Progress Report

修士 2 年
金田憲二

May 7, 2002

1 Execution Example

Figure 1 で示されている．

2 Preliminaries

ネットワークを (無向) グラフ $G = (V, E)$ と表す．ただし，V はノード (計算機) の集合，E はノード間の (TCP) コネクションである．$|V| = N$ とする．

とりあえず簡単のため，個々の計算機は自分の張ることの出来るコネクション全てを張っておくとする．

Definition 2.1 $Neigh_u$ はノード u の隣接ノードの集合とする．

$$Neigh_u \quad = \quad \{v \in V | (u, v) \in E\}$$

Definition 2.2 Q_{vu} はノード v からノード u へのメッセージが蓄えられる *queue*

Definition 2.3 ノード u に割り当てられている仮想ノード番号集合を Δ_u と表す．Δ_u は，連続した自然数からなる集合．

初期状態で *root node* に割り当てられる仮想ノード番号集合を Δ_{all} と表す *(*例*:* $\Delta_{all} = \{0, 1, \ldots, 2^{16} - 1\}$*)*．計算中の仮想ノード番号集合の総和となる．

また，アルゴリズムは event-driven な形で記述されている．つまり，アルゴリズムは { *guard* } *body* からなる *action* のリストで表され，*guard* が真の時, *body* は実行され得る． *body*h の実行は atomic に行われる．

3 Join/Leave Algorithm

3.1 Description of the Algorithm

Figure 2, 3, and 4 show the algorithm of joining / leaving of a node.

```
#include "phoenix.h"


int main(int argn, char ** argv, char ** envp) {
  ph_env_t env;

  env = phoenix_initialize(argn, argv, envp);

  phoenix_join(env);

  if (phoenix_is_root(env)) {
    ph_vp_t i;
    char buf[1000];
    ssize_t n;

    /* メッセージを 0 から 15 番に送信 */
    for (i = 0UL; i <= 15UL; i++) {

      snprintf(buf, 1000, "%lu", i);
      phoenix_send(i, buf, strlen(buf)+1, env);

      printf("send message to %lu\n", i);
    }
  }

  sleep(10);

  /* メッセージの受信を繰り返す */
  while (1) {
    char buf[1000];
    ssize_t n;

    bzero(buf, 1000);
    phoenix_recv(buf, 1000, env);
    printf("recv message from %s\n", buf);
  }

  phoenix_leave(env);
  phoenix_finalize(env);
}
```

Figure 1: Sample Program

```
1  var
2    Neigh_u : set of nodes;
3    Δ_u : set of vps;
4
5
6  Initialization :
7    begin
8      if u is a root then begin
9        Δ_u := Δ_all
10     else begin
11       Δ_u := ∅
12     end
13   end
```

Figure 2: Join/Leave Algorithm (Part 1, for node u)

3.2 Invariants of Algorithm

名前集合が個々のノードで disjoint となり，かつその和集合は不変であることを保証したい．具体的には，以下に述べられる assertion P と R が invariant であることを証明する．

The following assertion P states that $\Delta_{v_1}, \Delta_{v_2}, \ldots, \Delta v_n$ are disjoint.

Definition 3.1

$$P \quad \equiv \quad \forall v, w \in V, (v \neq w) \Rightarrow (\Delta_v \cap \Delta_w = \varnothing)$$

Definition 3.2 *Let the predicate $up(u, w)$ be true if and only if a (bidirectional) channel between u and w exists and is operating.*

$$\textbf{\textit{stable}} \quad \equiv \quad \forall v, w \in V, up(u, w) \Rightarrow Q_{wu} = \varnothing$$

Then, the following assertion R states that the union of Δ_v does not change through entire computation.

$$R \quad \equiv \quad \textbf{\textit{stable}} \Rightarrow \bigcup_{v \in V} \Delta_v = \Delta_{all}$$

