❖ DMI：計算資源の動的な参加/脱退をサポートする大規模分散共有メモリインタフェース ❖

原健太朗, 田浦健次朗, 近山隆（東京大学）

2009.8.6

発表の流れ

(1) 序論
(2) システムデザイン
(3) 関連研究
(4) 参加/脱退対応のプロトコル
(5) 性能評価
(6) 結論

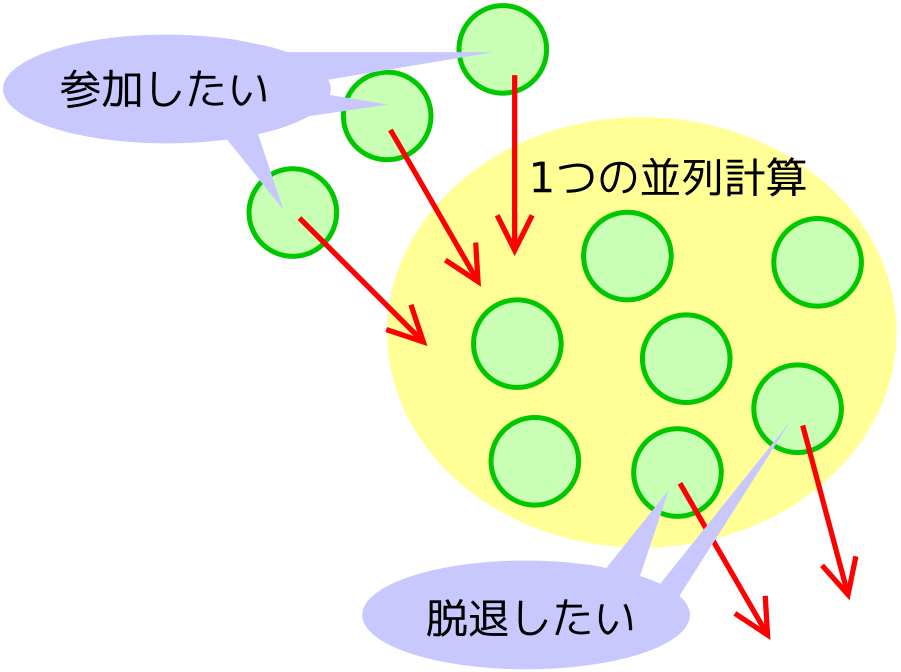
1. 序論

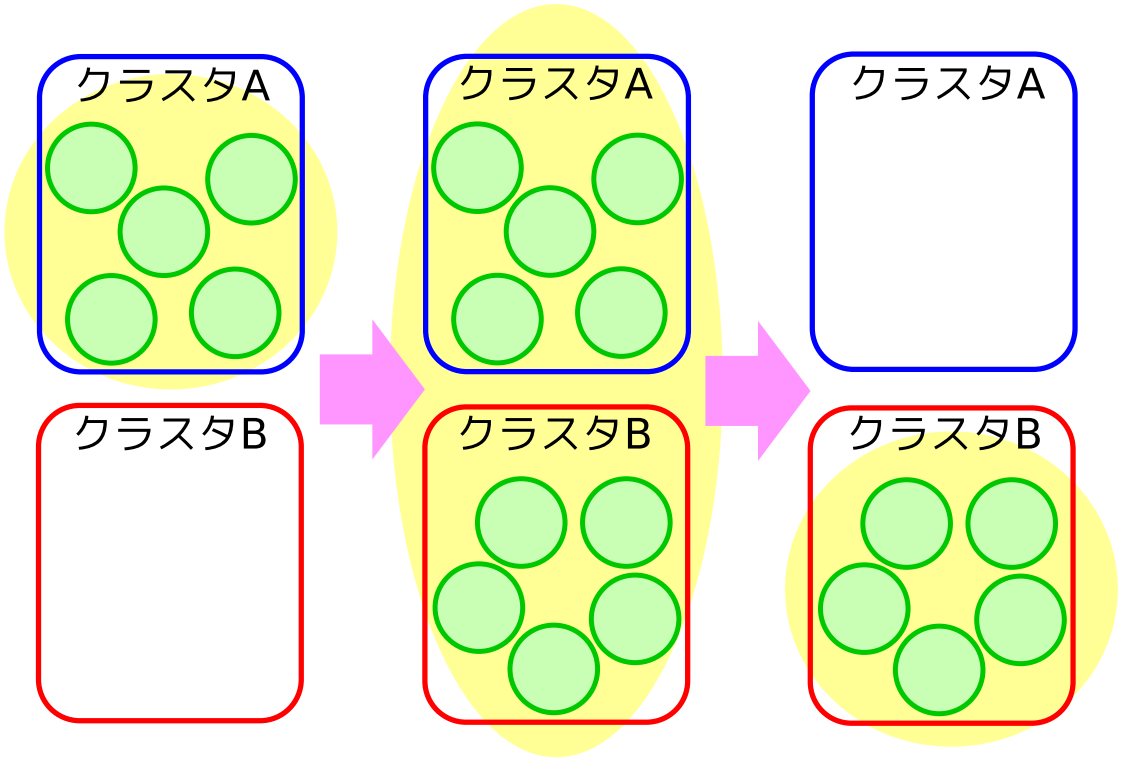
背景

- 並列分散コンピューティングの発展
  - → 産業界の応用分野での並列分散アプリの多様化・高度化
  - → 計算資源の大規模化・高性能化
- 基盤となる並列分散処理系への要請も多様化
  - → 計算資源の動的な参加/脱退のサポート
  - → 複雑なネットワーク構成への対応
  - → 耐故障
  - → …

計算資源の動的な参加/脱退のサポート (1)

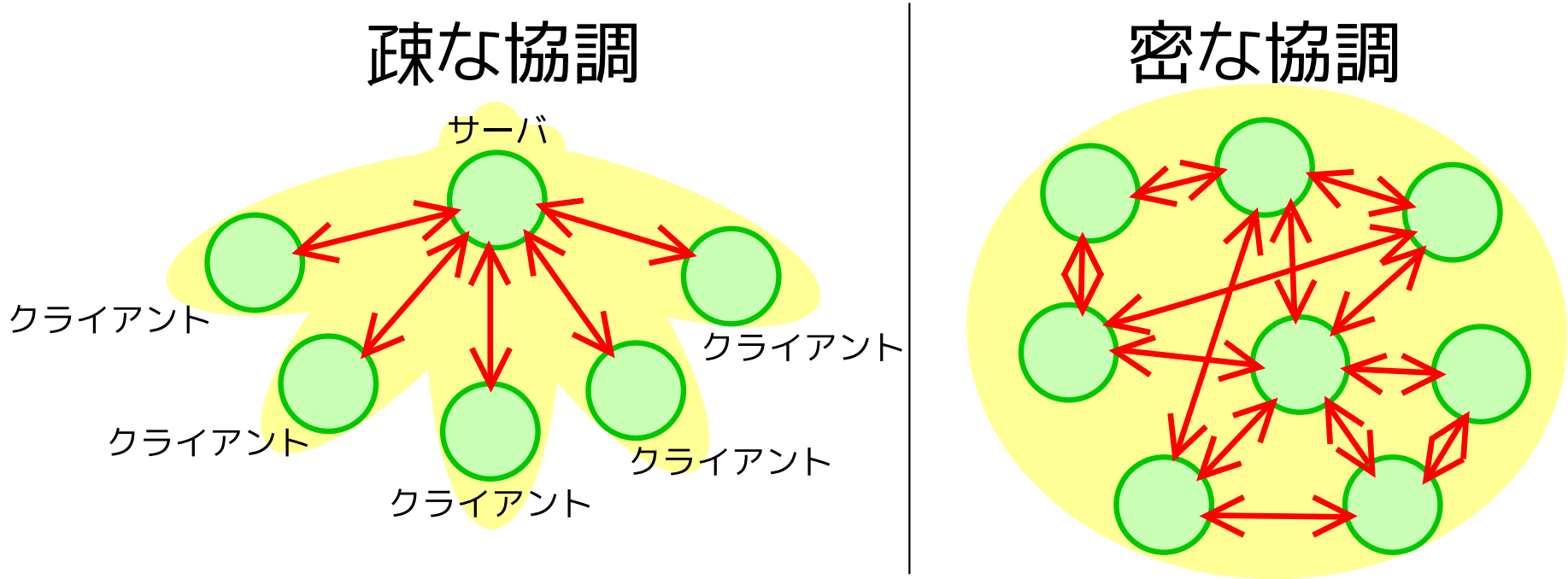
- 現状：計算資源は個人のものではない
  - → クラスタの運用ポリシー，課金制度，…
- 要請：
  - → 参加/脱退を越えて 1 つの並列計算を継続実行
  - → 計算環境の動的マイグレーション





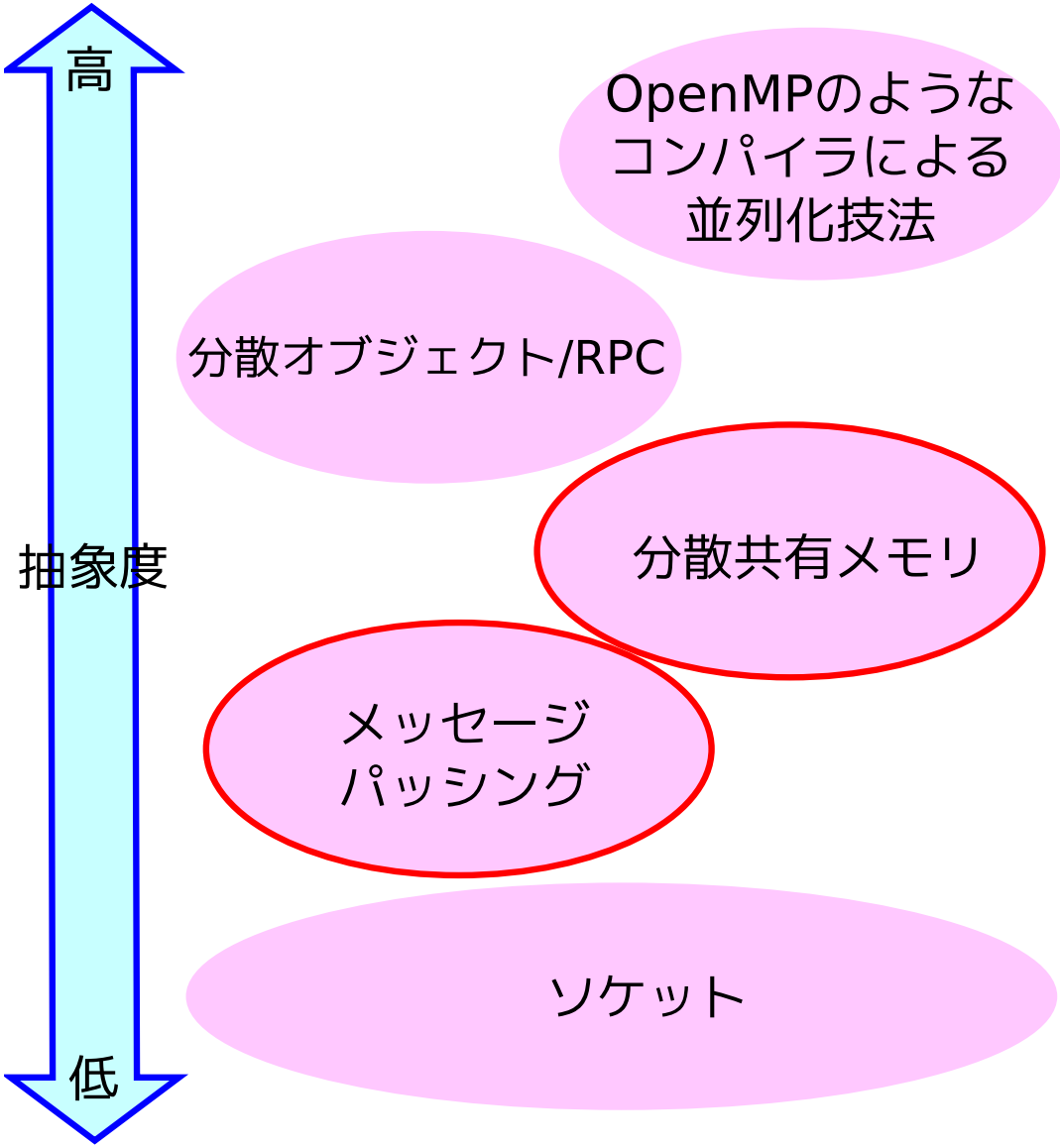
計算資源の動的な参加/脱退のサポート (2)

- 例：クライアント・サーバ方式
  - → 特定の計算資源に負荷が集中するためスケーラブルでない
  - → 計算資源どうしが疎に結び付くモデルの上で効率的に実行可能な処理は限られる [Taura,2001]
- 多数の計算資源がもっと密に協調するアプリもサポートできる処理系が必要



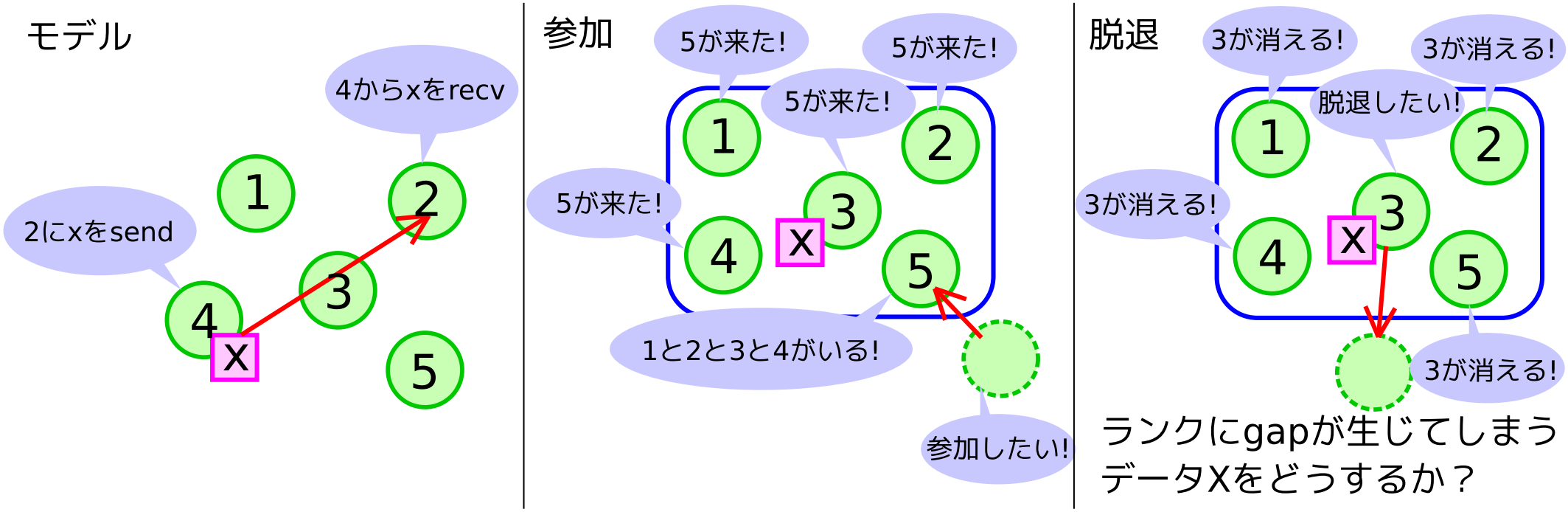
並列分散プログラミングモデル

➤ どのプログラミングモデルをベースにすべきか?



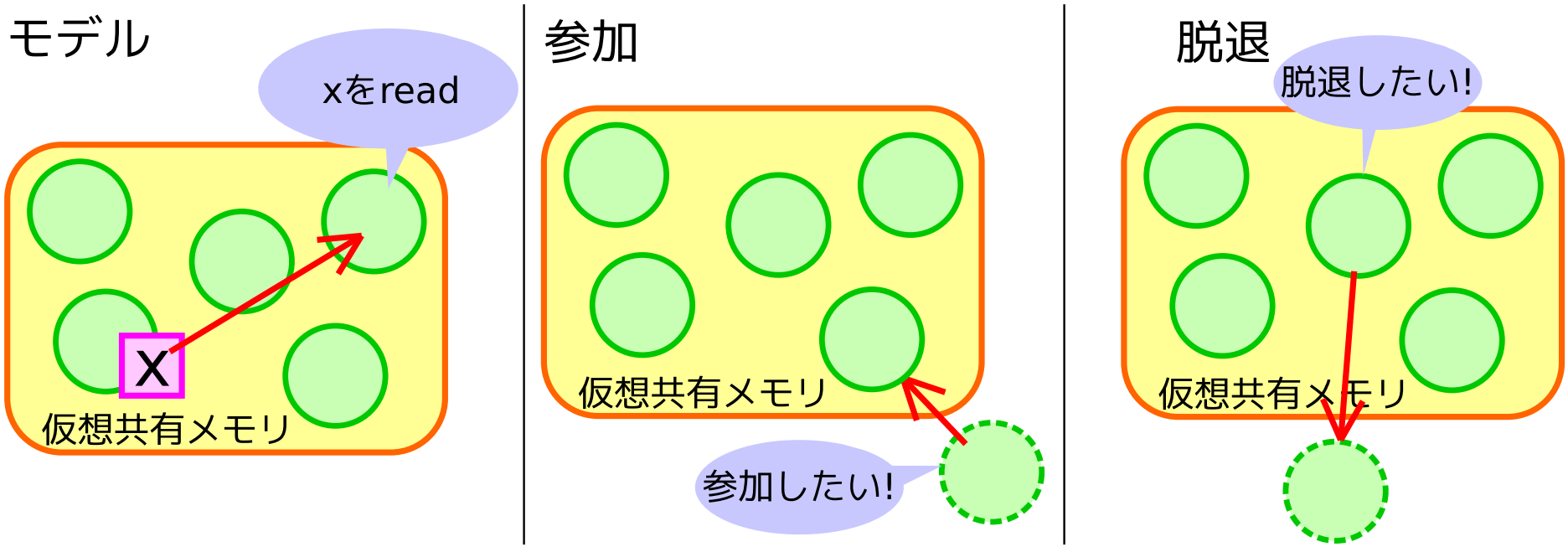
メッセージパッシング

- 一意的なランクを使用したデータの送受信 (send/recv) を明示的に記述
- データの所在管理はユーザプログラム側に任される
  - → ユーザプログラム側が「系内に誰がいて誰がどのデータを持っているのか」を把握する必要あり
  - → 参加/脱退時の記述は相当に複雑化



分散共有メモリ

- データの送受信 (send/recv) を隠蔽して，仮想的な共有メモリへのアクセス (read/write) に抽象化
- データの所在管理が処理系側で行われる
  - → ユーザプログラム側では「誰がどのデータを持っているか」はおろか「系内に誰がいるか」さえ把握する必要なし
  - → 参加/脱退に伴うユーザプログラムの記述が容易



本研究の提案

- DMI：Distributed Memory Interface
  - → **分散共有メモリをベースとして，多数の計算資源が密に協調するアプリ領域に対しても，計算資源の参加/脱退をサポートする並列分散ミドルウェア基盤**
  - → **独自のコンセプトに基づき，分散共有メモリとしての機能と性能を追求**

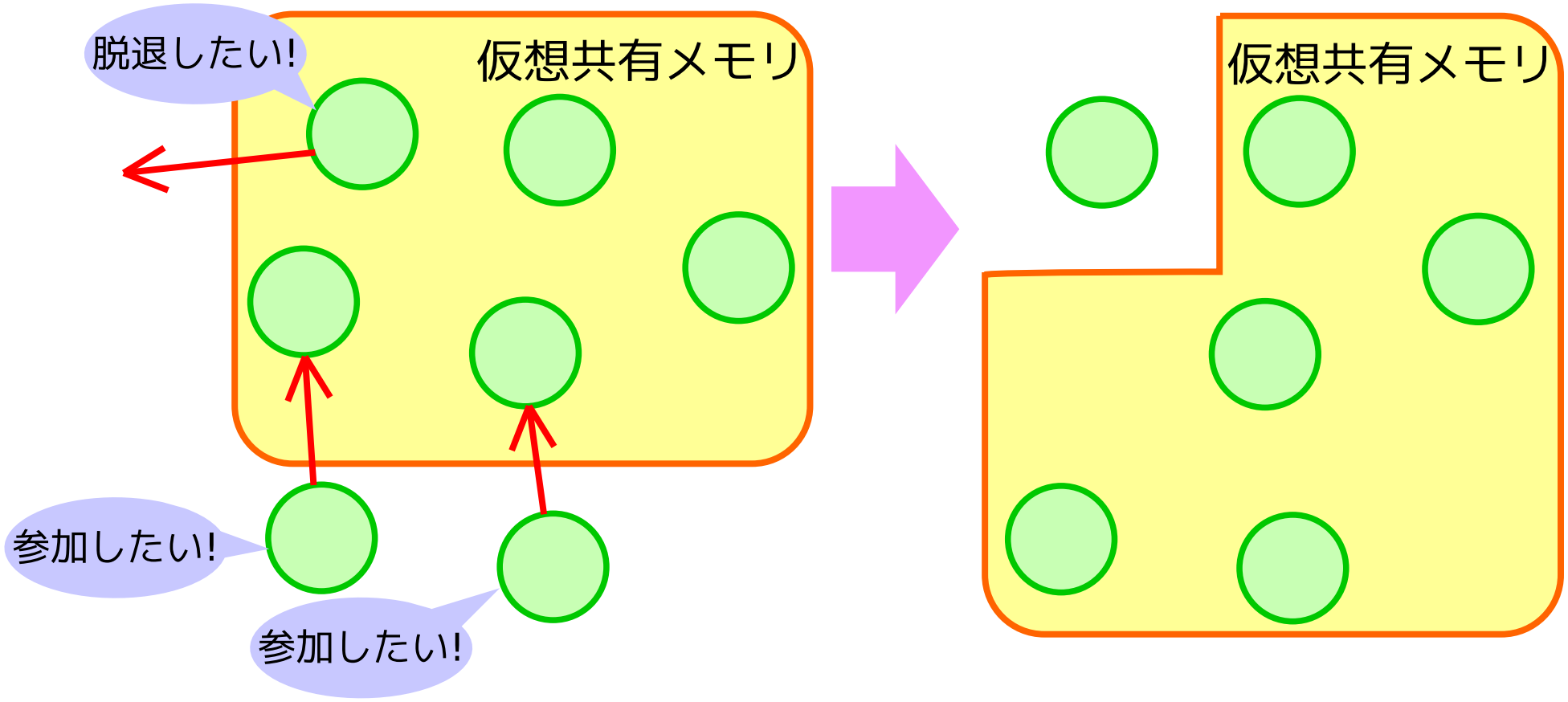
2. システムデザイン

4 大コンセプト

- 機能的要件
  - → 【1】動的な参加/脱退のサポート
  - → 【2】遠隔スワップシステム
  - → 【3】スレッドプログラミングとの対応性
- 性能的要件
  - → 【4】細粒度で明示的な最適化手段
    - ♦ 分散共有メモリにとって潜在的な性能の鈍さを補う

