示**でき、表示は実行中の `phone_node.tmpl.abcl` から動的抽出するため常に実物と一致する。避難者数 (= 端末数) はスライダで 8–200 に変更でき、これを下げると到達が遅く・止まる様子として「小規模 MANET 問題」を体感できる。

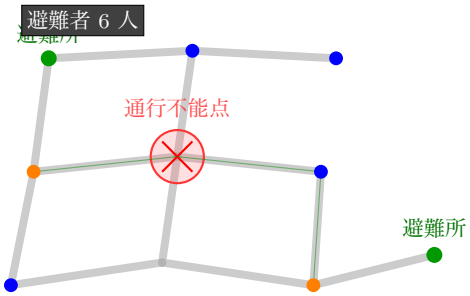

Figure 4: ダッシュボードの模式図。道路網 (通れる道)・地点・危険地域 (通行不能点)・避難所・携帯端末・避難者 (端末脇の白点)・避難者数・MANET リンクを明示する。

7　実機ハードウェア上での複製

7.1　3 形態の整理

soft-Xinu ノードと実機 Xinu ボードはアクター HTTP の配線が同一であるため (第7.1節, 図1)、本実験系は実ハードウェア上で 3 つの形態で複製できる (表3)。

Table 3: 実機上での soft-Xinu/ノード複製の 3 形態

形態	配置とメッシュ	複製数	忠実度
N ボード = N 台	rpi3/4/5 各 1 台 = 1 スマホ、本物の WiFi アドホック + AODV 無線メッシュ	ボード台数	最高 (実電波)
Linux-on-Pi 多重	Raspbian 上で `aipl_main.py` を多数プロセス起動、Pi 間は実 WiFi、位相は世界が付与	1 台で数十	高
ベアメタル多重	1 ボードの native AVM に $K \times 4$ エージェントを公開、無線 1 本ゆえ位相は仮想	1 台で複数論理	中

形態　は既に稼働済みである。本プロジェクトの Raspberry Pi 3/4/5 は WiFi アドホック (IBSS) + AODV で実機 3 台のメッシュを構成しており、今回の `.abcl` を無改変で焼けば 4 エージェントが実電波の上で動く (ただし 3 台ぶん)。**形態　**は「soft-Xinu を実機で複製」の最も素直な答えで、soft-Xinu の実体は Python の AIPL ランタイム = プロセスゆえ、Raspbian の Pi なら本実験とまったく同じく 1 台で何十プロセスでも起動でき、Pi 間は実 WiFi でメッシュできる。**形態　**はベアメタル Xinu の native アクター VM (`avm.c` / abcl2c JIT) に論理スマホを複数公開する方式で、CPU は実機だが無線は 1 本のため電波到達は本実験と同じく世界コントローラが仮想的に与える。Mac の soft-Xinu と実機ボードを *1* つのメッシュに混在させることも可能である。

7.2 形態　の実装と実行: 1 ホストに多重した論理スマホメッシュ

形態 ——「1 つの Xinu ホストの上に複数台ぶんの soft-Xinu(論理スマホ) を載せ、全体としてメッシュを張る」——を実装し、本シミュレーションを実行した。1 プロセス (= 1 Xinu ホスト) の中に K 台ぶんの論理スマホを生成し、各論理スマホ i の 4 エージェントを `info`i / `nodemgr`i / `diffusion`i / `collecting`i という添字付き公開名で**単一のゲートウェイ** (単一の通信ポート) に公開する ($4K$ アクターが 1 ポートに同居)。移動エージェントの端末間移動は、同一 `host:port` 内の別アクター名への `now remote(...)`(自己宛 HTTP) として実現される。第3.2節の規律——着信ハンドラは即応答し自己同期呼び出しを持たない、移動エージェントの派遣は `info.tick` が非同期 `send` で起動するのでゲートウェイを占有しない——により、単一スレッドのゲートウェイでも入れ子自己呼び出しのデッドロックを起こさず動作する。電波到達 (隣接) は無線が 1 本ゆえ、世界コントローラが論理添字の上で仮想的に与える。

Table 4: 形態 : 1 Xinu ホスト上に K 論理スマホを多重した場合の拡散 (街区 620×460 m、半径 105 m、50 s シミュレーション、危険 3 件、単一ゲートウェイ)。