4 Routing Algorithm

4.1 Packet Forwarding by Landmark Routing

仮想ノード番号を，2^B 進数で表記する (ただし $B \in \mathbf{Z}^+$)．

Definition 4.1 整数 n を 2^B 進数で表記した際の，左から i 桁目の値を，$n[i]$ と表す．また，n の桁数を $length(n)$ とする．

```
1  Procedure Join() :
2    begin
3      send ⟨join⟩ to w s.t. w ∈ Neigh_u
4    end
5
6
7  Processing a ⟨join⟩ message from neighbor w :
8    {A ⟨join⟩ is at the head of Q_wu }
9    begin
10     receive ⟨join⟩ from w ;
11     if Δ_u ≠ ∅ then begin
12       (* 委譲する Δ の決定 *)
13       Δ_u := Δ_u/Δ ;
14       send ⟨accept-join, Δ⟩ to w ;
15       (* 必要なら Application Object を w に送信 *)
16       Recompute-Routing-Table()
17     else begin
18       send ⟨reject-join⟩ to w
19     end
20   end
21
22
23 Processing a ⟨accept-join, Δ⟩ message from neighbor w :
24   {A ⟨accept-join, Δ⟩ is at the head of Q_wu }
25   begin
26     receive ⟨accept-join, Δ⟩ from w ;
27     (* 必要なら Application Object を w から受信 *)
28     Δ_u := Δ ;
29     Recompute-Routing-Table()
30   end
31
32
33 Processing a ⟨reject-join⟩ message from neighbor w :
34   {A ⟨reject-join⟩ is at the head of Q_wu }
35   begin
36     receive ⟨reject-join⟩ from w ;
37     Join()
38   end
```

Figure 3: Join/Leave Algorithm (Part 2, for node u)

```
1  Procedure Leave() :
2    begin
3      send ⟨leave, Δ_u⟩ to w s.t. w ∈ V and Δ_w is numerically closest to Δ_u ;
4      Δ_u := ∅;
5      (* 必要なら Application Object を w に送信 *)
6      Recompute-Routing-Table()
7    end
8
9
10 Processing a ⟨leave, Δ⟩ message from neighbor w :
11   {A ⟨leave, Δ⟩ is at the head of Q_wu }
12   begin
13     receive ⟨leave, Δ⟩ from w ;
14     (* 必要なら Application Object を w から受信 *)
15     Δ_u := Δ_u ∪ Δ ;
16     Recompute-Routing-Table()
17   end
```

Figure 4: Join/Leave Algorithm (Part 3, for node u)

Definition 4.2

$$\begin{aligned} prefixSet(\Delta_u) &= \{n | \forall \delta \in \Delta_u, \forall i (1 \leq i \leq length(n)), n[i] = \delta[i]\} \\ prefix(\Delta_u) &= max_{n \in prefixSet(\Delta_u)}\{n\} \end{aligned}$$

Example 4.3 $b = 2$,$\Delta_u = \{130, 131, 132, 133\}$ のとき ,$prefixSet(\Delta_u) = \{1, 13\}$, $prefix(\Delta_u) = 13$ となる .

以後 , 簡便のため , $130, 131, 132, 133 = 13*$ といったように表す .

Definition 4.4

$$\begin{aligned} shareSet(\Delta_u, \Delta_v) &= prefixSet(\Delta_u) \cap prefixSet(\Delta_v) \\ share(\Delta_u, \Delta_v) &= max_{n \in sharexSet(\Delta_u, \Delta_v)}\{n\} \end{aligned}$$

Example 4.5 $B = 2$, $\Delta_u = \{131*\}$, $\Delta_v = \{132*\}$ のとき , $shareSet(\Delta_u, \Delta_v) = \{1, 13\}$, $share(\Delta_u, \Delta_v) = 13$ となる .

Definition 4.6

$$\begin{aligned} \Delta_u^{(i,j)} &= \{n | (\forall k (1 \leq k \leq i), n[k] = prefix(\Delta_u)[k]) \wedge (n[i+1] = j)\} \\ Succ_u^{(i,j)} &= \{v \in (Neigh_u \cup \{u\}) | \Delta_v \cap \Delta_u^{(i,j)} \neq \varnothing\} \end{aligned}$$

ただし , $0 \leq i < \lceil log_{2^B} N \rceil$, $0 \leq j < 2^B$

簡単にいうと , $\Delta_u^{(i,j)}$ は , prefix の i 桁が u と一致し , $(i+1)$ 桁が j である仮想ノード集合 (ルーティングテーブルの i 行 j 列目) , $Succ_u^{(i,j)}$ は , $\Delta_u^{(i,j)}$ へのメッセージの転送先のノード集合 .