【1】動的な参加/脱退のサポート (1)

- 動的な参加/脱退に対応可能なコンシステンシプロトコルを定義
  - → 後述



【1】動的な参加/脱退のサポート (2)

- 従来の分散共有メモリ：SPMD 型のスタイル
  - → 「全員」の時系列的な挙動が明確な定型的処理に特化
  - → 「全員」の概念があると容易に参加/脱退を記述できない
- DMI：pthread 型のスタイル
  - → プログラム記述に際して「全員」の概念が不要
  - → 動的なスレッド生成/破棄を通じて，参加/脱退に伴う動的な並列度変化を容易に記述可能

【1】動的な参加/脱退のサポート (3)

- 便利な API：
  - → 参加中のノードを取得する API
  - → ノードの参加/脱退イベントをポーリングする API
  - → 参加ノードの総コア数が目標値になるまで待機する API
  - → …

【2】遠隔スワップシステム (1)

➤ **並列実行環境 ＋ 遠隔スワップシステム**

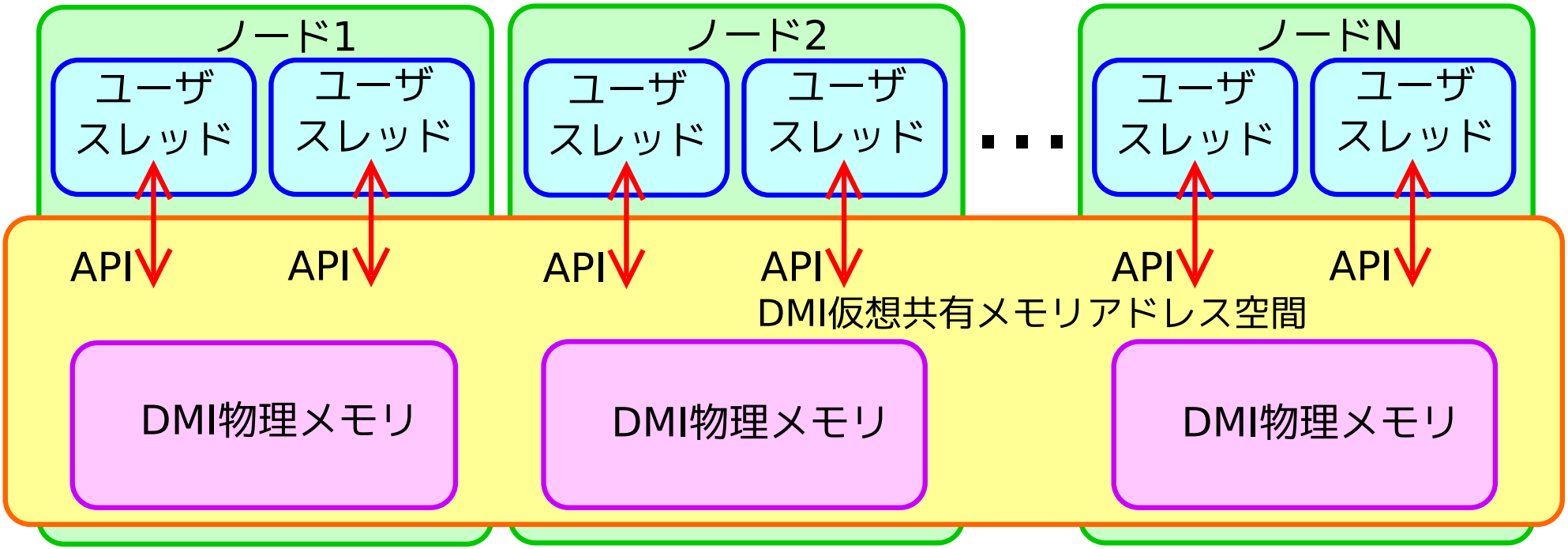
大規模分散共有メモリ

分散共有メモリ
(CPU数のスケーラビリティ)

並列実行環境の提供

遠隔スワップシステム
(メモリ量のスケーラビリティ)

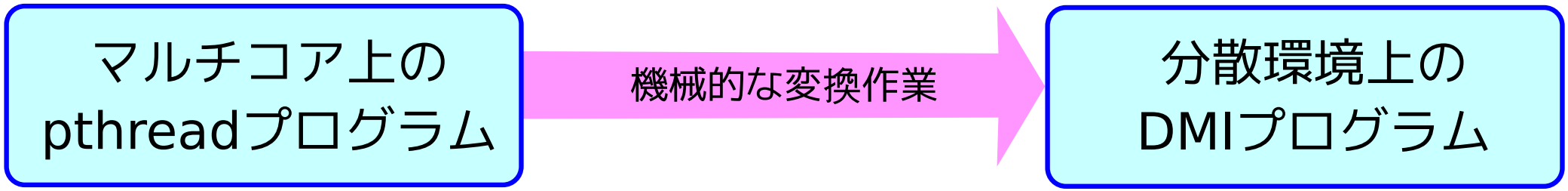
【2】遠隔スワップシステム (2)



- 各ノードの DMI 物理メモリを集めて DMI 仮想共有メモリを構築
- 各ユーザスレッドは全ノードの DMI 物理メモリに透過的にアクセス可能
- 複数ユーザスレッドが DMI 物理メモリを「共有キャッシュ」的に利用
  → マルチコアレベルの並列性も有効活用
- ページ置換

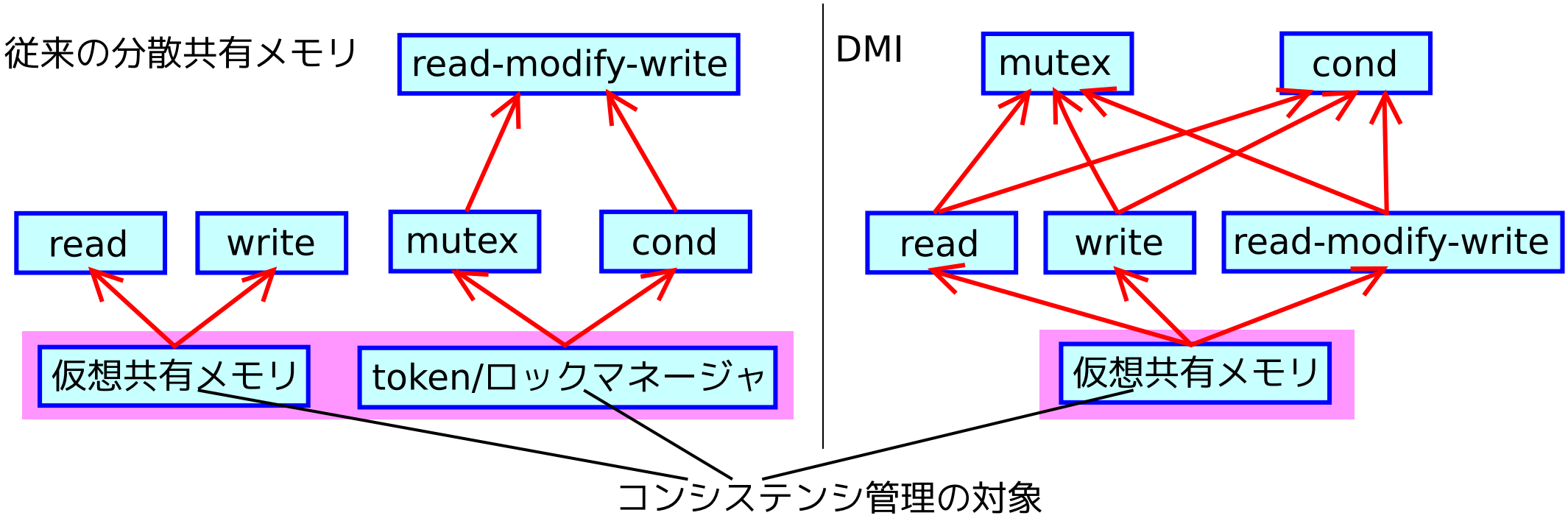
【3】スレッドプログラミングとの対応性 (1)

- pthread **プログラムに対して**ほぼ機械的な思考に基づく変換作業**だけで** DMI **のプログラムが得られるような** API
  - → create, join, detach
  - → mutex
  - → cond



【3】スレッドプログラミングとの対応性 (2)

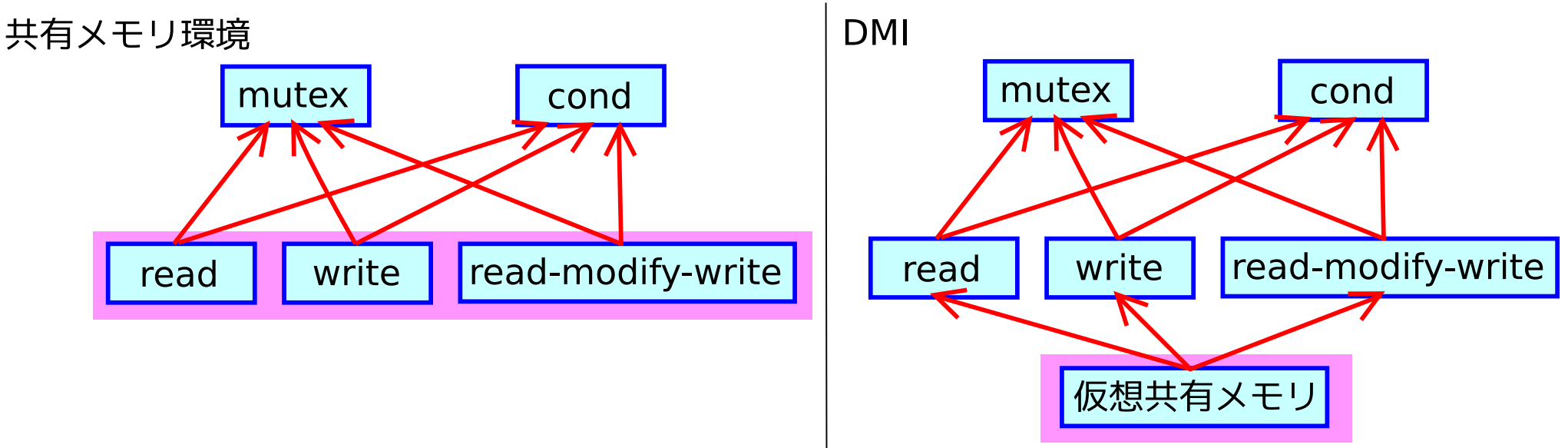
- **従来の分散共有メモリ：**token[Naimi et al,1996] **やロックマネージャを利用して** mutex **や** cond **を実装**
- DMI：read/write/fetch-and-store/compare-and-swap **を基盤として** mutex **や** cond **を実装**



- **実装上の利点：コンシステンシ管理の対象が仮想共有メモリだけで済む**

【3】スレッドプログラミングとの対応性 (3)

- 機能上の利点：共有メモリ環境の同期の階層関係を忠実に反映
  - → 「mutex より read-modify-write の方が軽い」
  - → 共有メモリ環境上の効率的なアルゴリズムをサポート
    - ♦ 例：wait-free なデータ構造



【4】細粒度で明示的な最適化手段 (1)

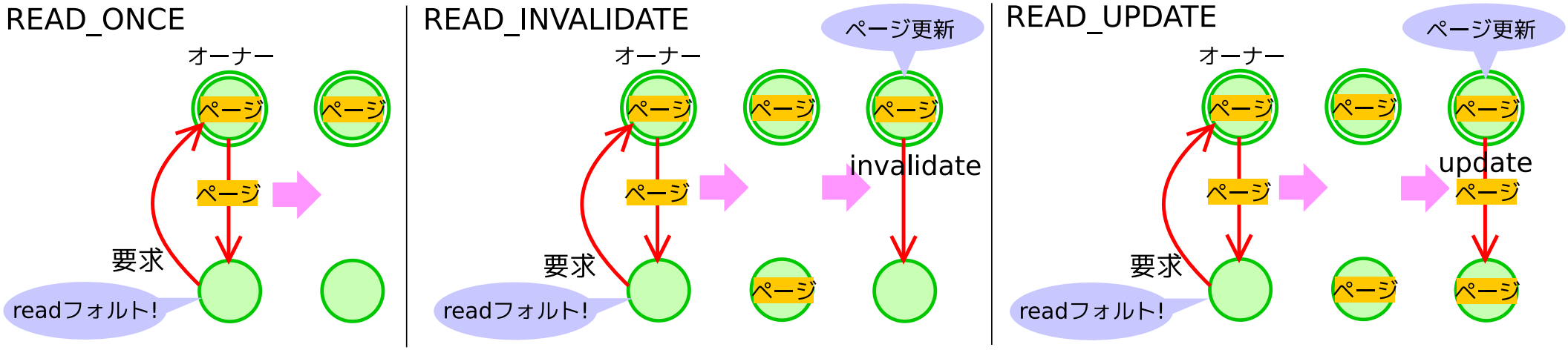
- (OS のメモリ保護機構に頼ることなく) ユーザレベルでコンシステンシ管理
  - → 「ページ」単位で Sequential Consistensy を保証
- アプリの挙動に合致した任意のページサイズでメモリ確保
  - → OS の 4KB 単位のメモリ保護機構を利用するよりもページフォルト回数を大幅に削減
  - → 例：行列丸ごと 1 個を 1 ページに指定

【4】細粒度で明示的な最適化手段 (2)

- **関数呼び出し型の** read/write
  - → DMI_read(...)，DMI_write(...)
  - → **「どう** read/write **したいのか」を関数の引数として指定可能**
    - ♦ **マルチモード** read/write **：データの物理的な所在に関する最適化**
    - ♦ **非同期** read/write **：通信時間隠蔽に関する最適化**

【4】細粒度で明示的な最適化手段 (3)

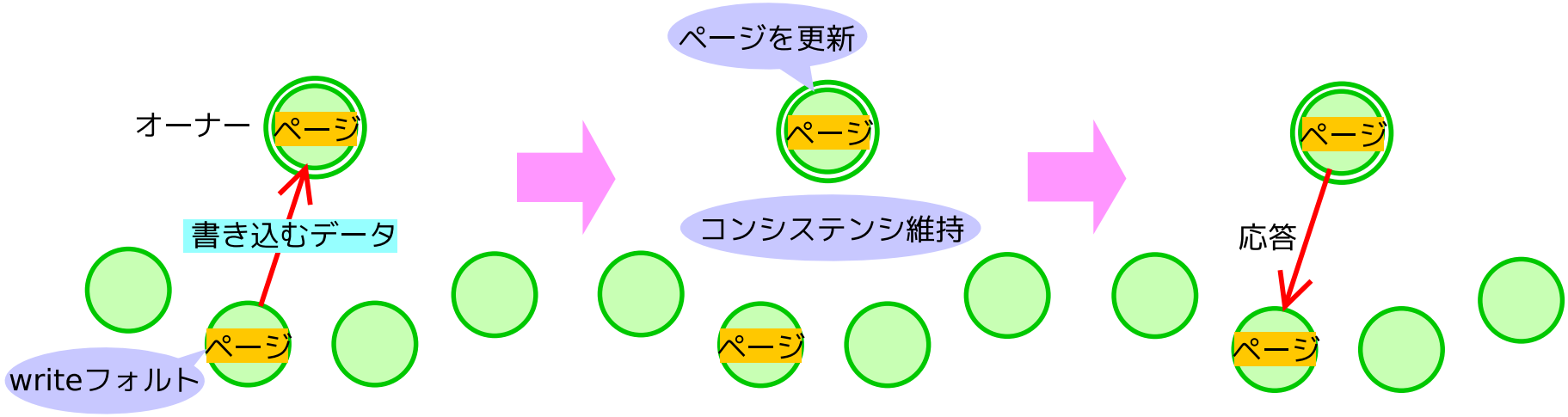
- **マルチモード** read：
  - → READ_ONCE：**今の 1 回だけ読めればいい**
  - → READ_INVALIDATE：**今読んだものはキャッシュしておきたいが，更新時にはキャッシュが無効化されてもいい**
  - → READ_UPDATE：**今読んだものをキャッシュするとともに，キャッシュをずっと最新に保ちたい**
- update **型と** invalidate **型をどうハイブリッドさせるか**を read **の粒度で**明示的に指定可能



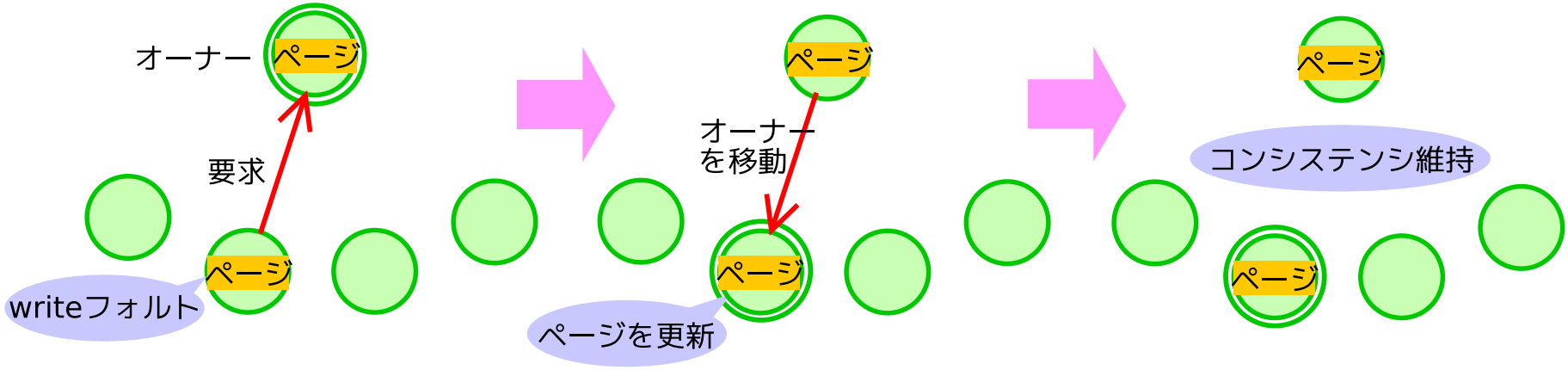
【4】細粒度で明示的な最適化手段 (4)

➤ **マルチモード** write：

➔ WRITE_REMOTE：**データの書き込みをオーナーに行わせる**



➔ WRITE_LOCAL：**オーナー権を奪った後で自分でデータを書き込む**



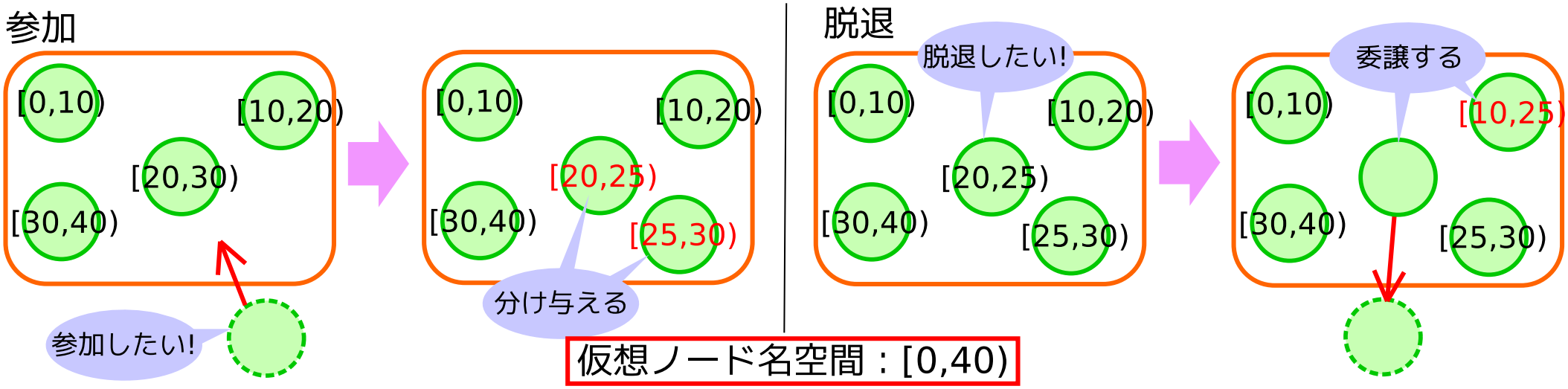
【4】細粒度で明示的な最適化手段 (5)

