Program – 9

AIM: To implement Election between wireless nodes.

Introduction and Theory

Election Algorithms

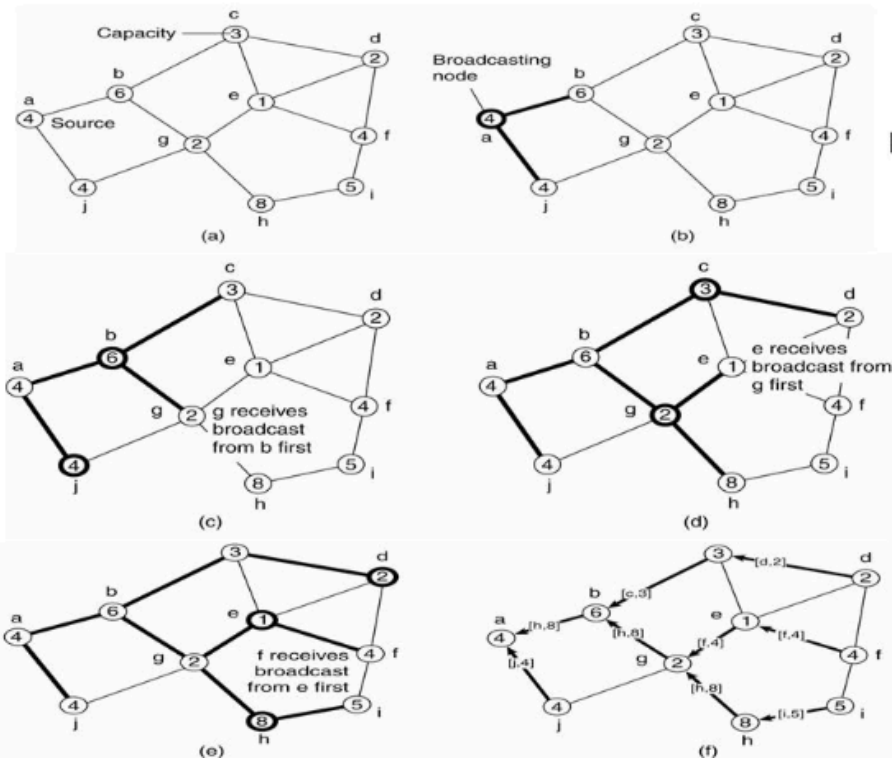
Election algorithms choose a process from group of processors to act as a coordinator. If the coordinator process crashes due to some reasons, then a new coordinator is elected on other processor. Election algorithm basically determines where a new copy of coordinator should be restarted. Election algorithm assumes that every active process in the system has a unique priority number. The process with highest priority will be chosen as a new coordinator. Hence, when a coordinator fails, this algorithm elects that active process which has highest priority number. Then, this number is sent to every active process in the distributed system.

Wireless Election process

1. Any node can initiate the election.
2. When a node receives its first ELECTION message, it makes the sender as its parent.
3. After this it forwards the ELECTION to all its neighbors.
4. If a node already has set its parent, it simply acknowledges.
5. If a node is a leaf it sends its own priority otherwise it waits for its children to finish.
6. When a node has collected all values, it passes it on to its parent.

Salient Points

- At each point only, the best possible candidate is passed.
- Once the source gets the results back it can select the coordinator, which it then broadcasts.
- The messages are tagged with process IDs and in case of multiple ELECTIONS, only the one from a higher pid is entertained.



Program – 9

Code

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>
#include <string.h>
#define MSG_CONFIRM 0


#define TRUE 1
#define FALSE 0
#define ML 1024
#define MPROC 32

typedef struct wireless_node
{
    int priority;
    int parent;
} wireless_node;

wireless_node w;

int max(int a, int b)
{
    return a >= b? a:b;
}

int connect_to_port(int connect_to)
{
    int sock_id;
    int opt = 1;
    struct sockaddr_in server;
    if ((sock_id = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
    {
        perror("unable to create a socket");
        exit(EXIT_FAILURE);
    }
    setsockopt(sock_id, SOL_SOCKET, SO_REUSEADDR, (const void
*)&opt, sizeof(int));
    memset(&server, 0, sizeof(server));
    server.sin_family = AF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(connect_to);

    if (bind(sock_id, (const struct sockaddr *)&server,
sizeof(server)) < 0)
    {
        perror("unable to bind to port");
        exit(EXIT_FAILURE);
```

Program – 9