Example 4.7 $B = 2$, $N = (2^2)^3$,
$u, v_1, v_2, \ldots v_7 \in V$, $v_1, v_2 \ldots v_7 \in Neigh_u$,

$$
\begin{aligned}
\Delta_u &= \{00*\}, \\
\Delta_{v_1} &= \{01*\}, \\
\Delta_{v_2} &= \{02*\}, \\
\Delta_{v_3} &= \{03*\}, \\
\Delta_{v_4} &= \{10*\} \cup \{11*\}, \\
\Delta_{v_5} &= \{12*\} \cup \{13*\}, \\
\Delta_{v_6} &= \{2**\}, \\
\Delta_{v_7} &= \{3**\},
\end{aligned}
$$

とするとき，

$\Delta_u^{(i,j)}$ は以下のようになる．

$i \setminus j$	0	1	2	3
0	{0** }	{1** }	{2** }	{3** }
1	{00* }	{01* }	{02* }	{03* }
2	{000 }	{001 }	{002 }	{003 }

$Succ_u^{(i,j)}$ は以下のようになる．

$i \setminus j$	0	1	2	3
0	$\{u, v_1, v_2, v_3\}$	$\{v_4, v_5\}$	$\{v_6\}$	$\{v_7\}$
1	$\{u\}$	$\{v_1\}$	$\{v_2\}$	$\{v_3\}$
2	$\{u\}$	$\{u\}$	$\{u\}$	$\{u\}$

u から δ へのメッセージは以下のような条件を満たす path $(v_0, v_1, \ldots, v_k)$ で転送することとなる．

$$
\begin{aligned}
v_0 &= u \\
v_{i+1} &\in Succ_{v_i}^{(i,\delta[i+1])} \\
\delta &\in \Delta_{v_k}
\end{aligned}
$$

これをアルゴリズムとして図示すると Figure 5 のようになる．

以上に挙げたような routing table を実現するためには，個々のノードが $Succ_u^{(i,j)}$ を把握しておかなければいけない．

ルーティングテーブルの大きさは $\lceil log_{2^B} N \rceil \times 2^b$ ．

ルーティングのホップ数は $O(log_{2^B} N)$ ．

4.2 Packet Forwarding by Landmark Routing + Distance Vector Routing

$Neigh_u = V/\{u\}$ つまり u から他の全ノードに直接コネクションを張ることができれば，$Succ_u^{(i,j)} \neq \varnothing$ が成り立ち，以上のような landmakr routing でメッセージを転送することができる．

しかし，実際には phoenix が構築するネットワークは irregular な graph である．

```
1
2  Processing a ⟨message, δ⟩ message from neighbor w :
3     {A ⟨message, δ⟩ is at the head of Q_wu }
4     begin
5        receive ⟨message, δ⟩ from w;
6        if δ ∈ Δ_u then begin
7           (* 自分宛のメッセージなので，メッセージを受理する *)
8        end else begin
9           (* prefix をより共有する方へメッセージを転送 *)
10          send ⟨message, δ⟩ to v ∈ Succ_u^(i,δ[i+1])
11                (ただし i = length(share({δ}, Δ_u)));
12       end
13    end
14
```

Figure 5: message forwarding (for node u)

そこで，landmark-routing table で送信できないところ ($Succ_u^{(i,j)} = \varnothing$ であるところ) を distance vector routing で送信する．

distance vector routing の基本的な idea は，以下の式を用いて，最短路を計算することにある．

$$d(u,v) \quad = \quad min_{w \in Neigh_u}\{d(u,w) + d(w,v)\}$$

ただし，$d(u,v)$ はノード u からノード v への最短距離である．

上の式で，min となるときの隣接ノード w にメッセージを転送するようにする．

また，右辺第 2 項の情報をノード u は知らない．隣接ノード間でこの情報を通信を行い，交換する必要がある．

個々のノードは自分の routing table が変更する度に，更新された内容を他のノードに教えなければいけない．

4.2.1

まず第一段階として，ルーティングテーブルの削減を行う．

具体的には，

$$d(u,v) \quad = \quad min_{w \in Neigh_u}\{d(u,w) + d(w,v)\}$$

を

$$d(u,\Delta_u^{(i,j)}) \quad = \quad min_{w \in Neigh_u}\{d(u,w) + d(w,\Delta_u^{(i,j)})\}$$