- read/write の**非同期**版
  - → **計算と通信のオーバーラップ**
  - → **プリフェッチ**

3. 関連研究

関連研究 (1)

- Phoenix[Taura at el,2003]：
  - → メッセージパッシングベースで動的な参加/脱退に対応
  - → ユーザプログラム側では物理的なノード名とは別の「仮想ノード名」を用いて通信を記述
- プログラム記述は複雑



関連研究 (2)

- DSM-Threads[Muller,1997]：
  - → 分散共有メモリベースで pthread を分散拡張
    - ♦ pthread との対応性を重視
  - → データの表現形式やアラインメントに関してヘテロな環境に対応
  - → token を用いた効率的な優先度付き排他制御
- 動的な参加/脱退には未対応

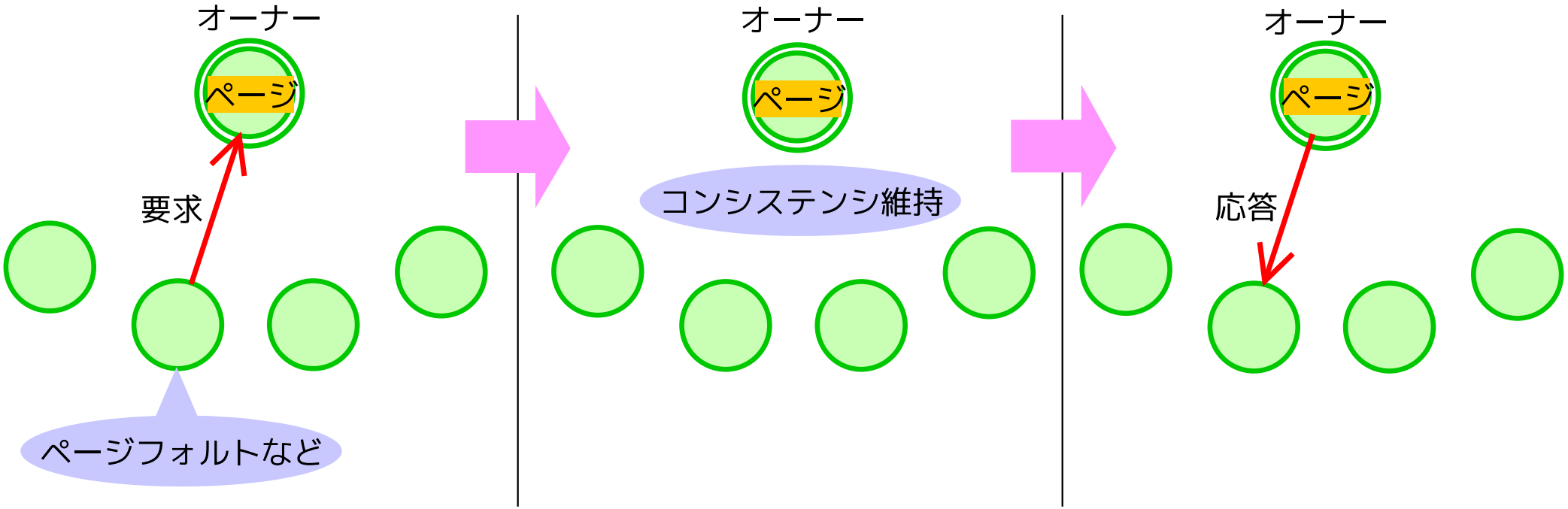
関連研究 (3)

- Teramem[Yamamoto et al,2009]：
  - → 逐次処理のための遠隔スワップシステム
  - → カーネルモジュールとして実装
  - → MMU の情報を利用した疑似 LRU
  - → 複数ページをまとめた遠隔スワップによるバンド幅を有効活用
  - → Myrinet 10G 環境で，GNU sort が HDD アクセスより 40 倍以上高速
- 並列実行環境は提供されず，動的な参加/脱退にも未対応

4. 参加/脱退対応のプロトコル

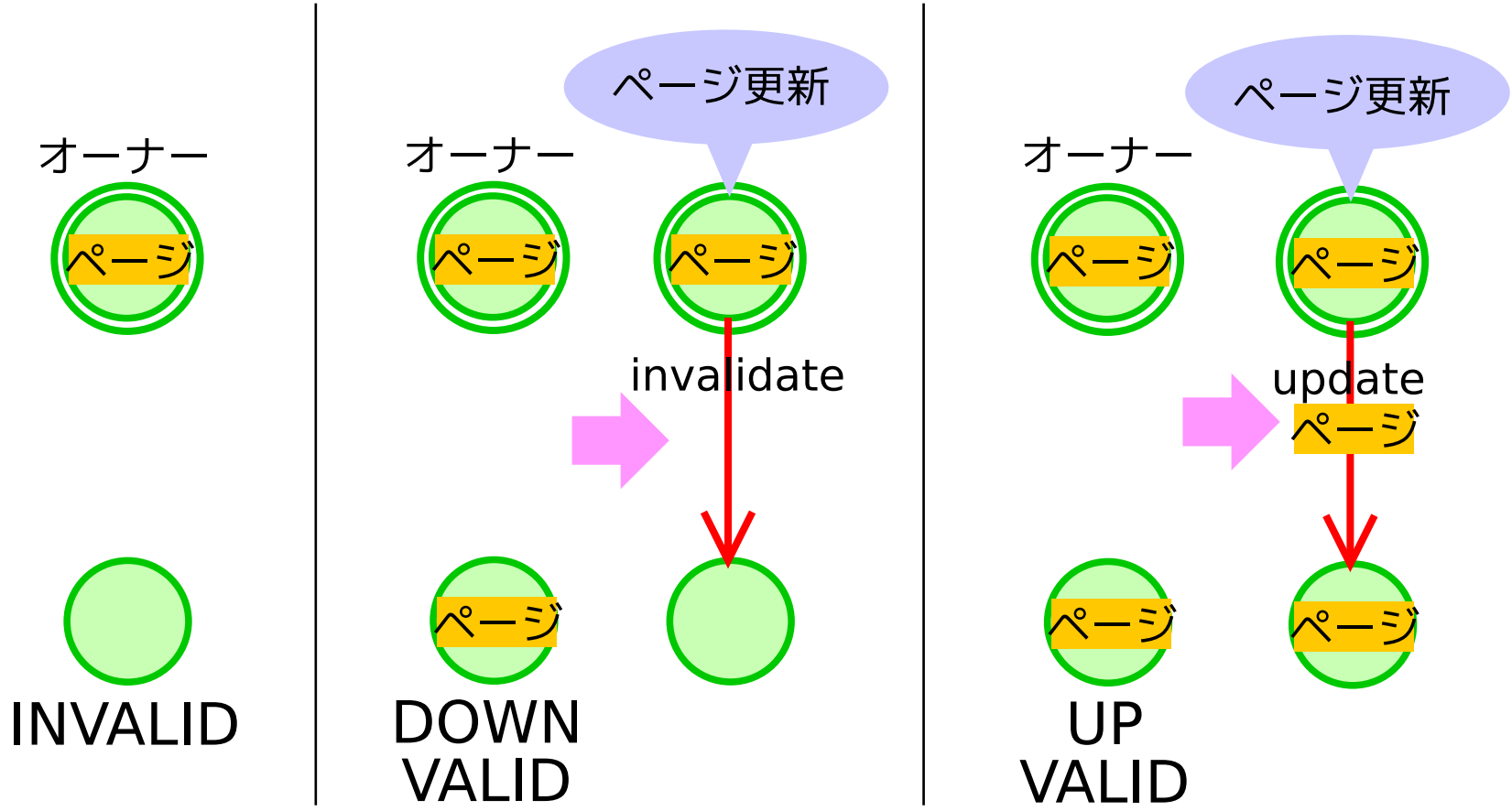
設計すべきプロトコルの概観 (1)

- メタ情報を管理するオーナーが各ページごとに 1 個存在
- 動的環境下では固定的なノードを設置できないため**オーナーは動的に変化**
- 基本的な挙動：要求メッセージをオーナーに通知し，Sequential Consistency 維持を行い，応答メッセージを返す



設計すべきプロトコルの概観 (2)

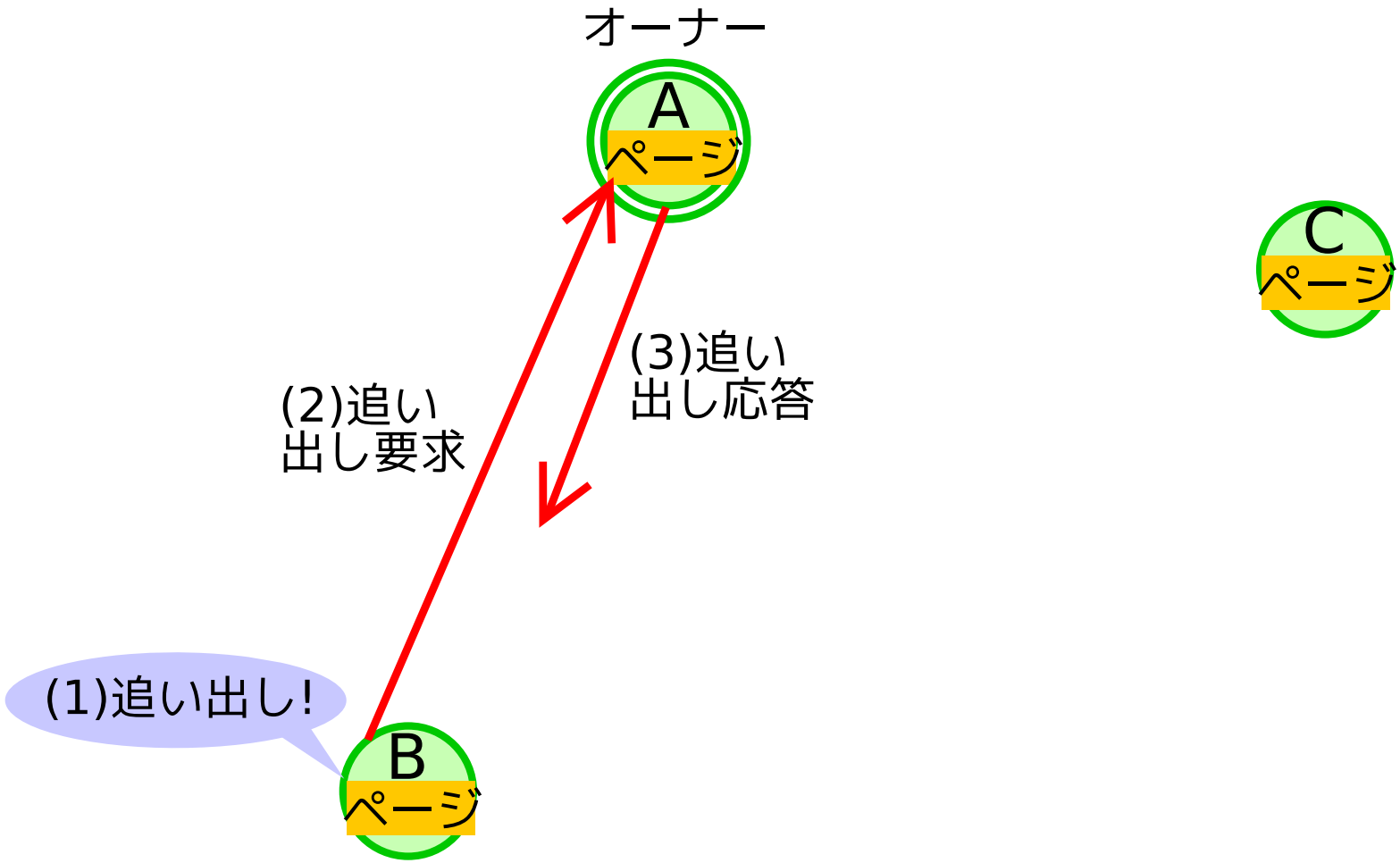
- (マルチモード read に対応するため) ページが取りうる 3 状態：
  - INVALID：無効
  - DOWN_VALID：有効だが，次回の更新時に無効化される
  - UP_VALID：有効で，今後も最新状態に保たれる



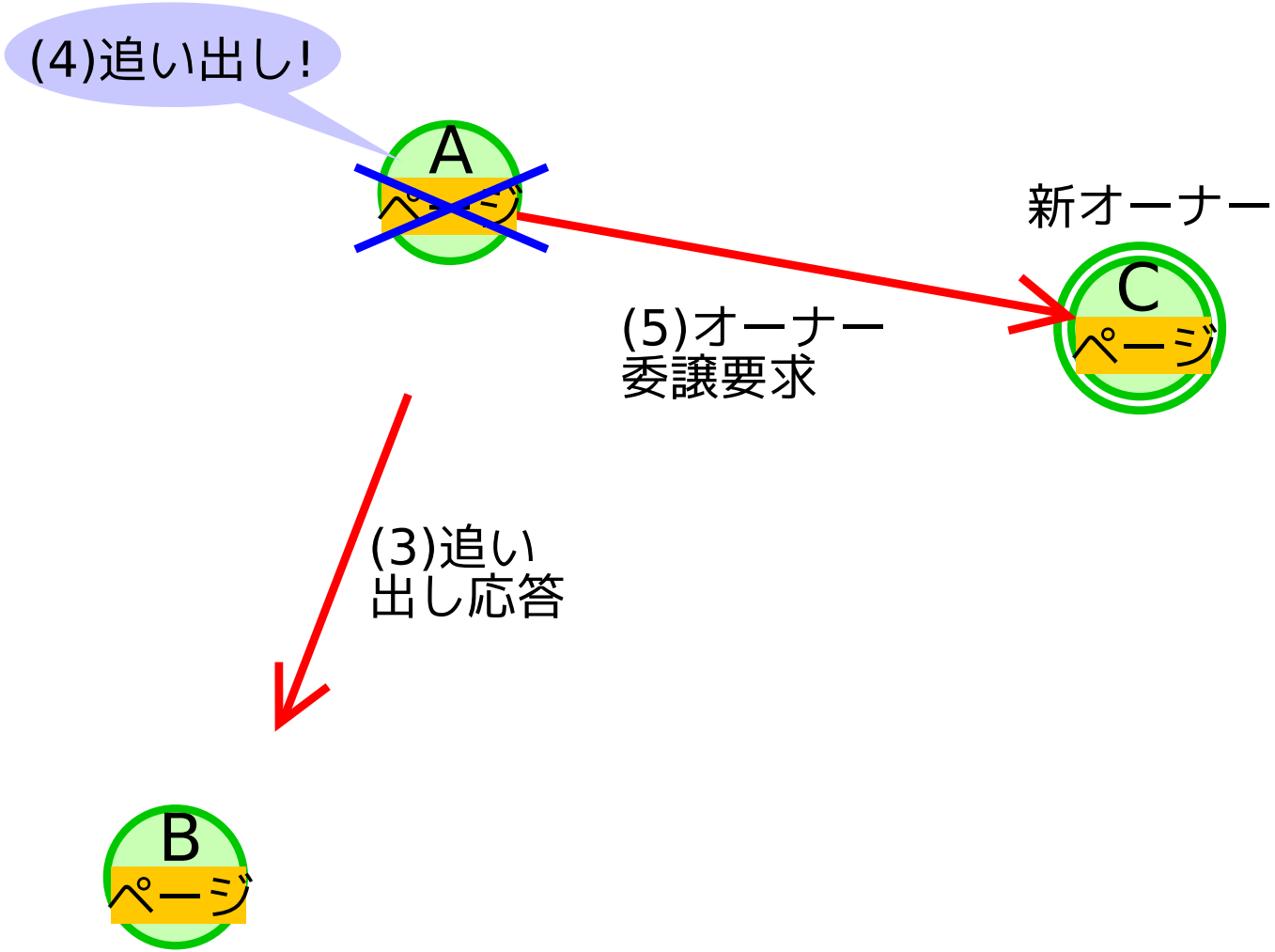
設計すべきプロトコルの概観 (3)

- 定義すべきプロトコル：
  - → read フォルト
    - READ_ONCE
    - READ_INVALIDATE
    - READ_UPDATE
  - → write フォルト
    - WRITE_LOCAL
    - WRITE_REMOTE
  - → (ページ置換と脱退にとって必要な) ページの追い出し
    - 自分がオーナーである場合
    - 自分がオーナーでない場合
- 多様で複雑！

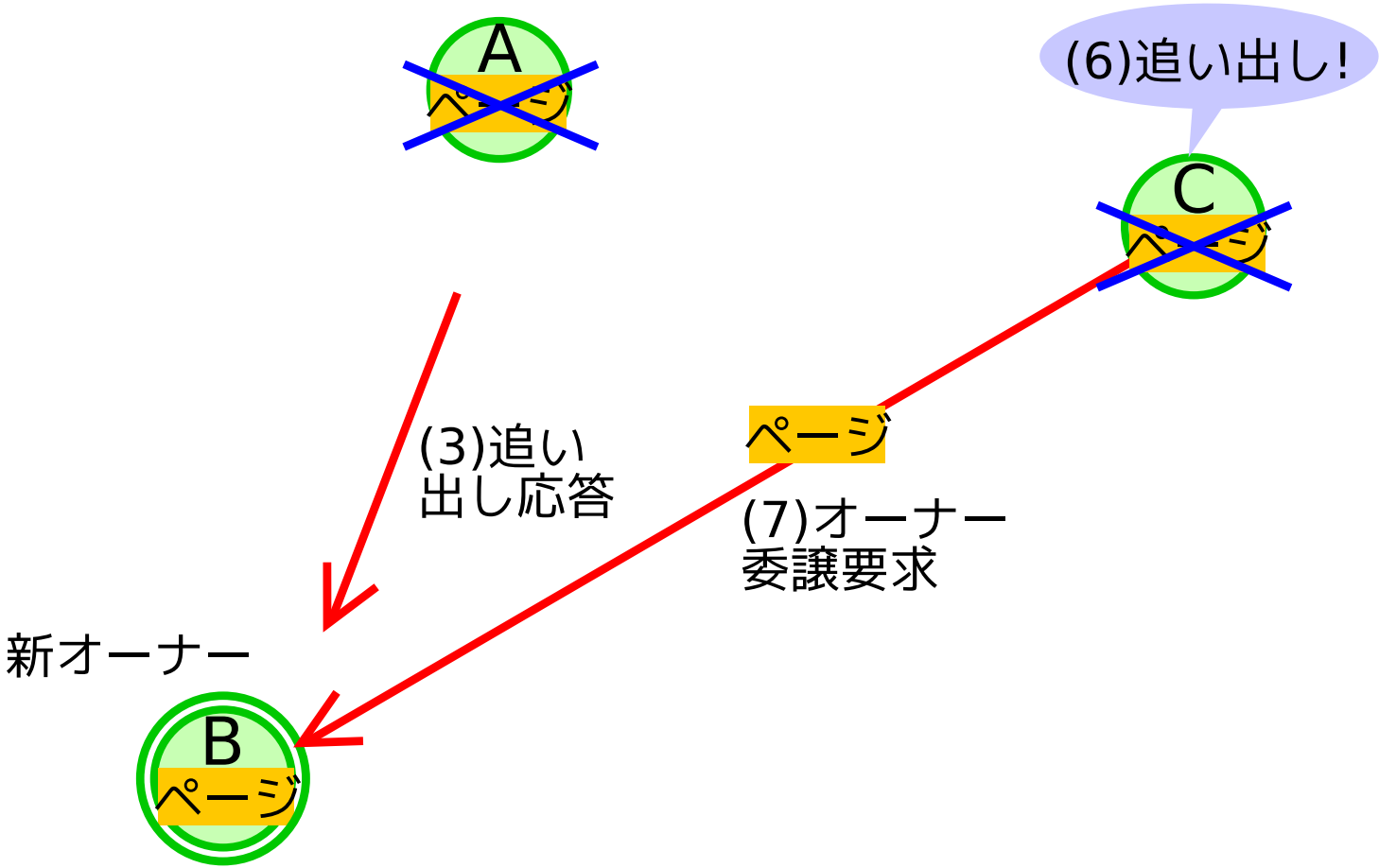
失敗例：プロトコル設計の難しさの確認 (1)