```
55      }
56      return sock_id;
57  }
58
59  void send_to_id(int to, int from, char message[ML])
60  {
61      struct sockaddr_in cl;
62      memset(&cl, 0, sizeof(cl));
63      cl.sin_family = AF_INET;
64      cl.sin_addr.s_addr = INADDR_ANY;
65      cl.sin_port = htons(to);
66      sendto(
67          from, \
68          (const char *)message, \
69          strlen(message), \
70          MSG_CONFIRM, \
71          (const struct sockaddr *)&cl, \
72          sizeof(cl));
73  }
74
75  void startElection(int id, int *procs, int num_procs, int self)
76  {
77      int itr;
78      char message[ML];
79      sprintf(message, "%s %d", "ELEC", self);
80      for (itr = 0; itr < num_procs; itr++)
81      {
82          if (procs[itr] != w.parent)
83          {
84              printf("Sending elections to: %d\n", procs[itr]);
85              send_to_id(procs[itr], id, message);
86          }
87      }
88  }
89
90  void announce_completion(int self, int *procs, int num_procs, int
91  coord)
92  {
93      int itr;
94      char message[ML];
95      sprintf(message, "%s %d", "DONE", coord);
96
97      for (itr = 0; itr < num_procs; itr++)
98      {
99          send_to_id(procs[itr], self, message);
100     }
101 }
102
103 void propagate_completion(int self, int *procs, int num_procs, char
104 M[ML])
105 {
106     int itr;
107     for (itr = 0; itr < num_procs; itr++)
108     {
109         send_to_id(procs[itr], self, M);
110     }
111 }
```

Program – 9