の形にして計算を行っている．

$ndis$ の大きさが $|Neigh_u| \times N$ であったものが，$|Neigh_u| \times \lceil log_{2^B} N \rceil \times 2^B$ になっている．

```
1  var
2     Neigh_u : set of nodes; (* A set of u's neighbors *)
3     Δ_u : set of vps;
4     Dis_u : array of 0 ... N; (* Dis_u[Δ_u^(i,j)] estimates distance from u to Δ_u^(i,j) *)
5     Succ_u : array of nodes; (* Succ_u[Δ_u^(i,j)] is a successor in a path from u to
      Δ_u^(i,j) *)
6     ndis_u : array of 0 ... N; (* ndis_u[w, Δ_u^(i,j)] estimates d(w, Δ_u^(i,j)) *)
7
8
9  Initialization :
10    begin
11       forall w ∈ Neigh_u, i(0 ≤ i < ⌈log_{2^B} N⌉), j(0 ≤ j < 2^B) do begin
12          ndis_u[w, Δ_u^(i,j)] := ∞;
13       end;
14       forall i(0 ≤ i < ⌈log_{2^B} N⌉), j(0 ≤ j < 2^B) do begin
15          if Δ_u^(i,j) ⊆ Δ_u then begin
16             Succ_u[Δ_u^(i,j)] := u;
17             Dis_u[Δ_u^(i,j)] := 0;
18          end else begin
19             Succ_u[Δ_u^(i,j)] := udef;
20             Dis_u[Δ_u^(i,j)] := ∞;
21       end
22       forall w ∈ Neigh_u do begin
23             send ⟨update, Δ_u, 0⟩ to w
24       end
25    end
26
27
```

Figure 6: Routing Algorithm (Part 1, for node u)

```
1  Procedure Recompute-Routing-Table(Δ_u^(i,j)) :
2    begin
3      if Δ_u^(i,j) ⊆ Δ_u then begin
4        Succ_u[Δ_u^(i,j)] := u;
5        Dis_u[Δ_u^(i,j)] := 0;
6      end else begin
7        d := 1 + min_{w∈Neigh_u}{ndis_u[w, Δ_u^(i,j)]};
8        if d < ∞ then begin
9          Dis_u[Δ_u^(i,j)] := d;
10         Succ_u[Δ_u^(i,j)] := w with 1 + ndis_u[w, Δ_u^(i,j)] = d;
11       end else begin
12         Dis_u[Δ_u^(i,j)] := ∞;
13         Succ_u[Δ_u^(i,j)] := udef;
14       end
15     end;
16     if Dis_u[Δ_u^(i,j)] has changed then begin
17       forall v ∈ Neigh_u do begin
18         send ⟨update, Δ_u^(i,j), Dis_u[Δ_u^(i,j)]⟩ to v
19       end
20     end
21   end
22
23
```

Figure 7: Routing Algorithm (Part 2, for node u)

```
1  Processing a ⟨update, Δ, d⟩ message from neighbor w :
2    {A ⟨update, Δ, d⟩ is at the head of Q_wu }
3    begin
4      receive ⟨update, Δ, d⟩ from w;
5      ndis_u[w, Δ] := d;
6      Recompute-Routing-Table(Δ);
7    end
```

Figure 8: Routing Algorithm (Part 3, for node u)

```
1  Upon failure of channel uw :
2    begin
3      receive ⟨fail, w⟩ ;
4      Neigh_u := Neigh_u/{w} ;
5      forall i(0 ≤ i < ⌈log_{2^B} N⌉), j(0 ≤ j < 2^B) do begin
6        Recompute-Routing-Table(Δ_u^{(i,j)});
7      end
8    end
9
10
11 Upon repair of channel uw :
12   begin
13     receive ⟨repair, w⟩ ;
14     Neigh_u := Neigh_u ∪ {w} ;
15     forall i(0 ≤ i < ⌈log_{2^B} N⌉), j(0 ≤ j < 2^B) do begin
16       ndis_u[w, Δ_u^{(i,j)}] := ∞;
17       send ⟨update, Δ_u^{(i,j)}, Dis_u[Δ_u^{(i,j)}]⟩ to w
18     end
19   end
```

Figure 9: Routing Algorithm (Part 4, for node u)

4.2.2

次にコネクションの数の削減を目指す．

Figure 10 は，ルーティングに使用していないコネクションを削除するアルゴリズム．これを使って，コネクションの追加・削除を定期的に行い，コネクションの数を制限する．

```
1  Procedure Delete-Connection() :
2    begin
3      (* 削除しても問題ないコネクション v(∈ Neigh_u/Succ_u) の選択 *)
4      send ⟨delete⟩ to v
5    end
6
7
8  Processing a ⟨delete⟩ message from neighbor w :
9    {A ⟨delete⟩ is at the head of Q_wu }
10   begin
11     receive ⟨delete⟩ from w;
12     if w ∉ Succ_u then begin
13       (* 実際にコネクションを削除する *)
14     end
15   end
```

Figure 10: Routing Algorithm (Part 5, for node u)