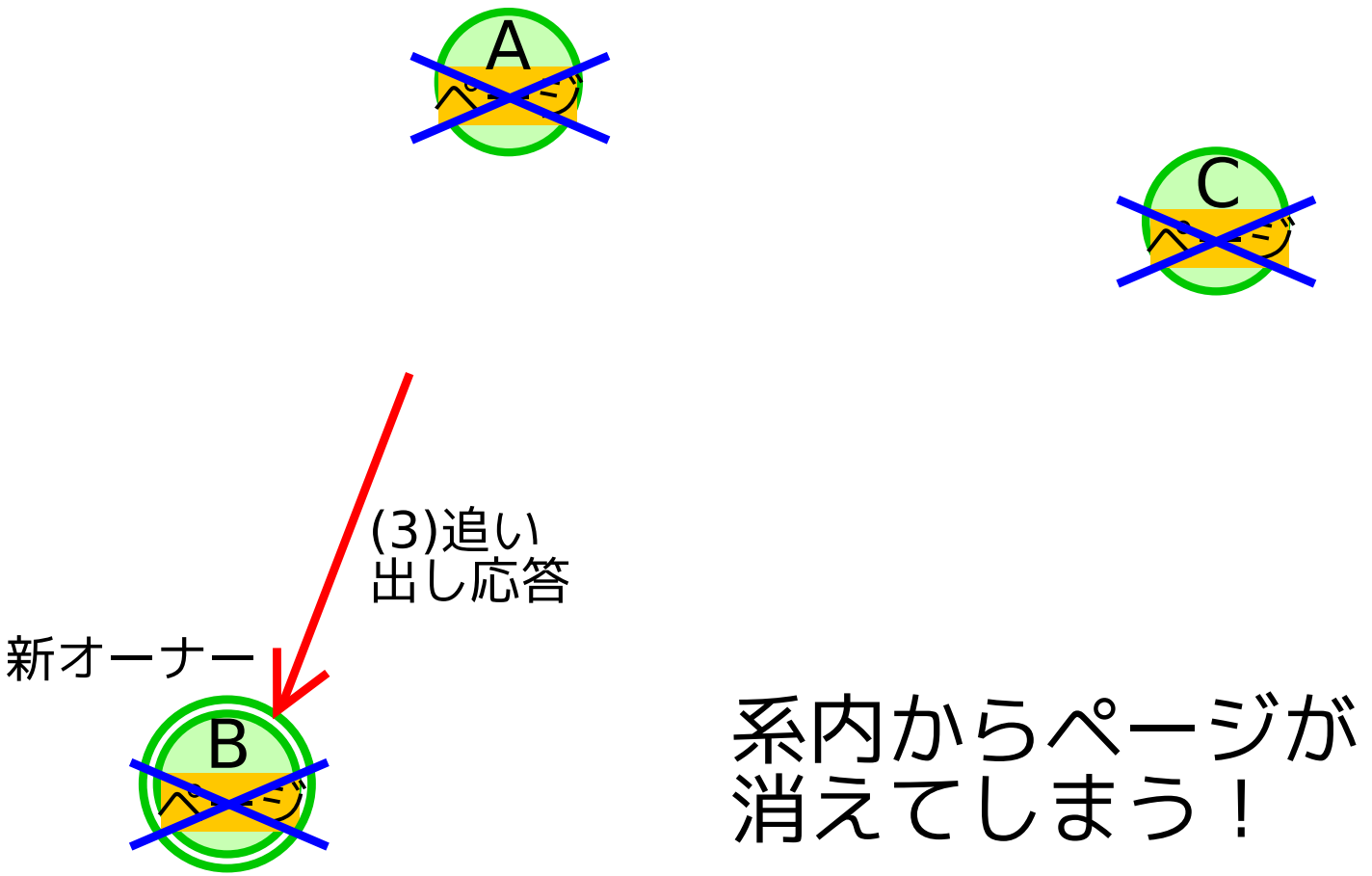
失敗例：プロトコル設計の難しさの確認 (2)



失敗例：プロトコル設計の難しさの確認 (3)



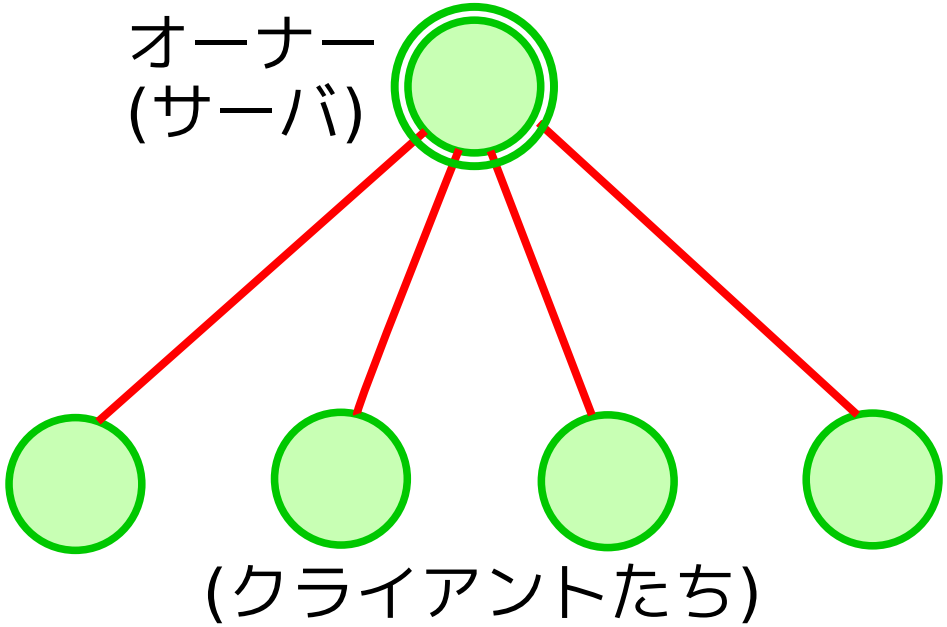
失敗例：プロトコル設計の難しさの確認 (4)



- **オーナーが動的に変化する状況でのプロトコル設計は難解！**
- **どうすれば複雑なプロトコルを設計できるか?**

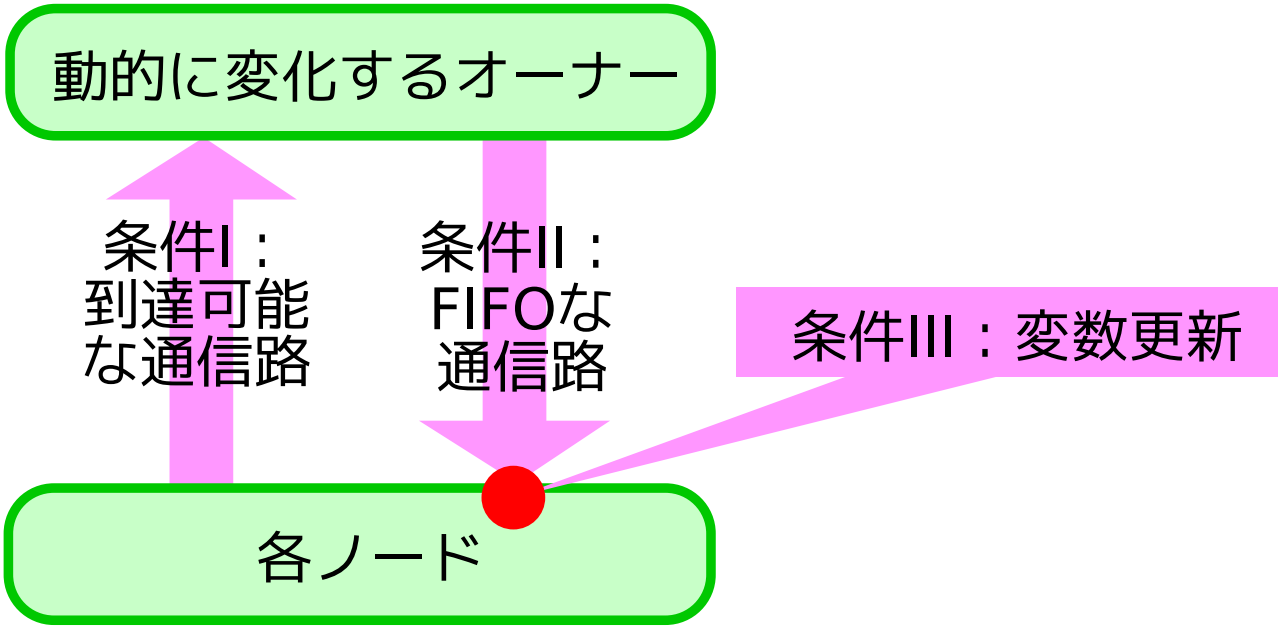
観察：オーナーが固定されている場合

- クライアント・サーバ方式
- どんなに複雑なプロトコル設計も自明
  - → 理由：各ノードの状態変化を**オーナーの意図通り**に行えるから
    - (1) 各ノードからオーナーへの通信路が存在
    - (2) オーナーから各ノードへの FIFO な通信路が存在
    - (3) オーナーからのメッセージを受信した時点でしか各ノードの状態変化が引き起こされない



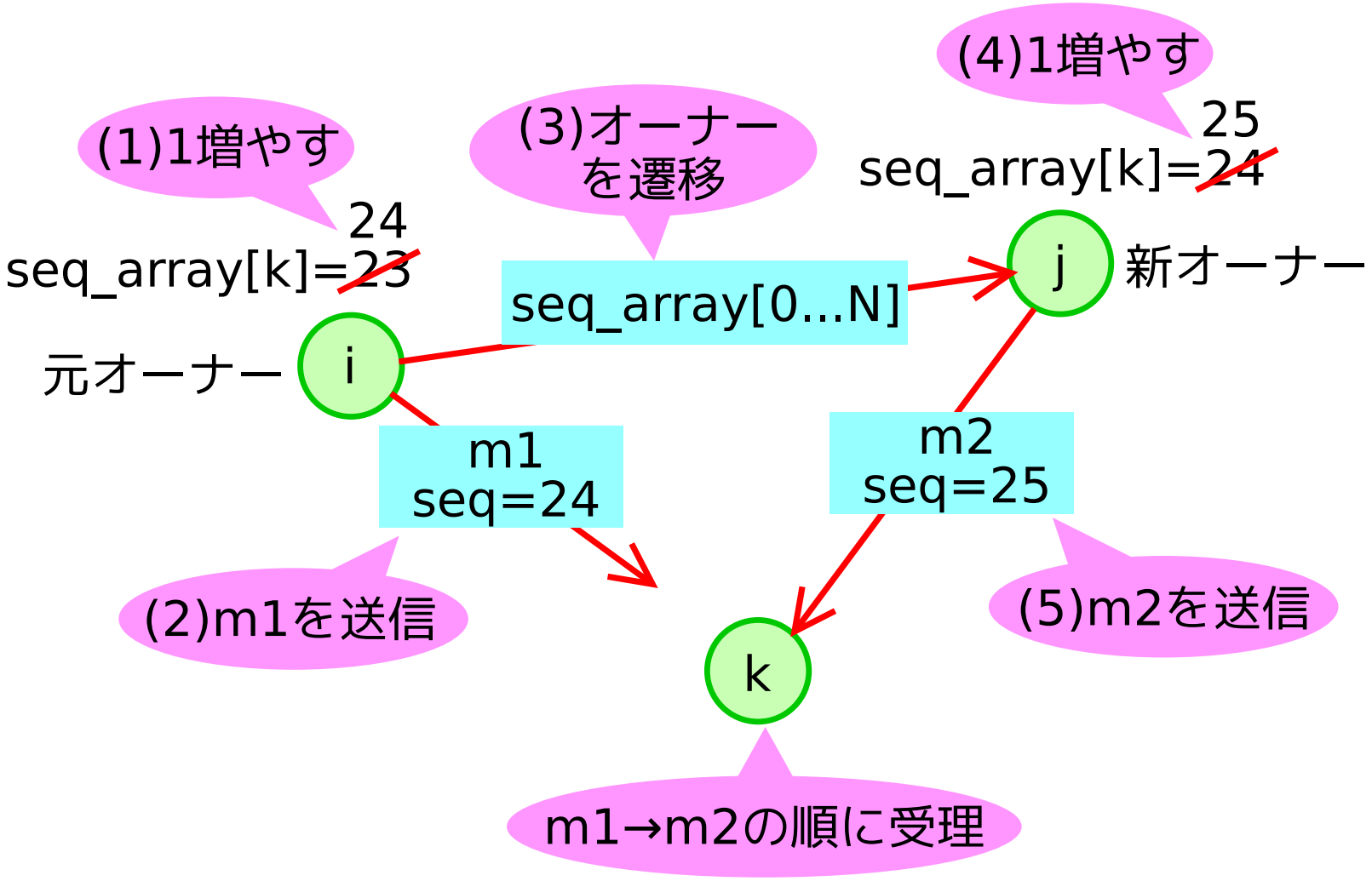
オーナーが変化する場合のプロトコル設計規約

- ということは，オーナーが動的に変化したとしても，以下の 3 条件さえ保証すれば，あたかもオーナー固定であるかのように複雑なプロトコルを容易に設計可能
  - → 条件 Ⅰ：各ノードからオーナーへの通信路が存在
  - → 条件 Ⅱ：オーナーの遷移に関係なくオーナーから各ノードへの FIFO な通信路が存在
  - → 条件 Ⅲ：各ノードの変数はオーナーからのメッセージを受信した時点でしか更新されない



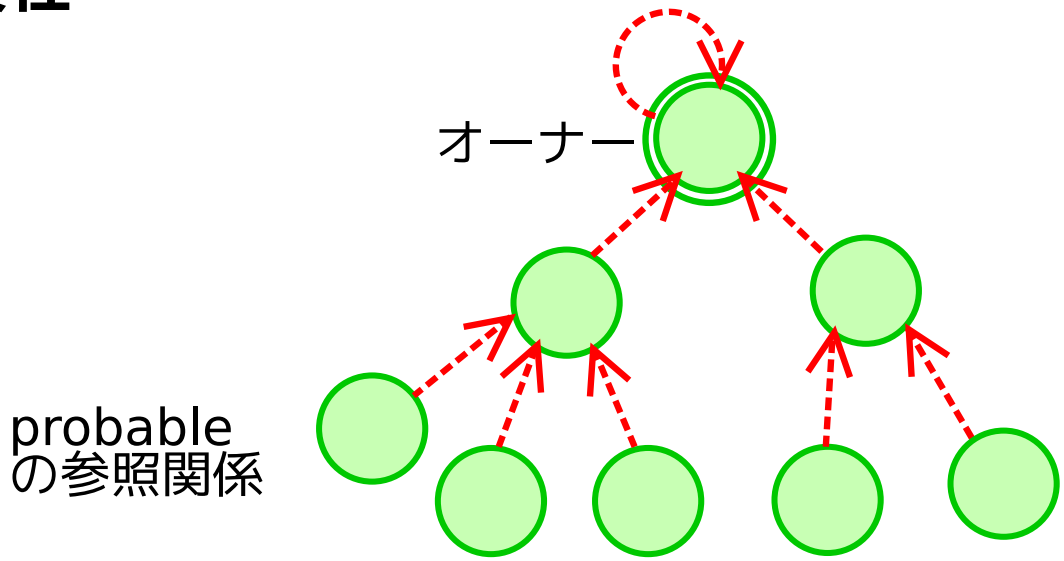
条件 II の実現法：メッセージの順序制御

- オーナーから各ノードへのメッセージに順序番号を付与し，**各ノードでメッセージを順序制御**



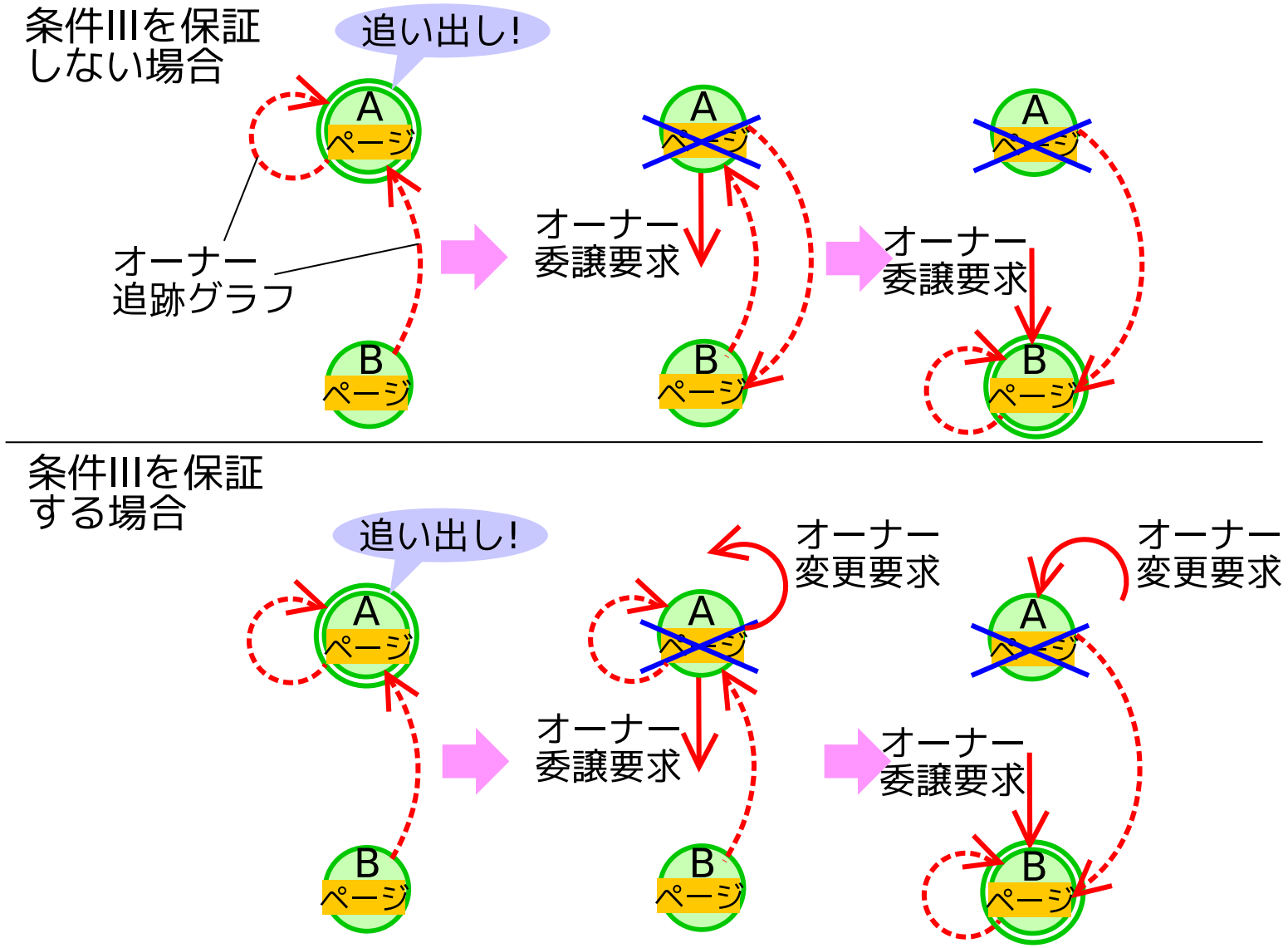
条件 I の実現法：オーナー追跡グラフ

- **オーナー追跡グラフ** [Li et al,1989]
  - → 各ノードが変数 probable を保持
    - ♦ probable= 「たぶんこのノードがオーナー」
  - → 全ノードを通じた probable の参照関係がオーナーに収束するグラフになるように管理
  - → オーナー宛の要求メッセージは各ノードでフォワーディングすることで，やがてオーナーに到達可能
- 各ノードの probable をどう管理すればよいか?
  - → 条件 III の必要性