```
112
113 int main(int argc, char* argv[])
114 {
115     int self = atoi(argv[1]);
116     int n_procs = atoi(argv[2]);
117     int procs[MPROC];
118     int sender, pcnt = 0, ecnt = 0;
119     int sock_id, coord_id;
120     int itr, len, n, start, ix;
121     char buffer[ML], flag[ML], p_id[ML], msg[256];
122     struct sockaddr_in from;
123     w.priority = atoi(argv[3]);
124     w.parent = -1;
125     coord_id = w.priority;
126     for(itr = 0; itr < n_procs; itr += 1)
127         procs[itr] = atoi(argv[4 + itr]);
128     start = atoi(argv[4 + n_procs]) == 1? TRUE:FALSE;
129
130     printf("Creating node at %d\n", self);
131     sock_id = connect_to_port(self);
132
133     if (start == TRUE)
134     {
135         startElection(sock_id, procs, n_procs, self);
136     }
137     while(TRUE)
138     {
139         if (start != TRUE && ecnt + 1 == n_procs)
140         {
141             sprintf(msg, "RTRN %d", coord_id);
142             send_to_id(w.parent, \
143             sock_id,
144             msg);
145             printf("Sending to parent %d\n", w.parent);
146         }
147         if (pcnt == n_procs)
148         {
149             if (start == TRUE)
150             {
151                 printf("Announcing completion\n");
152                 announce_completion(sock_id, procs, n_procs,
153 coord_id);
154                 exit(1);
155             }
156             else
157             {
158                 sprintf(msg, "RTRN %d", coord_id);
159                 send_to_id(w.parent, \
160                 sock_id,
161                 msg);
162                 printf("Sending to parent %d\n", w.parent);
163             }
164         }
165         memset(&from, 0, sizeof(from));
166         // printf("Tring read\n");
167         n = recvfrom(sock_id, (char *)buffer, ML, MSG_WAITALL,
168 (struct sockaddr *)&from, &len);
```

Program – 9

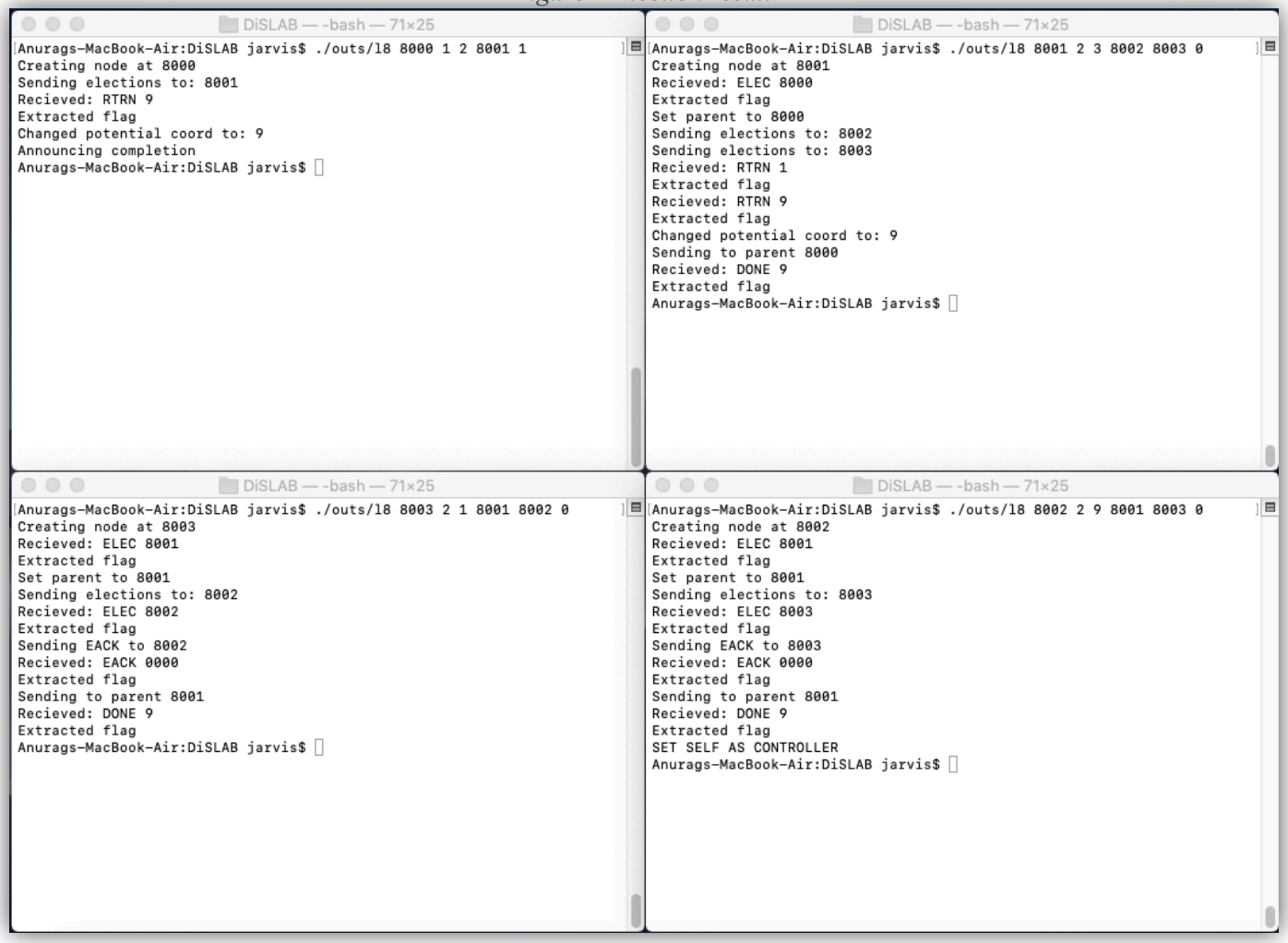
```
169             buffer[n] = '\0';
170             printf("Recieved: %s\n", buffer);
171             for(itr = 0; itr < 4; itr++)
172             {
173                 // printf("%c %d\n", buffer[itr], itr);
174                 flag[itr] = buffer[itr];
175             }
176             flag[itr] = '\0';
177             printf("Extracted flag \n");
178             if (strcmp(flag, "RTRN") == 0 || strcmp(flag, "DONE") == 0)
179             {
180                 for(ix=0, itr = itr + 1; itr < 6; itr++)
181                     p_id[ix++] = buffer[itr];
182             }
183             else
184             {
185                 for(ix=0, itr = itr + 1; itr < 9; itr++)
186                     p_id[ix++] = buffer[itr];
187             }
188             p_id[ix] = '\0';
189             sender = atoi(p_id);
190             // printf("%s %d\n", flag, sender);
191 
192             if (strcmp(flag, "ELEC") == 0)
193             {
194                 if (w.parent == -1)
195                 {
196                     w.parent = sender;
197                     printf("Set parent to %d\n", w.parent);
198                     if (n_procs == 1 && procs[0] == w.parent)
199                         pcnt ++;
200                     startElection(sock_id, procs, n_procs, self);
201                 }
202                 else
203                 {
204                     printf("Sending EACK to %d\n", sender);
205                     send_to_id(sender, sock_id, "EACK 0000");
206                 }
207             }
208             else if (strcmp(flag, "EACK") == 0)
209             {
210                 ecnt += 1;
211                 continue;
212             }
213             else if (strcmp(flag, "RTRN") == 0)
214             {
215                 pcnt += 1;
216                 if (w.priority < sender)
217                 {
218                     printf("Changed potential coord to: %d\n", sender);
219                     coord_id = sender;
220                 }
221                 else
222                     coord_id = max(coord_id, w.priority);
223             }
224             else if (strcmp(flag, "DONE") == 0)
225             {
```

Program – 9

```
226                 if (w.priority != sender)
227                     propagate_completion(sock_id, procs, n_procs,
228 buffer);
229                 else
230                     printf("SET SELF AS CONTROLLER\n");
231                 exit(1);
232             }
233             // printf("Waiting\n");
234         }
235         return 0;
236 }
237
```

Results and Outputs:

Figure 1 Election result



Findings and Learnings:

1. We successfully implemented Election in wireless networks.