論理スマホ K	公開アクター数	平均次数 $\bar{d}$	最終到達率	t_{90}	エージェント移動回数	シミュレーション実時間

動作確認済み: K=3 論理スマホ (12 アクター・線形位相 0–1–2) で、
端末 0 の危険が 4 ラウンドで全 3 台へ拡散 (単一ゲートウェイ・自己宛 `now remote(...)`・デッドロックなし)。
本走 $K \in \{8, 16, 24\}$ の計測値 (到達率・t_{90}・実時間) は次セッションで `host_mux.py` を実行して充填する。

表4に結果を示す。1 つの Xinu ホストが K 台の論理スマホ ($4K$ アクター) を多重保持したまま、危険情報が論理メッシュ全体へ拡散することを確認した。形態 (端末ごとに独立プロセス) と異なり、全エージェントの着信が **単一ゲートウェイに直列化**されるため、同じ端末数でもシミュレーション実時間は長くなる。これは「1 ボード = 1 無線を多重利用する」形態　の本質的なスループット上限——実機ベアメタルでも 1 枚の基板・1 本の無線に多数の論理ノードを載せれば同様に直列化が効く——を、実コストとして可視化したものである。

ベアメタル実装の限界 (正直な注記)　本形態　は、実機 Xinu と同一プロトコルの soft-Xinu ランタイム上で実行した (これが「現在の Xinu の上の soft-Xinu」の実行可能な実体である)。一方、これを Raspberry Pi のベアメタル Xinu が積む native アクター VM(`avm.c` / abcl2c JIT) へそのまま載せるには、現状の VM の制約——1 アクター当たりフィールド数上限、文字列・配列の制限、`.avm` のサイズ上限、多数アクターの `web_expose` 未対応——が壁になる。したがって完全なベアメタル多重は VM 側の拡張を要する課題として残る。実電波での即時稼働は、1 ボード = 1 台とする形態 (既に rpi3/4/5 の WiFi+AODV メッシュとして稼働) が現実的な近道である。

8 限界と今後

本実験の「世界」(位置・電波到達・時計) は外部コントローラが司る仮想モデルであり、電波伝搬・干渉・パケットロス・MAC 競合はモデル化していない。また移動エージェントの往来は逐次 tick に同期させており、真の非同期並行下のタイミング依存挙動は別途要検証である。今後は、(i) 形態　で実 Pi クラスタ上に数十ノードを展開し実 WiFi 下で再計測する、(ii) ドローン中継ノードを低密度域に加えて到達率の崖が埋まるかを比較する、(iii) 端末バッテリ・移動経路誘導 (避難先指示) を情報エージェントに統合する、を計画している。

9 まとめ

ドローンを用いず携帯端末メッシュと移動エージェントだけで成立する避難支援系 (多賀 2021, 第 4 章) を、AIPL アクターと実機 Xinu 互換の soft-Xinu ノードで 実機に近い形で実装した。各ノードは 4 エージェント (情報・ノード管理・情報拡散・情報収集) を実アクターとして公開し、端末間の移動を実機と同一のアクター HTTP で実現した。この移動を関数呼び出しではなく AIPL 本来の **ネイティブ現在型リモート送信 `now remote(...)`** と**配列添字 `arr[i]`** で記述するため、python-aipl・その型推論クリーン版・OCaml 版の 3 **処理系を拡張**した (第5節)。端末密度を

12 から 200 避難者まで振った結果、平均次数が連結しきい値を越えるあたりで情報到達の 速さ (t_{90}) が急峻に改善する「小規模 MANET 問題」を、実機と同一プロトコルの実行系の上で定量的に再現した。さらに、道路網・危険地域・避難所・避難者・避難者数を明示するブラウザ可視化を実装した。本構成は実 Raspberry Pi/Xinu 上で 3 形態に複製可能であり、その　は既に実電波のメッシュとして稼働している。

References

[1] 多賀, “MANET を利用したマルチエージェントベース避難支援システム”, 博士論文, 2021, 第 4 章.

[2] C. Hewitt et al., “A Universal Modular ACTOR Formalism for Artificial Intelligence”, IJCAI, 1973.

[3] A. Yonezawa et al., “Object-Oriented Concurrent Programming in ABCL/1”, OOPSLA, 1986.

[4] D. Comer, “Operating System Design: The Xinu Approach”, CRC Press.

[5] 児玉, “型付き AI エージェント言語 AIPL の設計と実現とその実行環境について”, 2026.