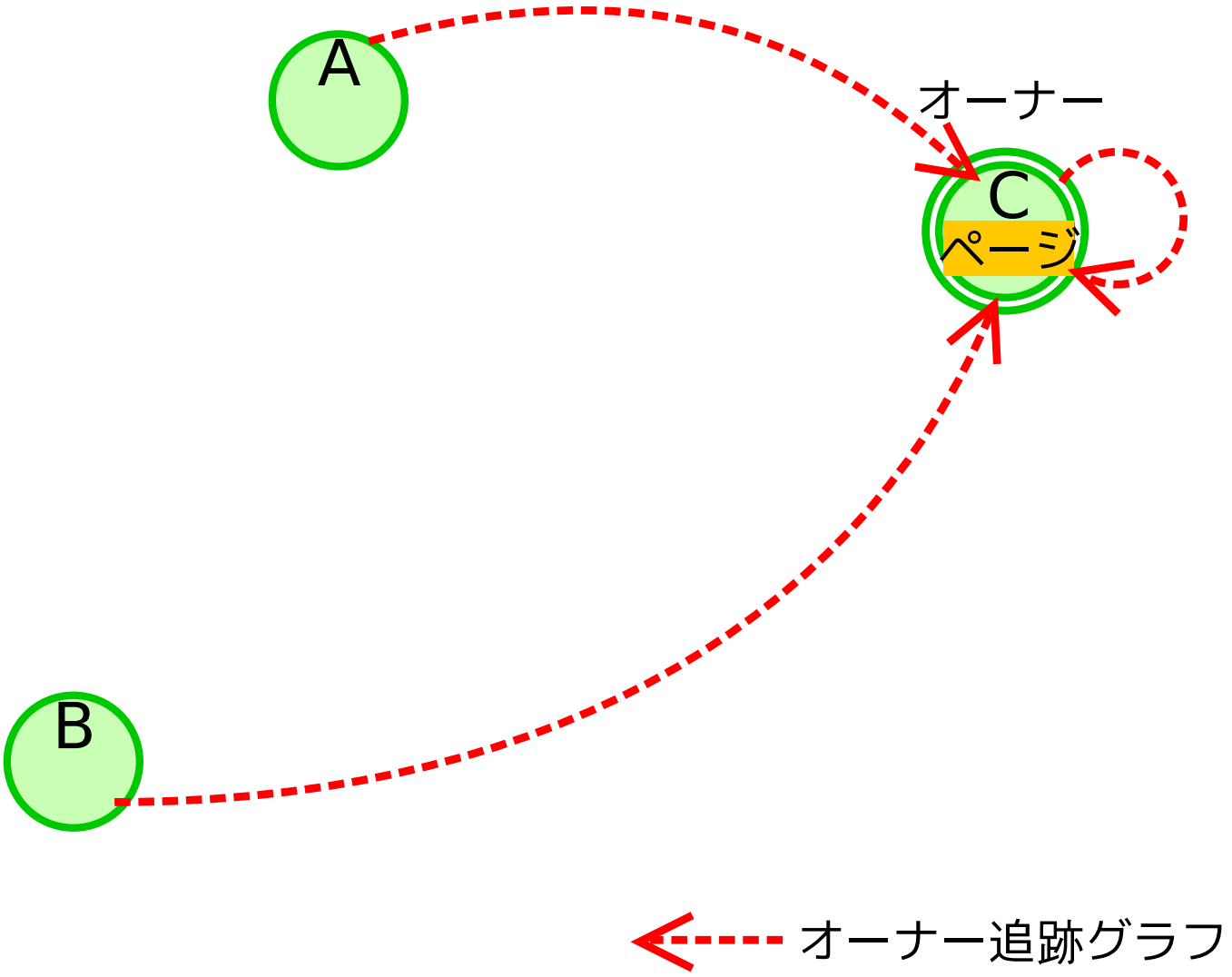


条件 III の必要性

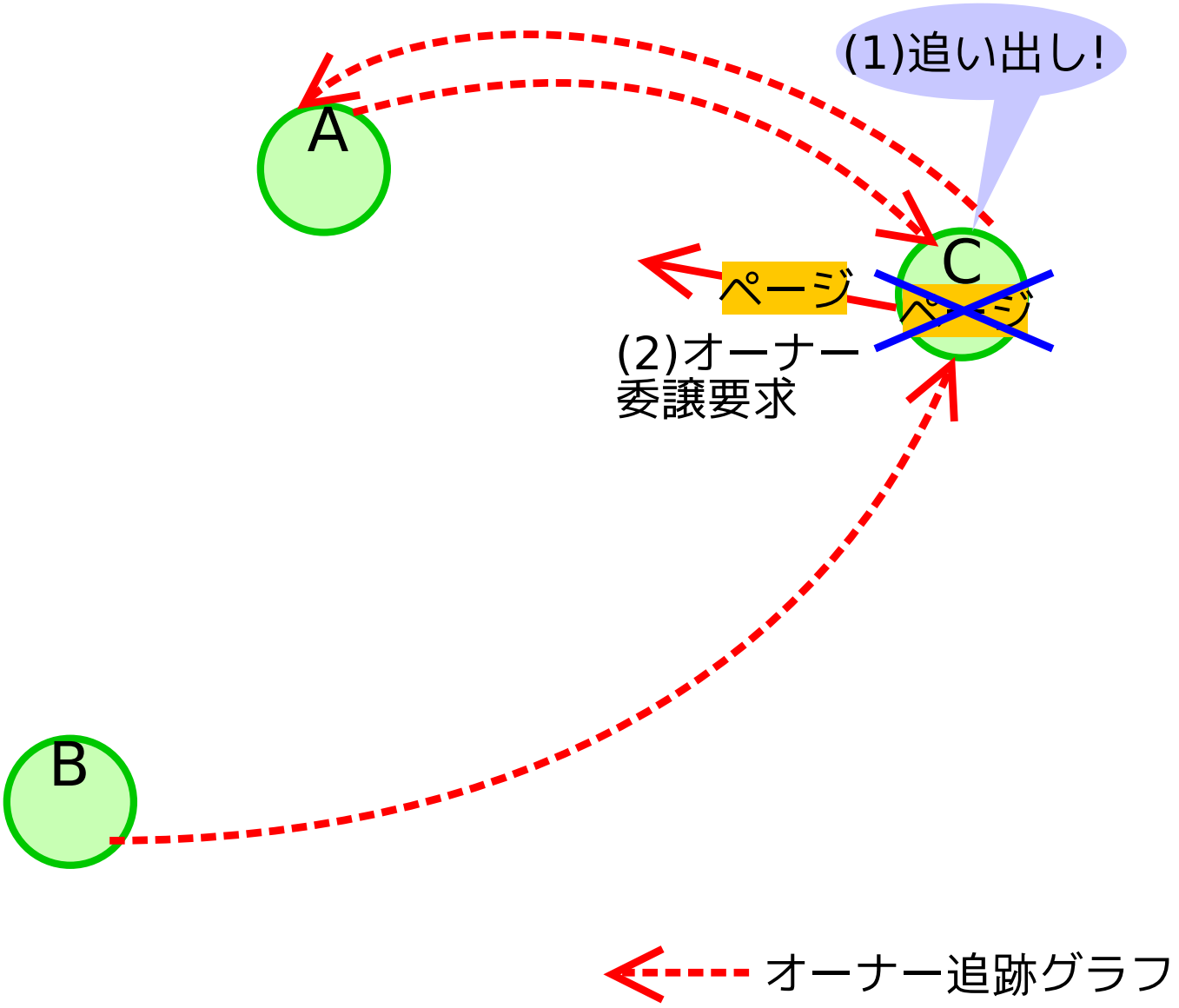
- **条件 III：(probable などの) 各ノード上の変数は，オーナーからの順序制御されたメッセージを受信した時点でしか更新しない**



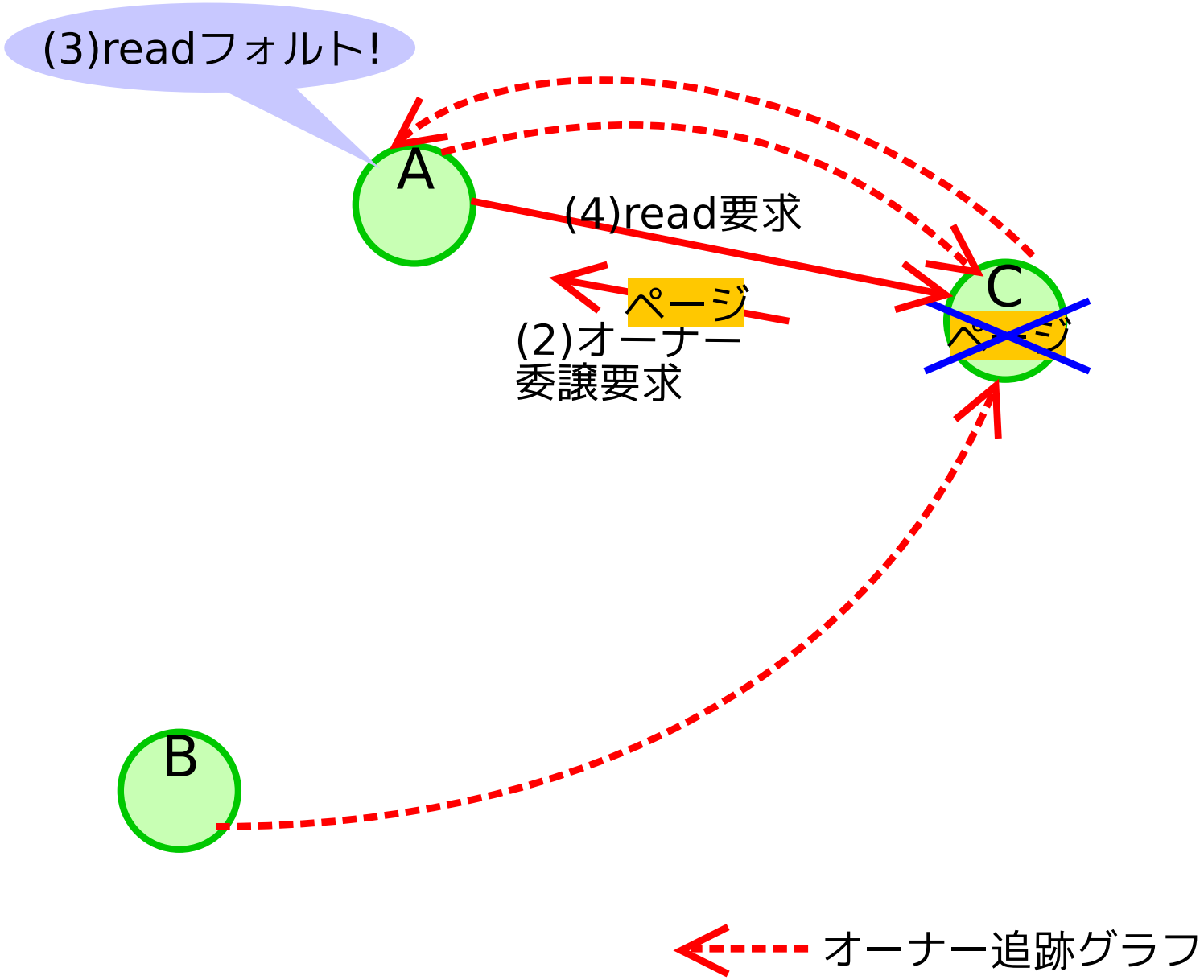
失敗例：条件 Ⅲ の必要性の確認 (1)



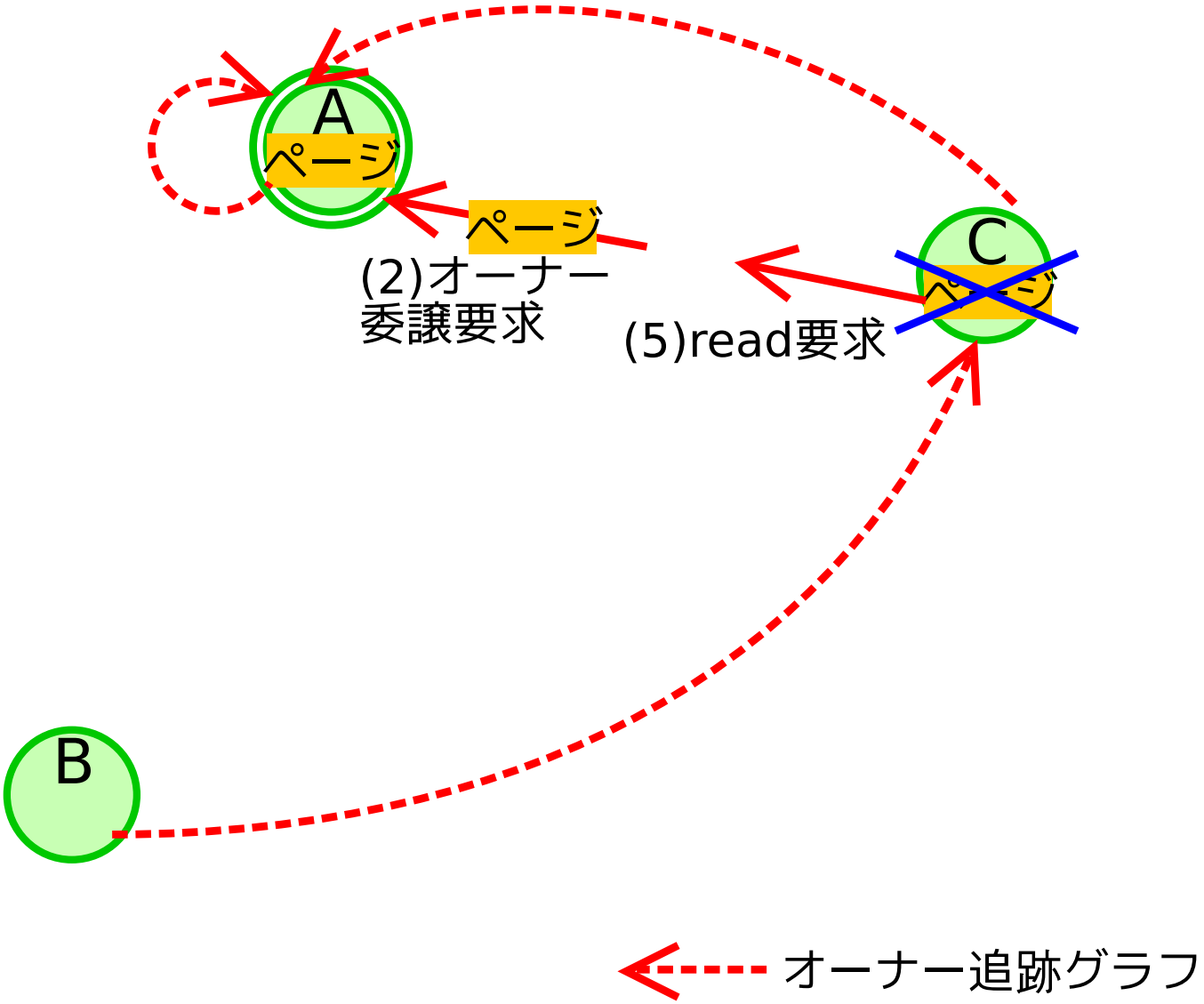
失敗例：条件 III の必要性の確認 (2)



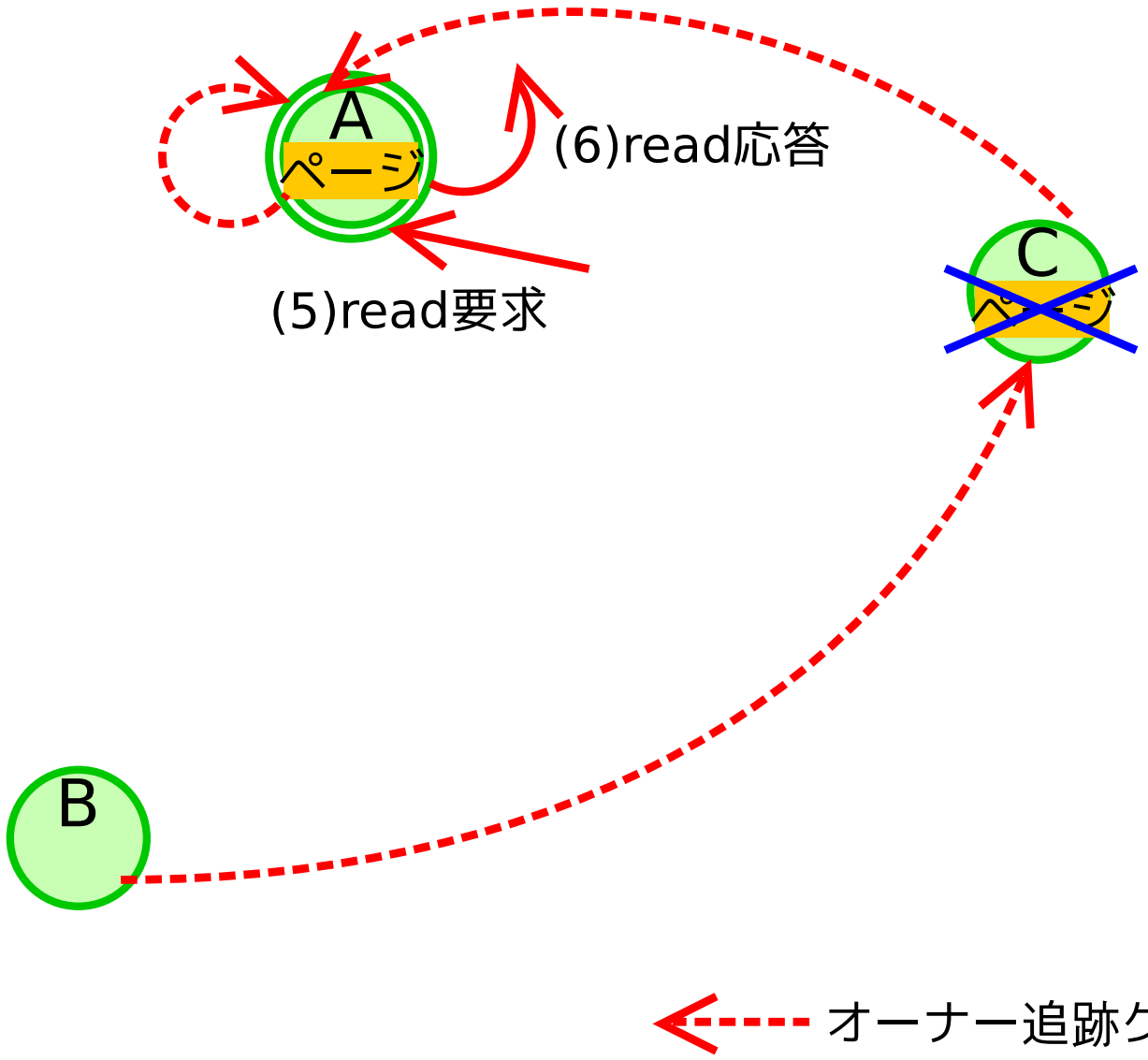
失敗例：条件 Ⅲ の必要性の確認 (3)



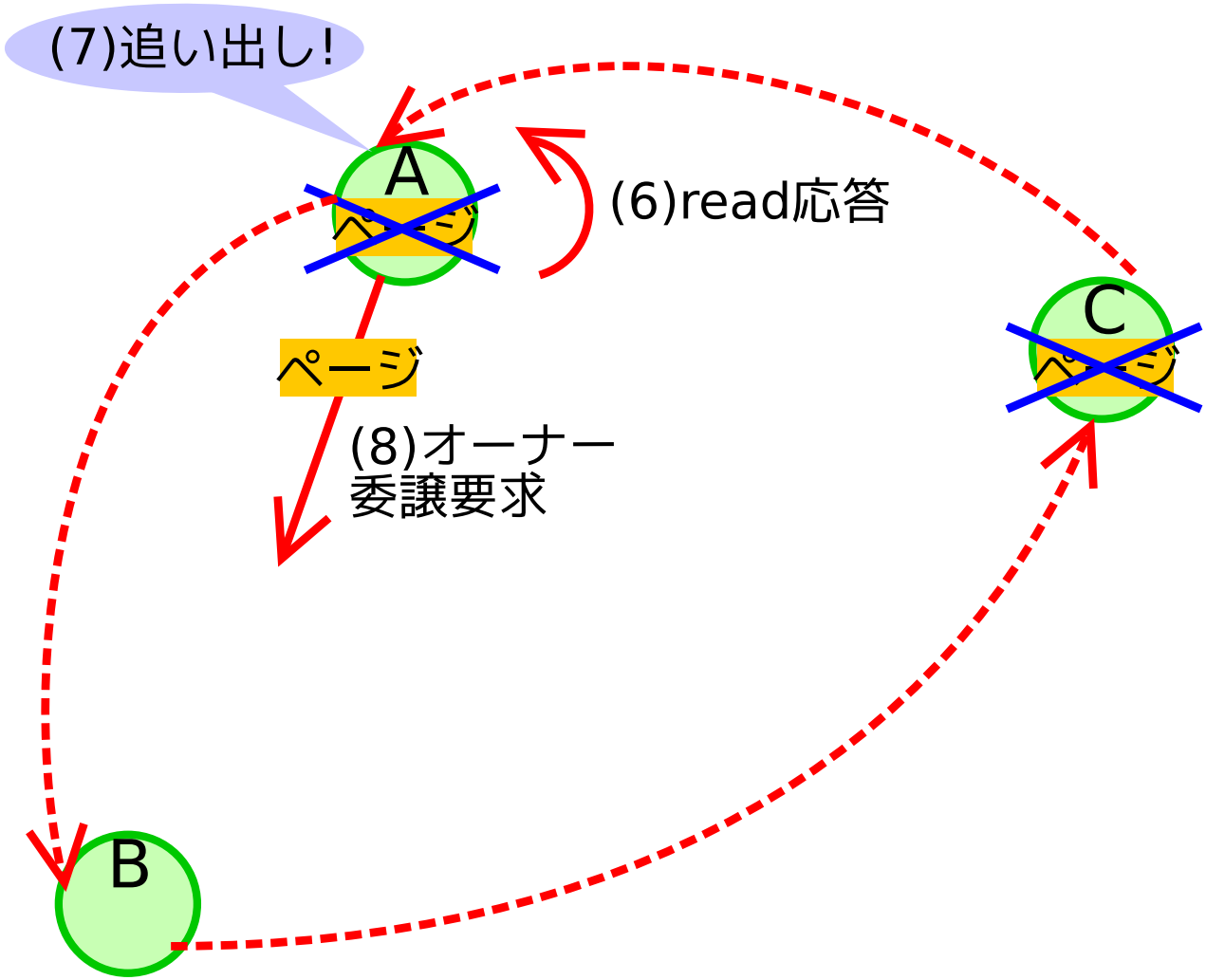
失敗例：条件 Ⅲ の必要性の確認 (4)



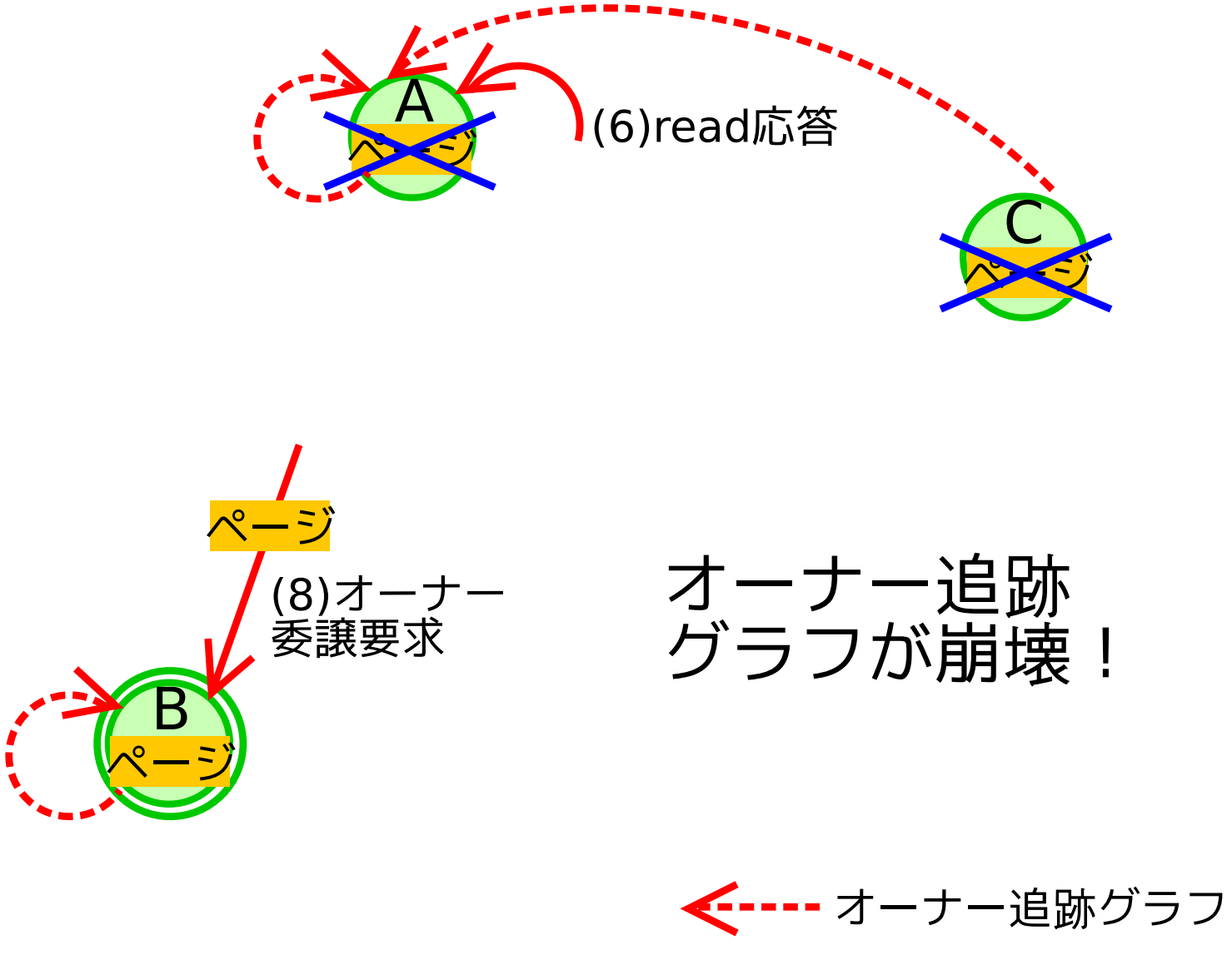
失敗例：条件 Ⅲ の必要性の確認 (5)



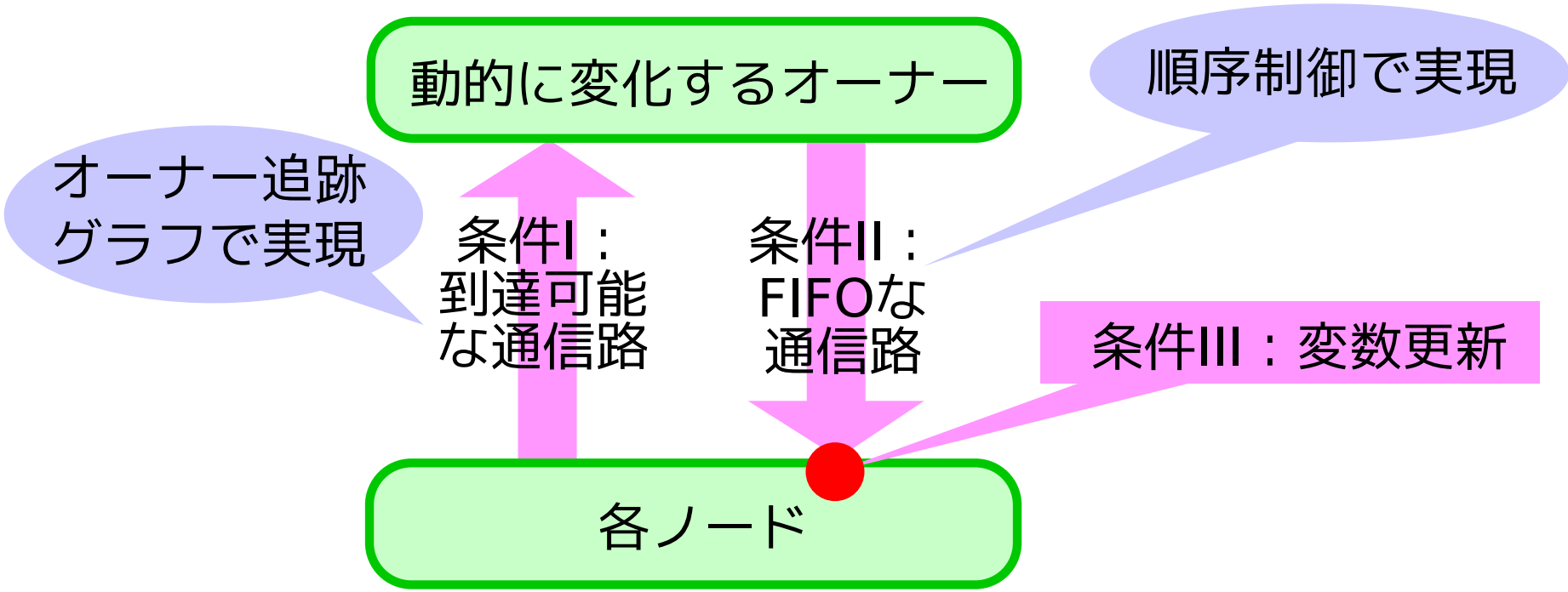
失敗例：条件 III の必要性の確認 (6)



失敗例：条件 III の必要性の確認 (7)

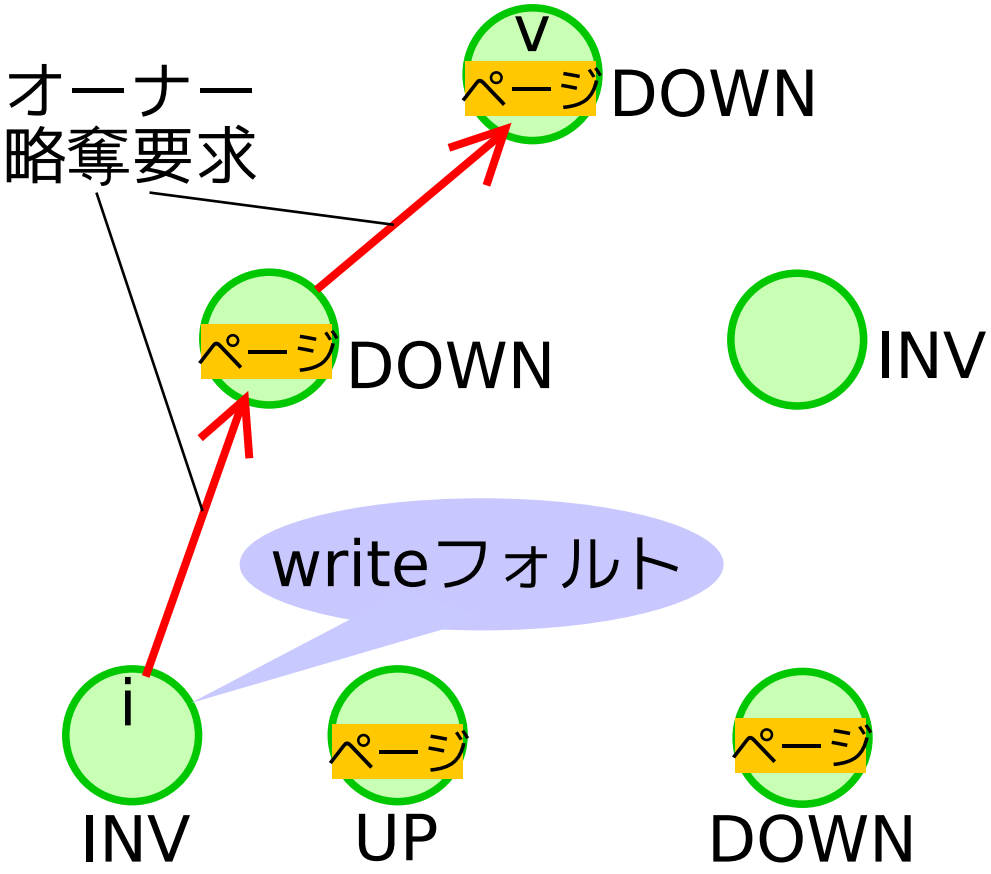


プロトコルの設計規約のまとめ

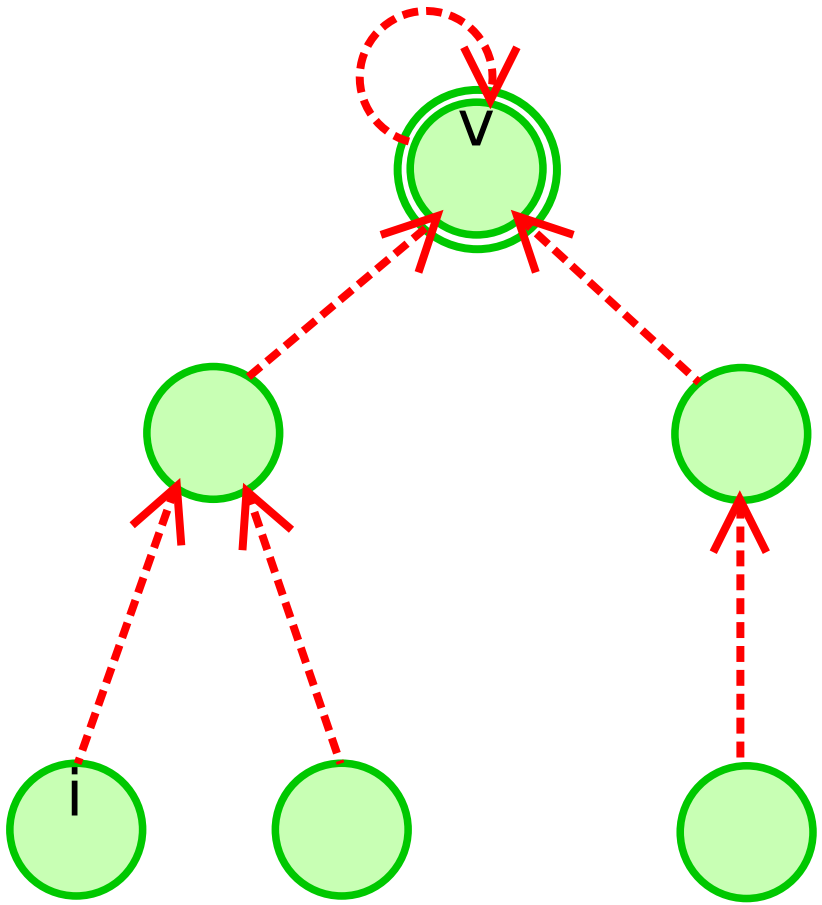


- **この設計規約を守れば，クライアント・サーバ方式のイメージで多様で複雑なプロトコルを容易に設計可能**

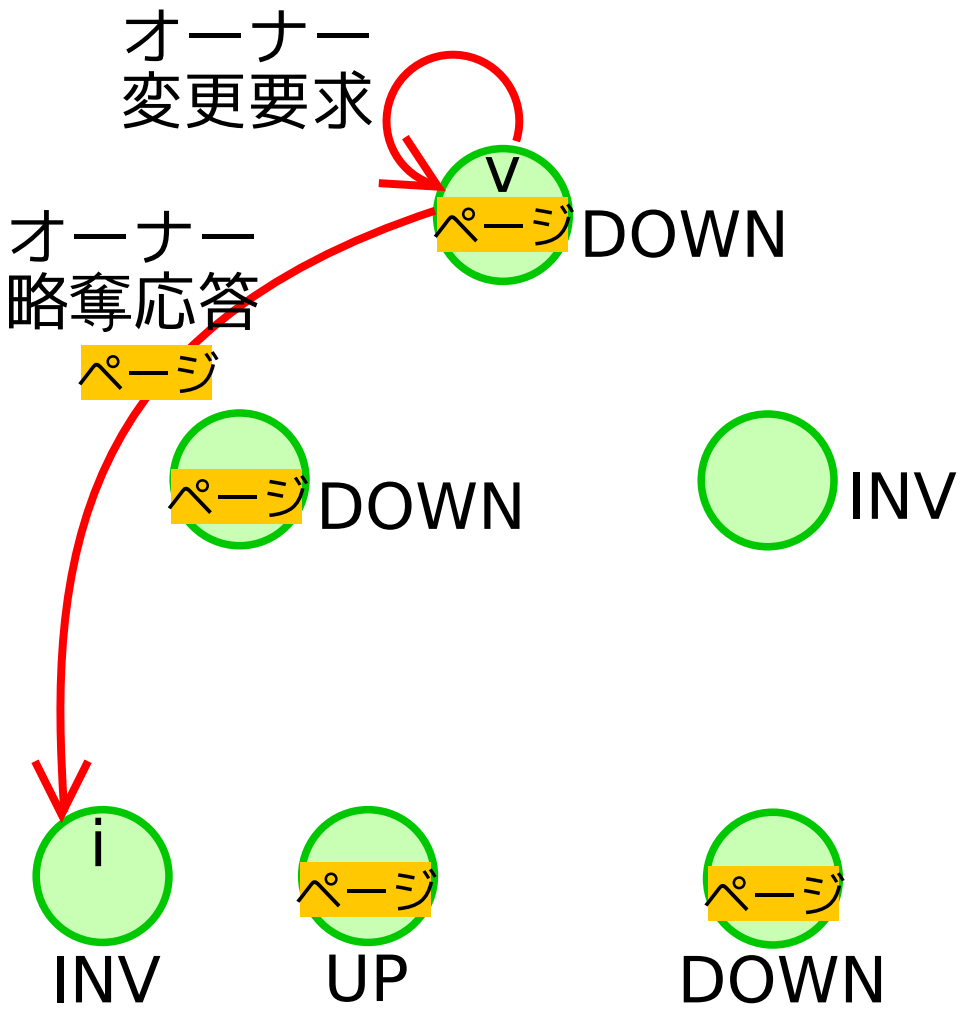
具体例：write フォルト [WRITE_LOCAL](1)



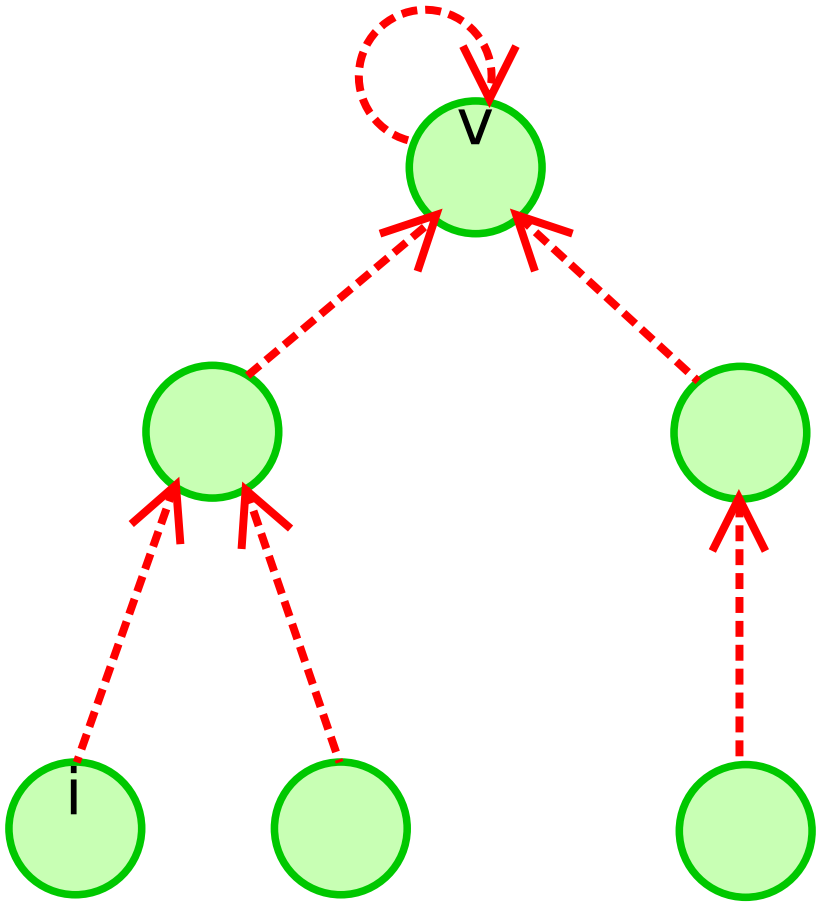
オーナー追跡グラフ



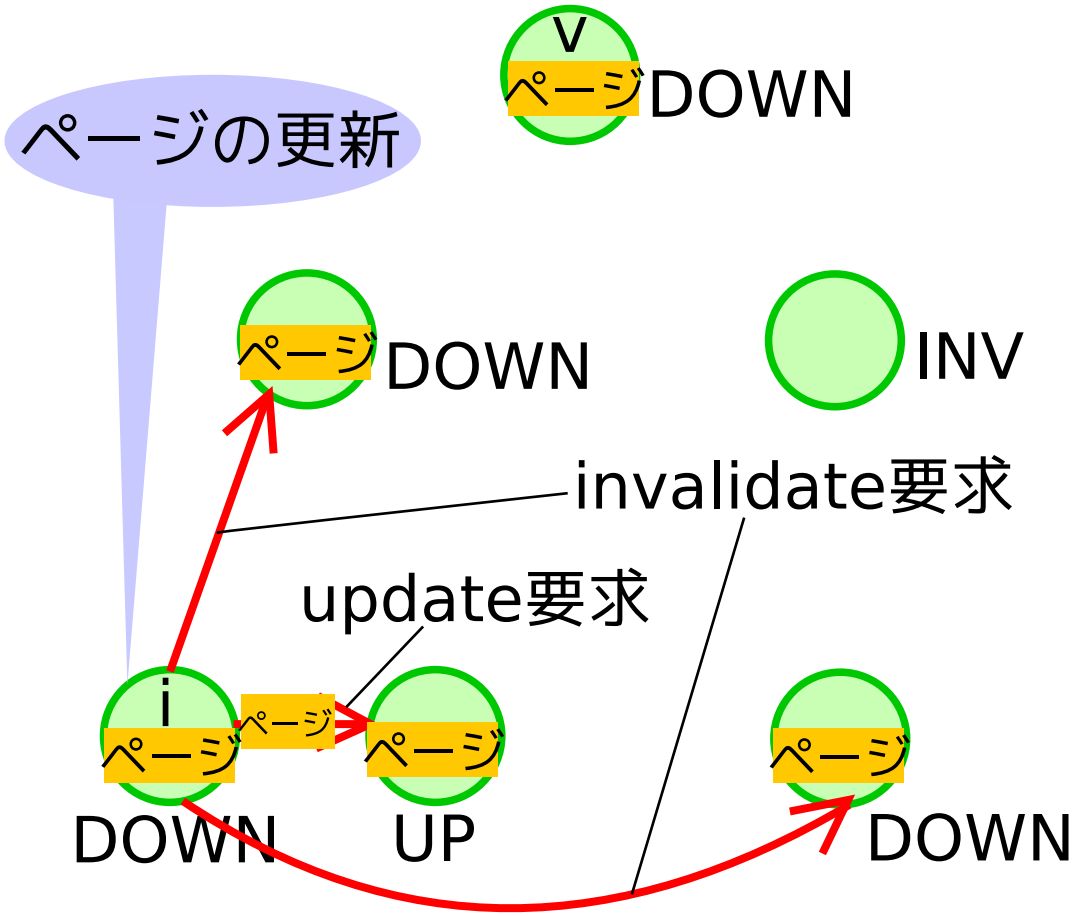
具体例：write フォルト [WRITE_LOCAL](2)



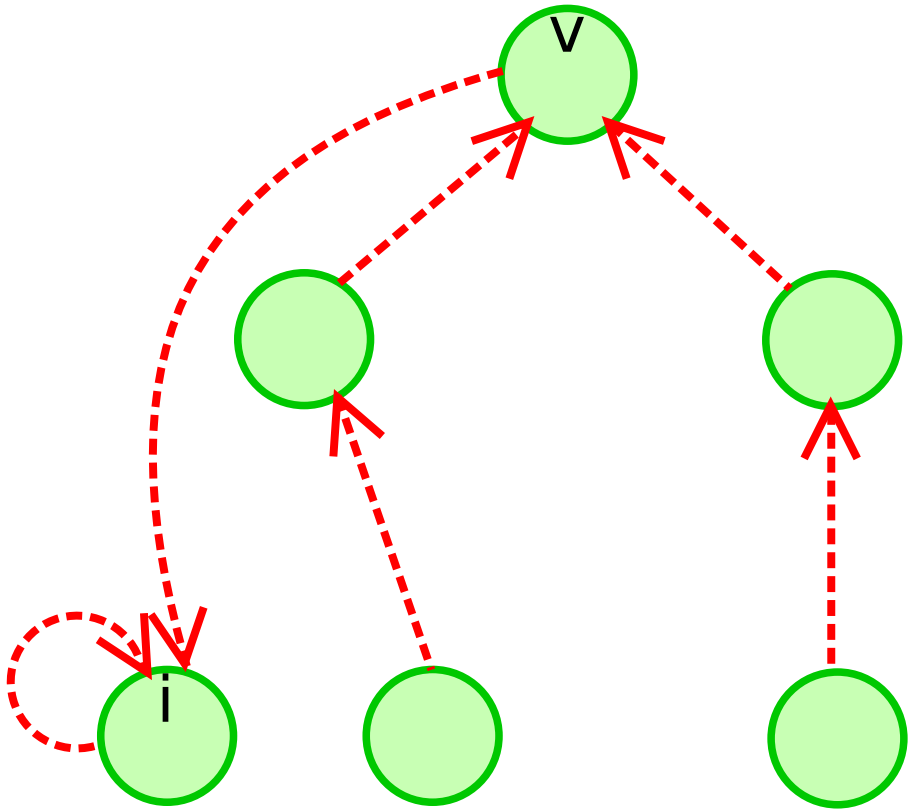
オーナー追跡グラフ



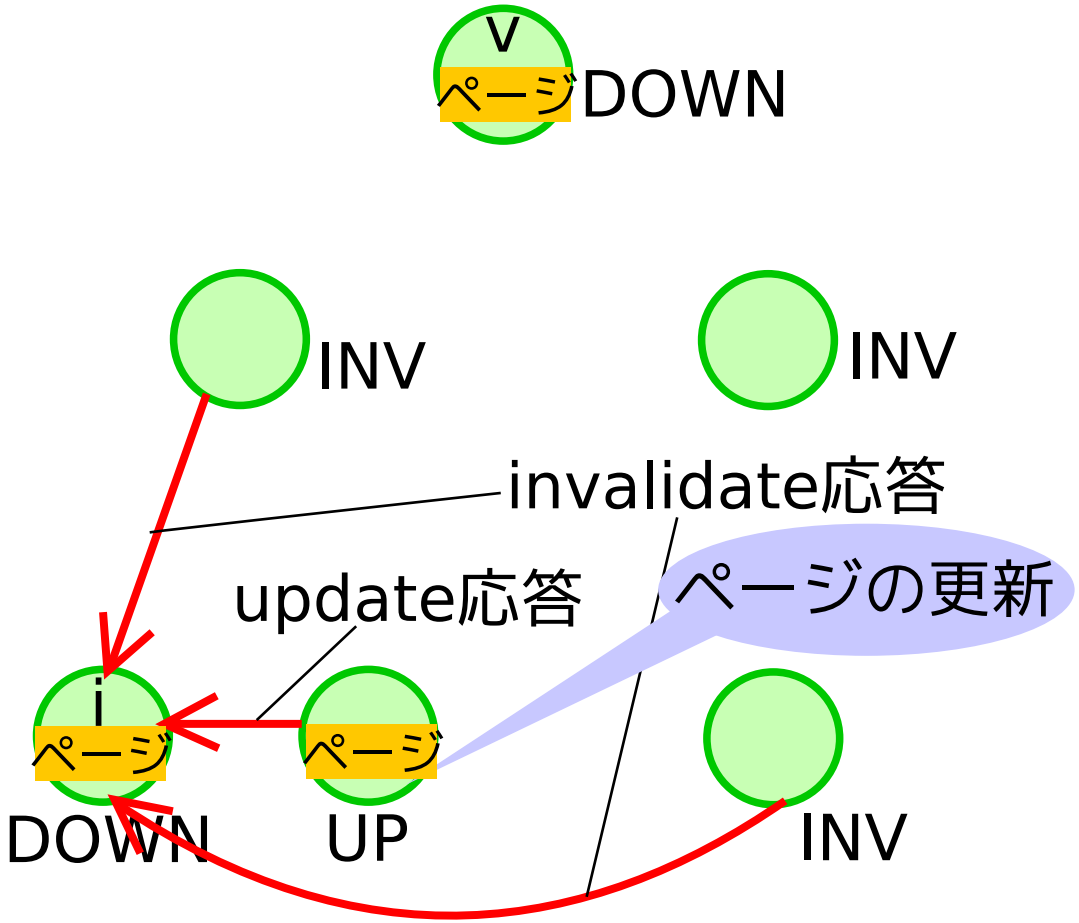
具体例：write フォルト [WRITE_LOCAL](3)



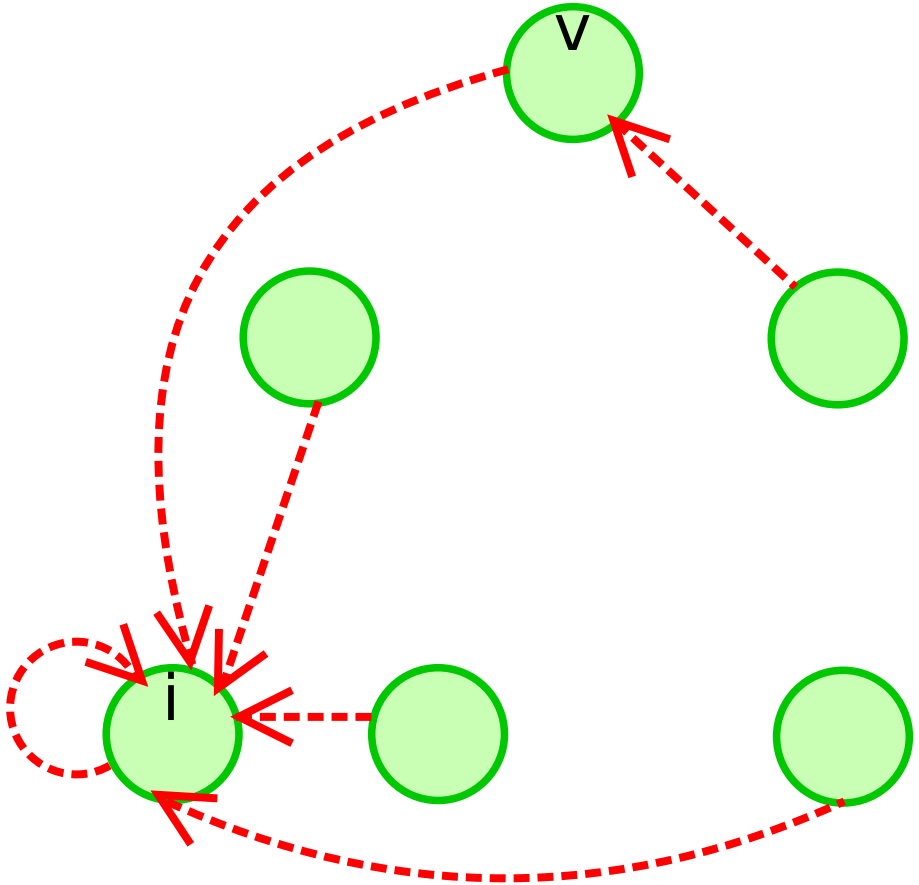
オーナー追跡グラフ



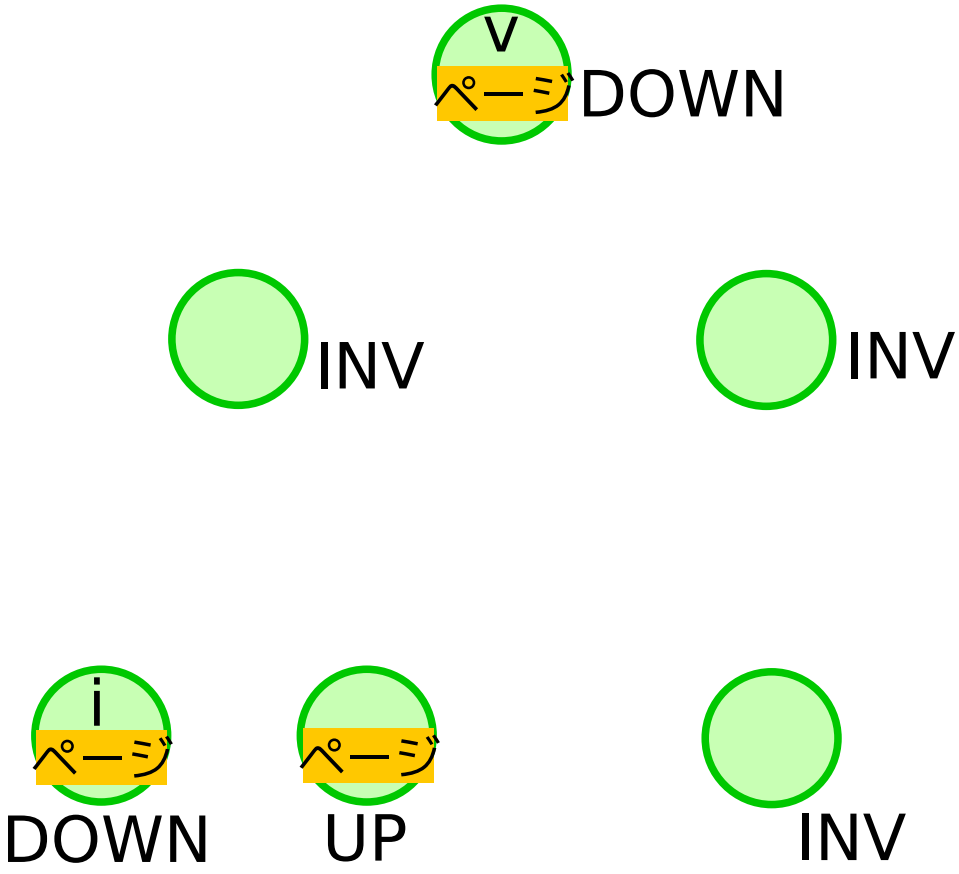
具体例：write フォルト [WRITE_LOCAL](4)



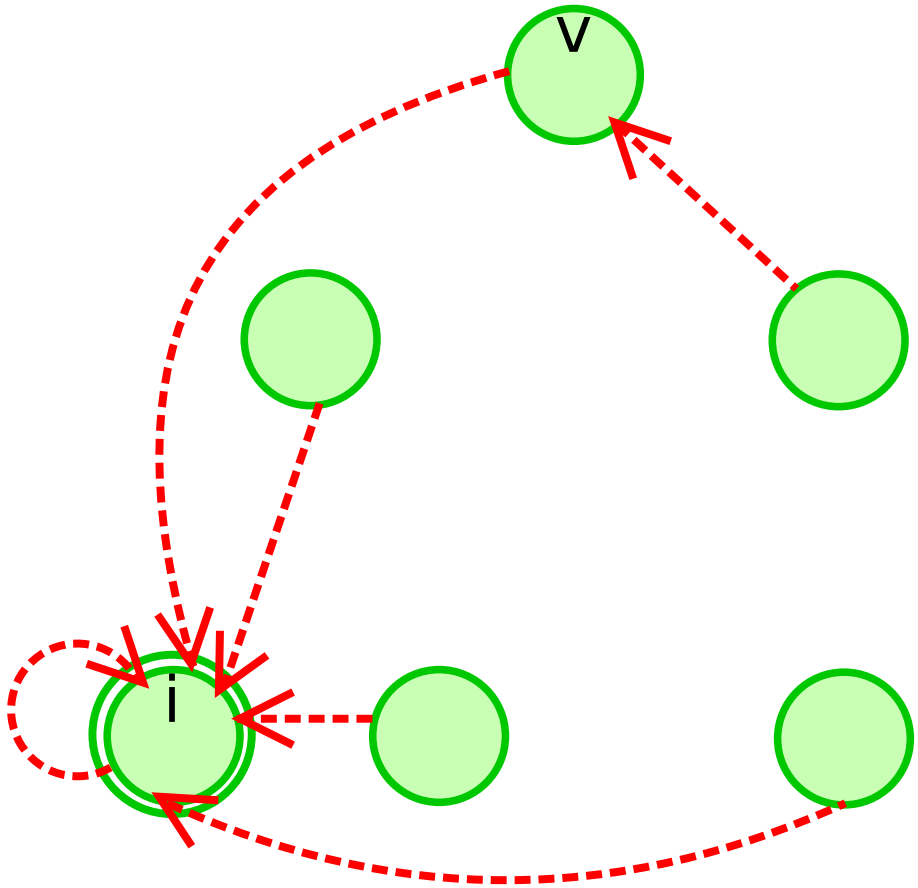
オーナー追跡グラフ



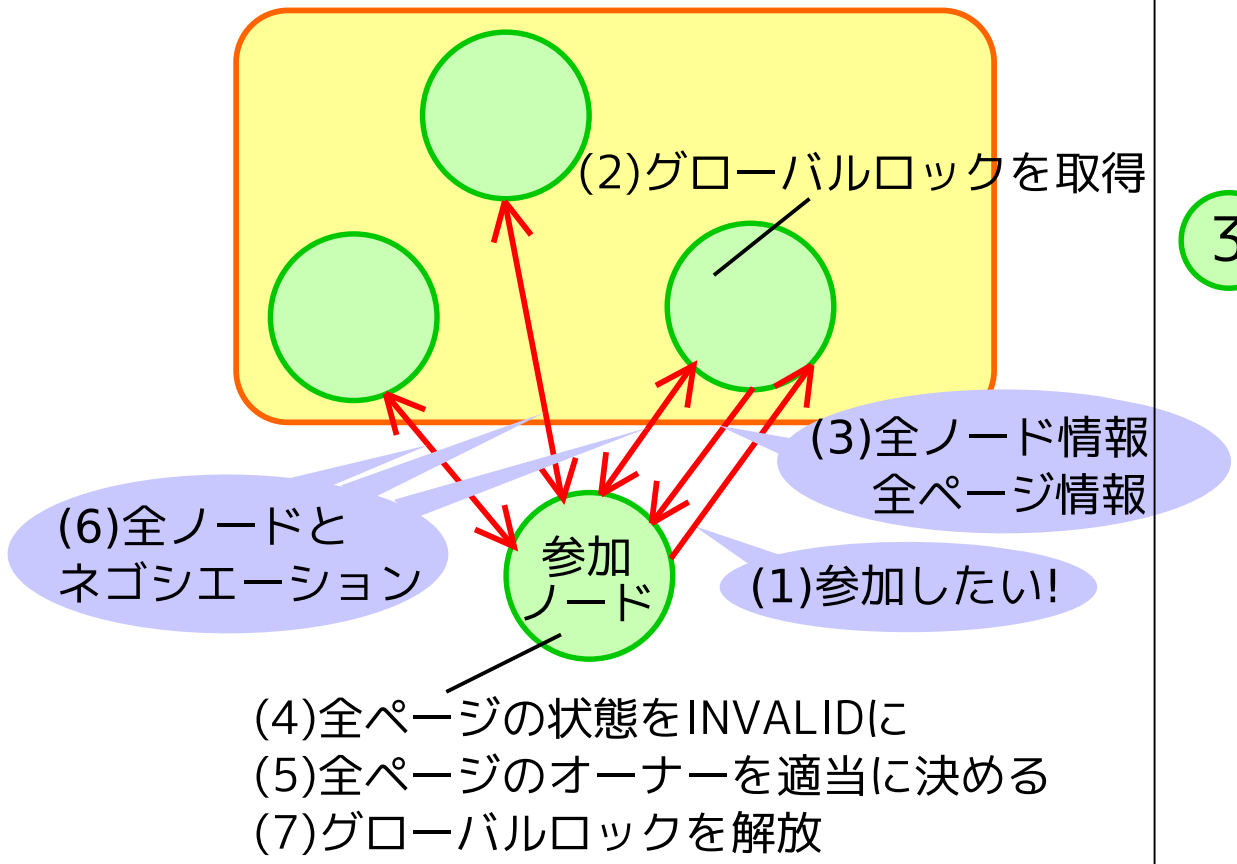
具体例：write フォルト [WRITE_LOCAL](5)

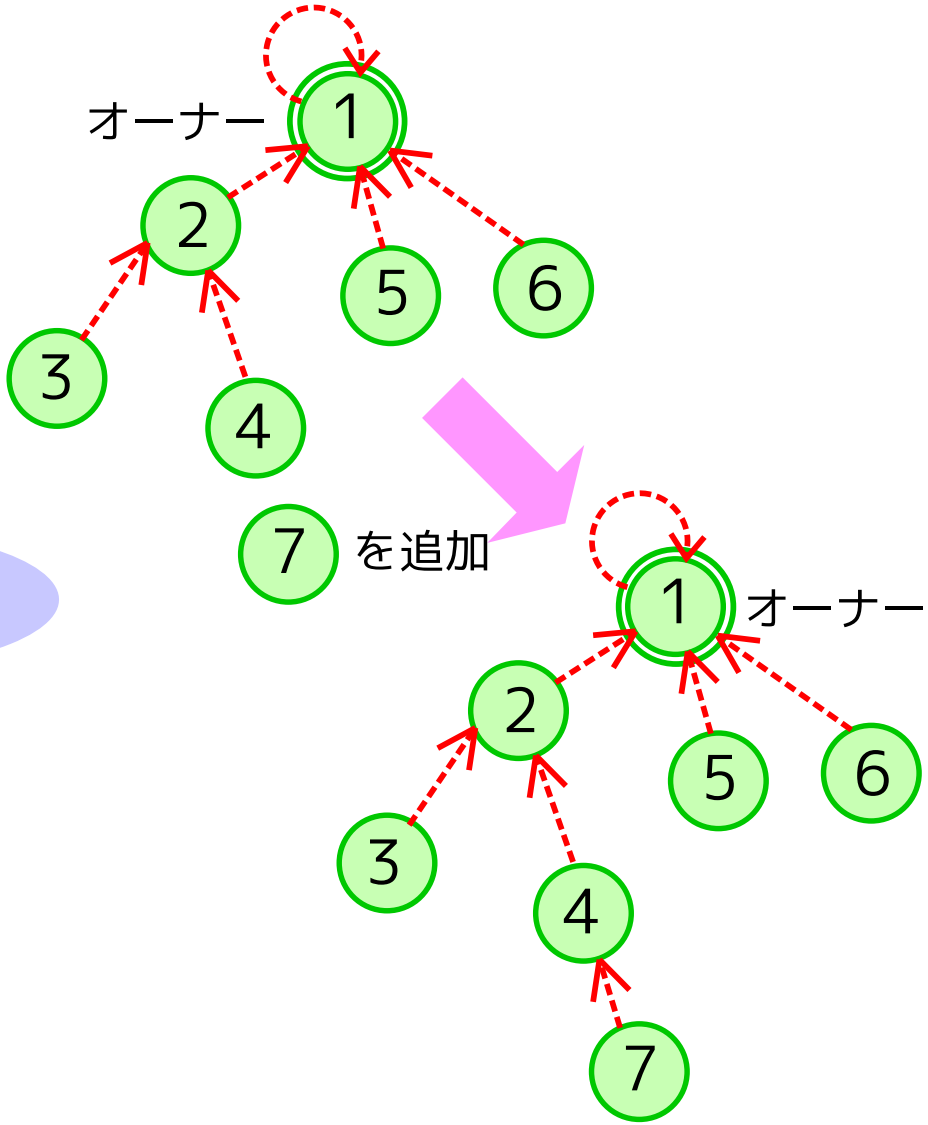


オーナー追跡グラフ

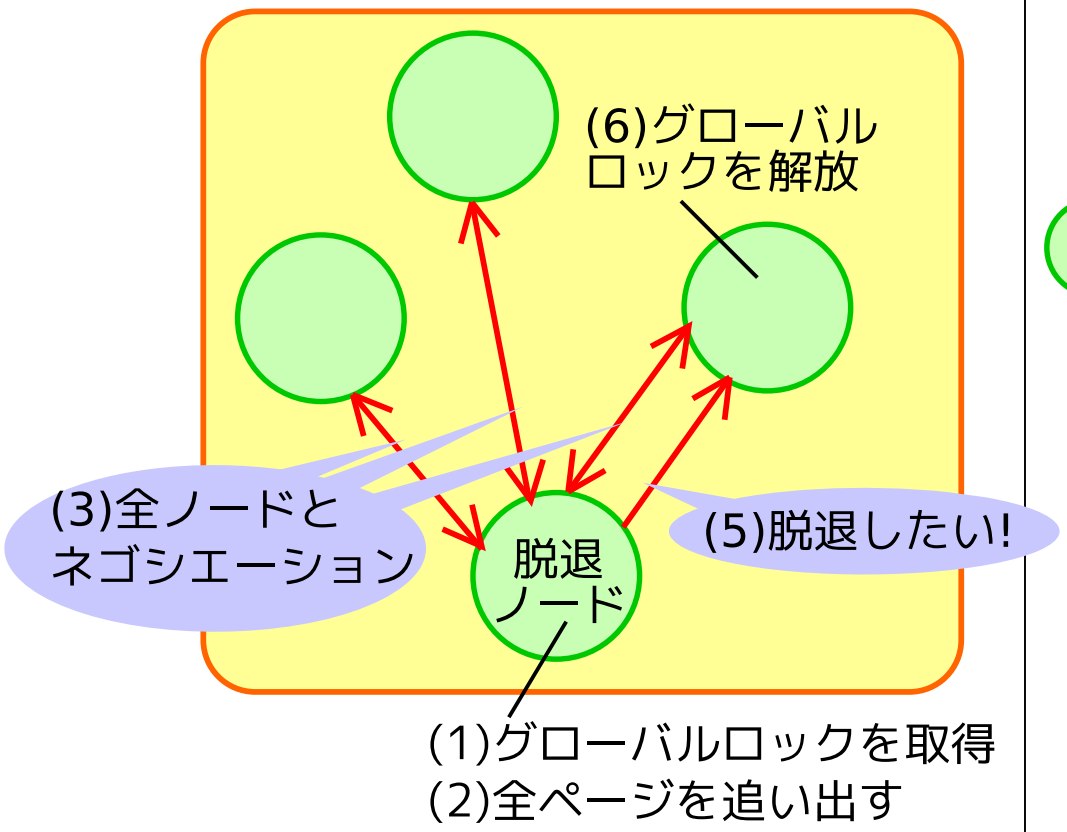


ノードの参加

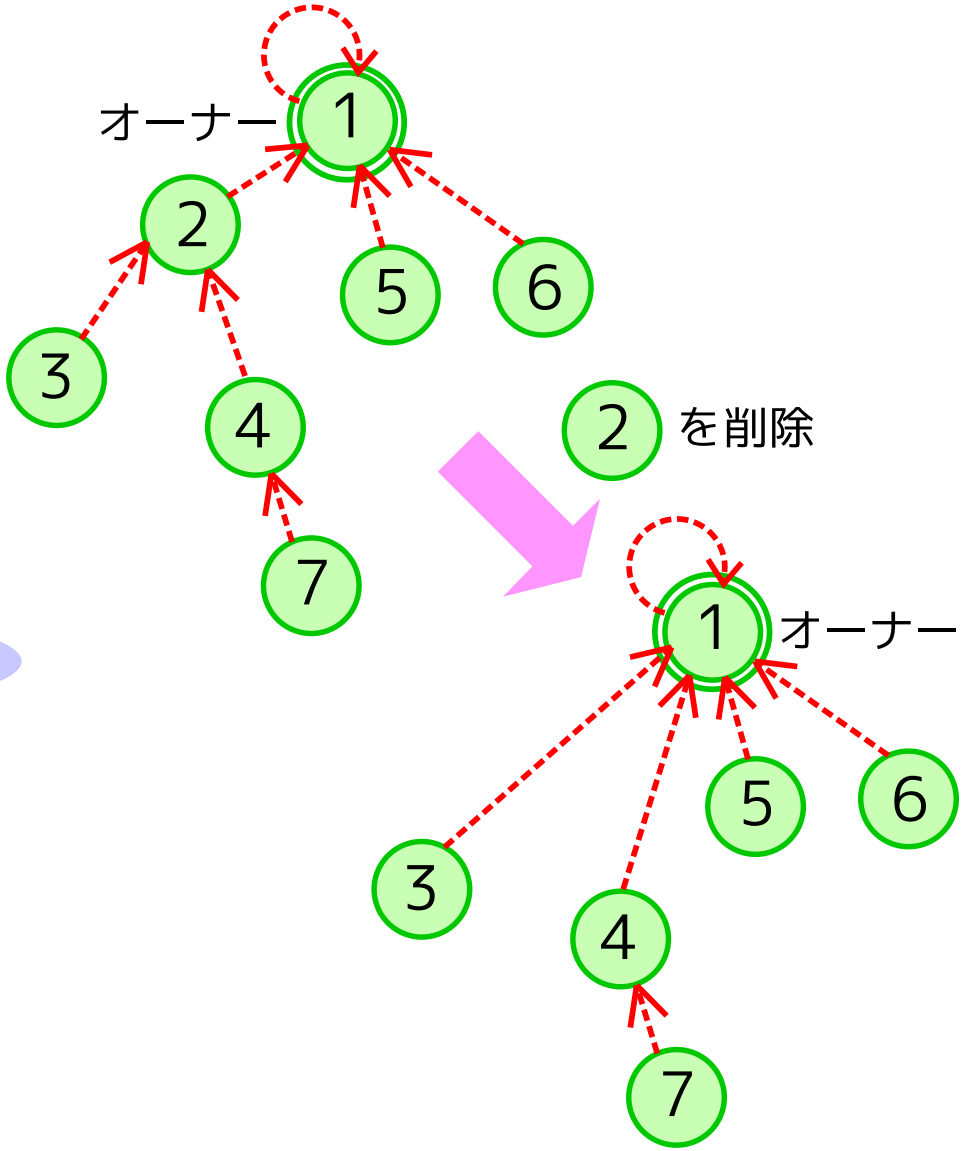

(2)グローバルロックを取得
(3)全ノード情報
全ページ情報
(6)全ノードと
ネゴシエーション
参加
ノード
(1)参加したい!
(4)全ページの状態をINVALIDに
(5)全ページのオーナーを適当に決める
(7)グローバルロックを解放


オーナー
1
2
3
4
5
6
7
を追加
1
オーナー
2
3
4
5
6
7

ノードの脱退



(4)全ノード上でオーナー追跡グラフの再形成

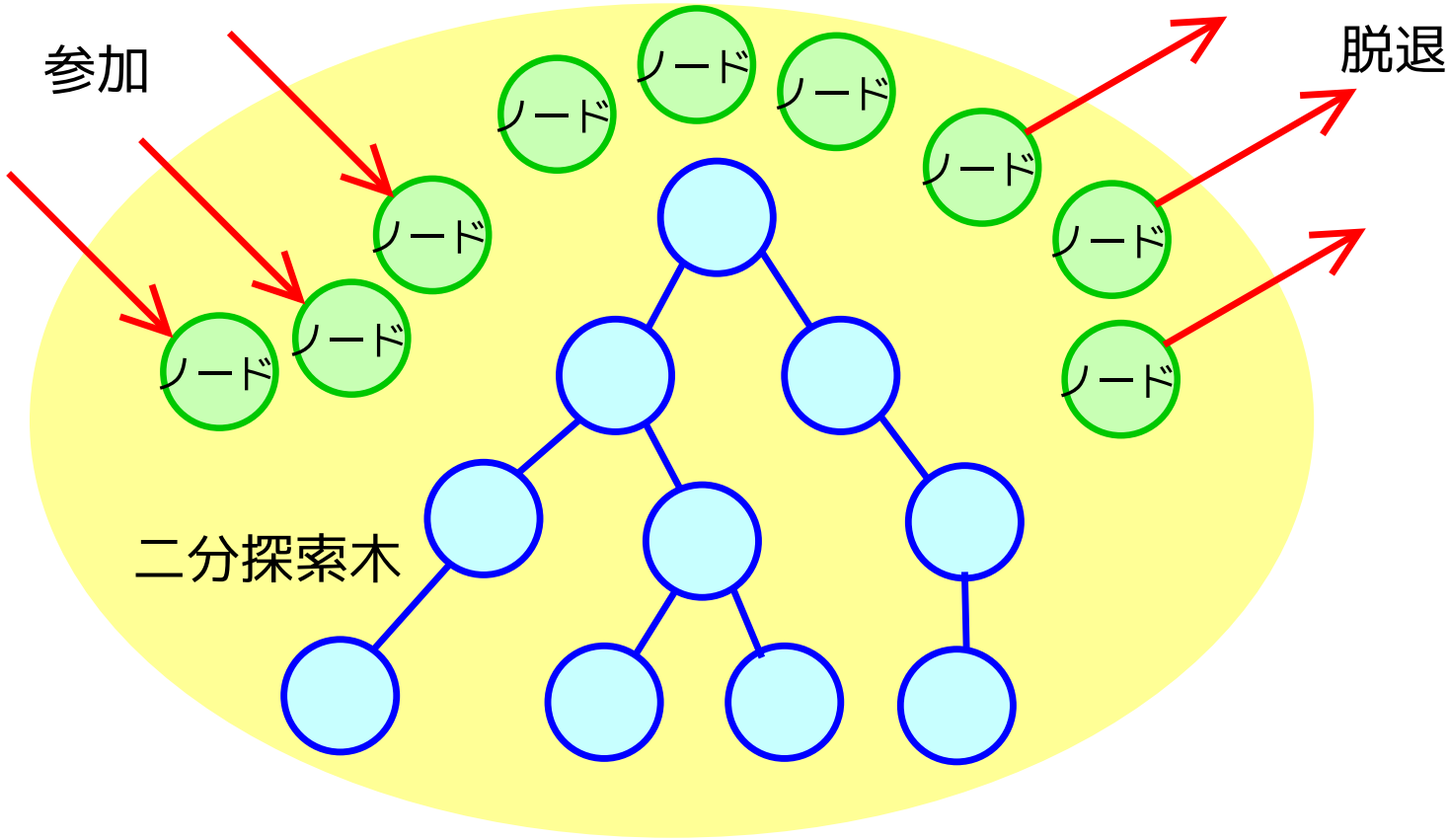


5. 性能評価

- 一部の API や機能は未実装

実験：二分探索木への並列なデータの挿入/削除

- 実験：二分探索木に対して，動的に参加/脱退する多数のノードが適切な排他制御を行いながらデータを挿入/削除
  - → 動的で複雑なグラフ構造を扱う処理
- 種別：共有メモリベースだからこそ記述できるアプリ

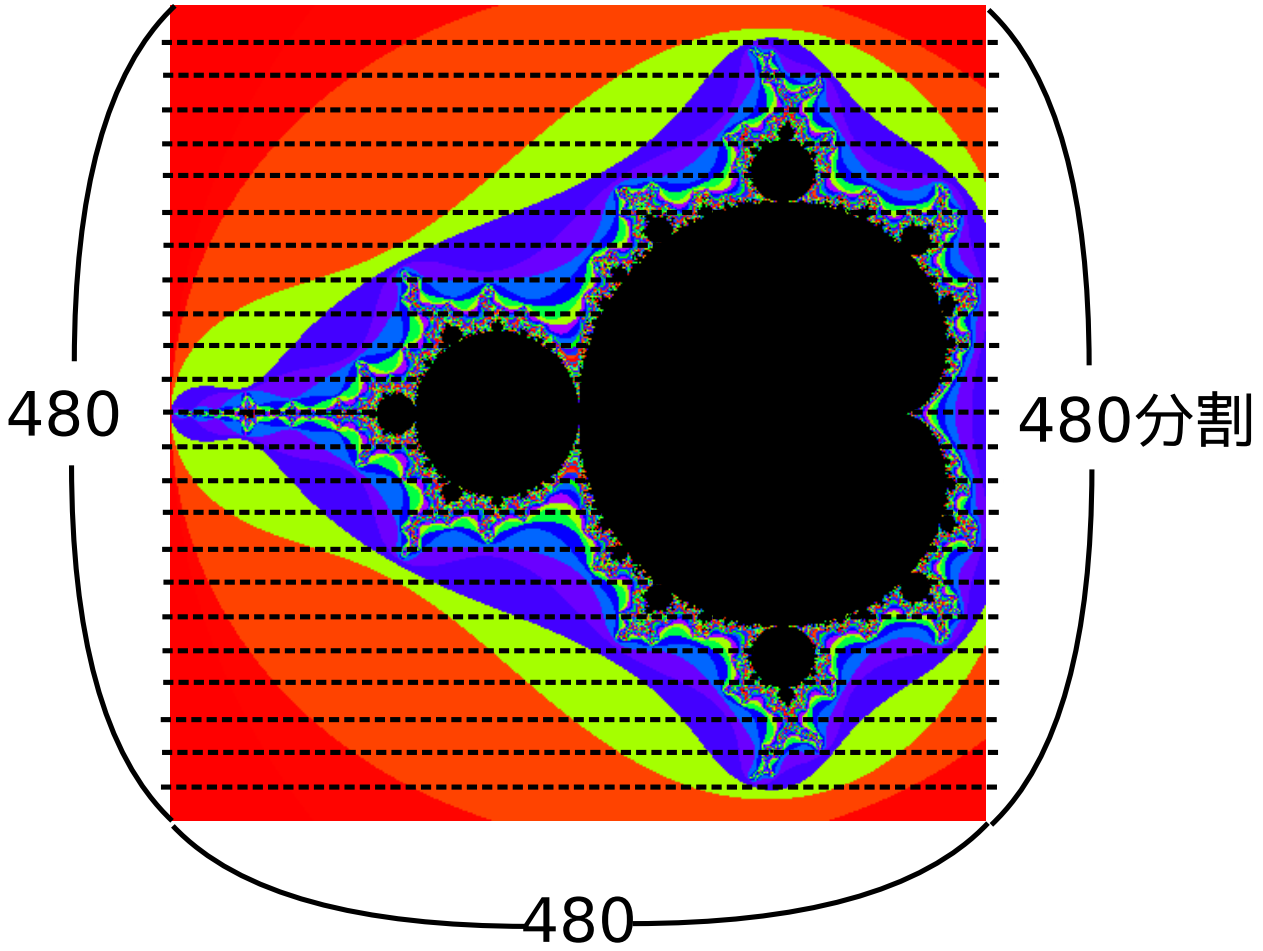


結果：二分探索木への並列なデータの挿入/削除

- 結果：
  - 22 ノード 88 スレッドを動的に参加/脱退させても，二分探索木の中身が正しくソートされた状態で計算が継続
  - pthread プログラムに対する機械的な思考に基づく変換作業で DMI のプログラムが得られた
    - pthread：663 行，DMI：759 行
- 従来の処理系では動的な参加/脱退をサポートできなかったような，多数の計算資源が密に協調するアプリ領域に対しても，DMI のアプローチが適用できる可能性を示唆

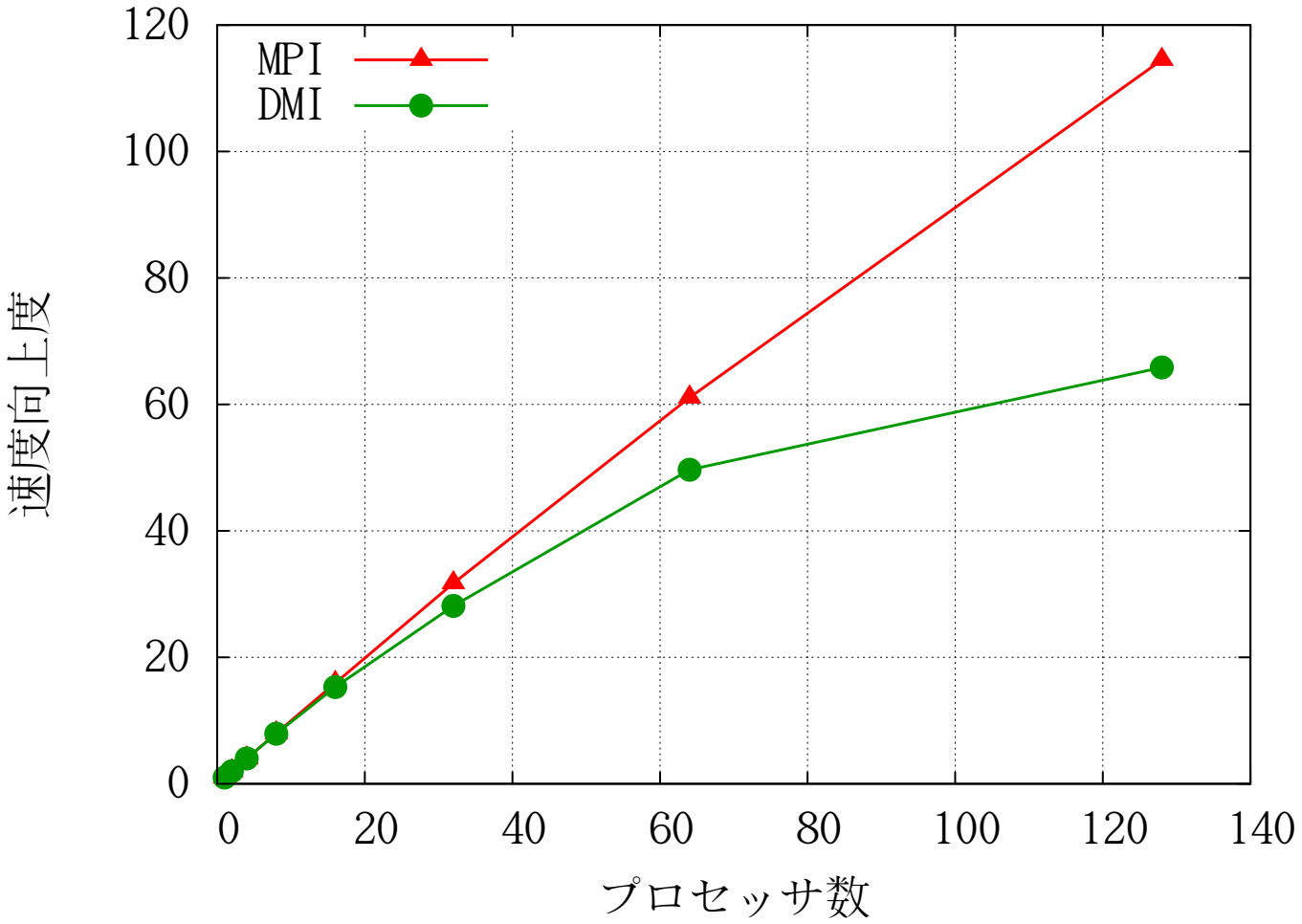
実験：マンデルブロ集合の並列描画

- 実験：$z_0 = 0, z_{n+1} = z_n^2 + c$ なる複素数列 $\{z_n\}$ が $n \to \infty$ で発散しない c の範囲を並列描画
  - → 480×480 の描画領域を横に 480 分割したマスタ・ワーカ方式で記述
- 種別：Embarrassingly Parallel なアプリ



結果：マンデルブロ集合の並列描画

- 結果：
  - → 32 プロセッサ程度までは MPI 並にスケール
  - → 128 プロセッサ時の並列度は，MPI が 115，DMI が 66



実験：遠隔スワップを利用した連続アクセス

- 実験：連続アクセスに関して，1GbE 環境での DMI_read()/非同期 DMI_read() と HDD アクセスを性能比較
  - → DMI_read()：
    - ページサイズ 100MB のページ 160 個を 20 ノード上に分散確保
    - 各ノードの使用メモリ量上限を 4GB に設定
    - ノード 0 で 16GB を連続アクセス
  - → HDD：16GB のファイルを連続アクセス

HDDアクセス：

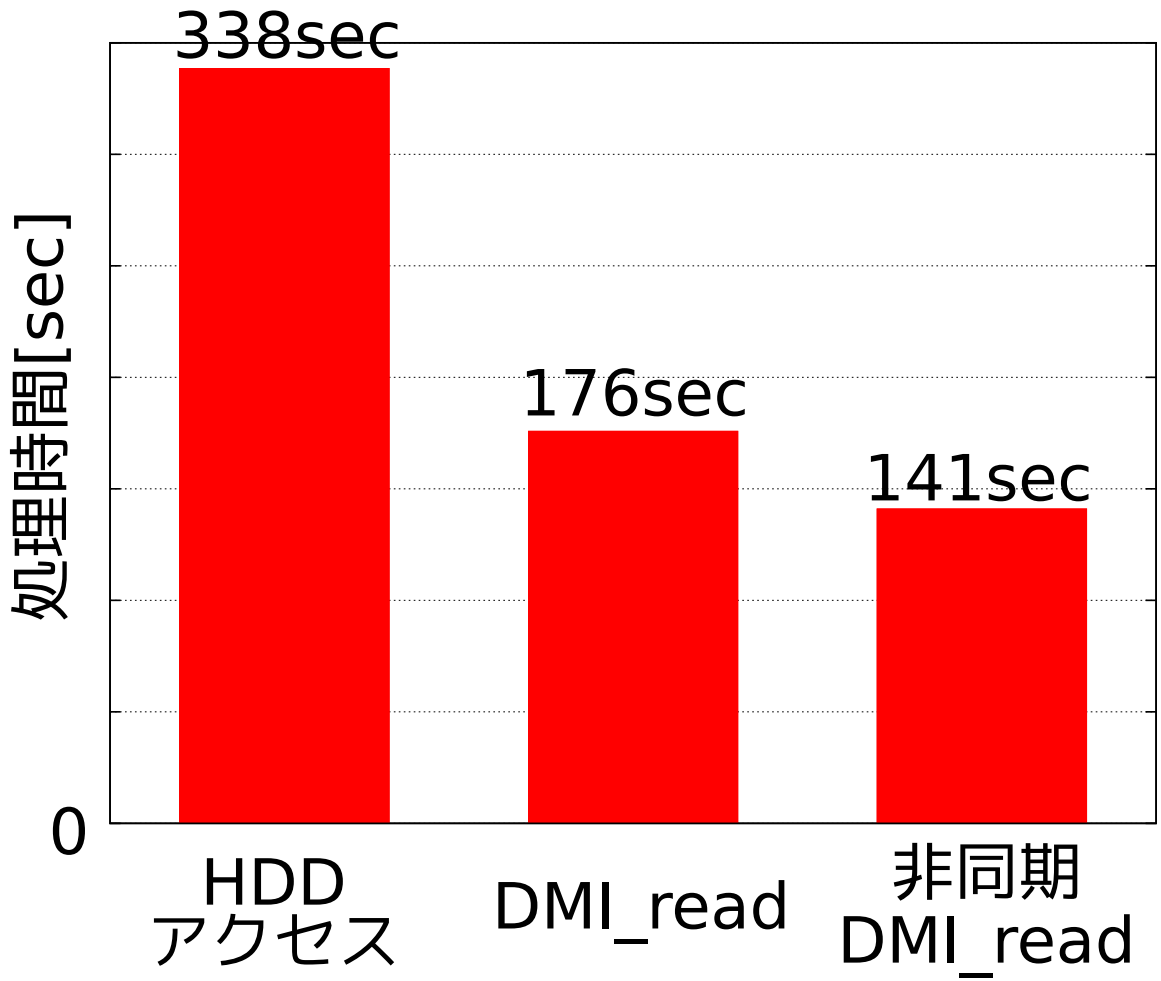
```
for (i = 0; i < 160; i++) {
    fread(p, 100MB, 1, file);
    for (j = 0; j < 100MB; j++) {
        p[j];
    }
}
```

DMI：

```
for (i = 0; i < 160; i++) {
    DMI_read(addr, 100MB, p);
    for (j = 0; j < 100MB; j++) {
        p[j];
    }
}
```

結果：遠隔スワップを利用した連続アクセス

- 結果：
  - → DMI_read() は HDD アクセスの 1.9 倍高速
  - → 非同期 DMI_read() は HDD アクセスの 2.4 倍高速



❖ 6. 結論

まとめ

- DMI の特徴：
  - (1) 動的な参加/脱退に対応するコンシステンシプロトコルを新たに開発
  - (2) 参加/脱退に伴う動的な並列度変化を pthread 型のスタイルによって容易に記述可能
  - (3) スレッドプログラミングと対応する API や同期機構
  - (4) 並列実行環境 + 遠隔スワップシステム
  - (5) 細粒度で明示的な最適化手段
- 従来の処理系では動的な参加/脱退をサポートできなかったような，多数の計算資源が密に協調するアプリに対しても，DMI のアプローチを適用できる可能性

今後の課題

- 一部の未実装機能を実装
- 緻密な性能評価
  - → 特に遠隔スワップシステム
- プロトコルの再検討
  - → 今のプロトコルは各ノードが「システム全体」の知識を持っていることが前提
    - ♦ 本当に参加/脱退が重要になるような大規模な計算環境に対応できない
  - → 各ノードがもっと「局所的」な知識に基づいて動作するように改善

Are there any questions?

Thank you!