SIMPLE NETWORK CONTROLLER
DESIGN AND IMPLEMENTATION

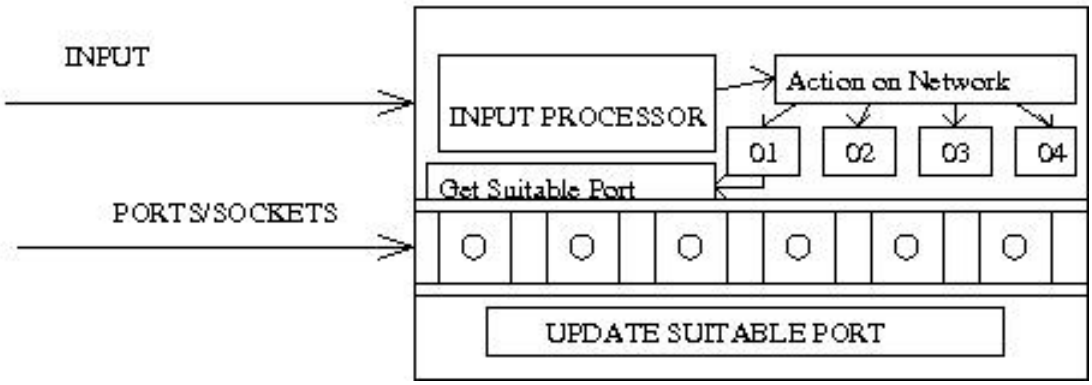
by Ayub <mrayub@gmail.com>
<ayub@technovilla.com>

Features:

- A very flexible system
- Type system is encapsulated from programming language types
- Unlimited number of networks can be controlled
- Unlimited number of ports can be in a controlled network. Although requirement was 6 ports
- Each component is encapsulated with its own properties in its own data type
- Load balancing on ports by a specialized technique. New process always gets attached on the port which has minimum number of processes attached
- Binary search tree has been used for process management and to make the program robust

Design:

01 => INIT
02 => COMP
03 => DOWN
04 => REST



Flow of control/Implementation: [snc.c, snc.h]

1. One line of input is read from file [or standard input].
2. Check what type of input it is. Available types are 'INIT', 'COMP', 'DOWN', 'REST'. 'INIT' for Initiating a process, 'COMP' for completion of a process, 'DOWN' for a port going down, 'REST' for a port restored.
3. When 'INIT' is found, the program reads in the PID and destination IP from input. 'action_on_network' function is called. It checks if any suitable port which is ACTIVE and not heavily loaded is found. The process is attached with this port if found. Otherwise it is an error when all ports are INACTIVE. A process entry is dynamically allocated and initialized. 'proc_init' function is called. 'proc_init' inserts the process entry to the process tree of the network and the process tree of the port which it is attached to.
4. When 'COMP' is found, the program reads in the PID. It then calls 'action_on_network', which in turn calls 'proc_comp'. 'proc_comp' finds the processentry from process tree of network. It then deletes the entry from process tree of network. Then it deletes the entry from process tree of the port where it was attached.
5. When 'DOWN' is found, the program reads in the PORT. It calls 'action_on_network' which calls 'port_down'. 'port_down' sets the status of the port as PORT_DOWN. It calls 'trigger_interrupt' for the process tree of this port, requesting to set their status PROC_SUSPENDED. 'trigger_interrupt' function sets status of all the processes of the given tree as PROC_SUSPENDED.
6. When 'REST' is found, it works the same as 'DOWN', just the difference is that it marks the port status to be PORT_UP and the processes in the process tree to be PROC_ACTIVE.
7. update_port_to_use() function is called after each input action. It updates the 'port_to_use' member of the network_t structure to be the less loaded port which is UP.
8. In 'snc.h', each component of the network has been encapsulated by creatingtype for individual component. '_t' is added after each “typedef'ed” types to make them separate from variable names. For example, 'tree_t' is a type 'struct tree { ... }'. I didn't impose any limit for the network. It is very flexible to be expanded to as many ports asneeded. That is what a good designer should design :)

Finally, I wish this document will help you understand the working procedure of the program. Please feel free to ask any details about any part of the program. When asking details, please specify filename and the line number range, or function name to describe. Tree functions are self-explanatory, so I didn't use any comments, only where I felt needed, I added comments. The rest of the code contains enough comments. The code listing is appended with this document.

Ayub <mrayub@gmail.com>
<ayub@technovilla.com>
Techno Villa [www.technovilla.com]

CODE LISTING

snc.h

```
/*
 * snc.h
 * header for snc.c
 */
#ifndef SNC_H
#define SNC_H

#include <stdio.h>              /* what to say about it ?*/
#include <string.h>
#include <stdlib.h>             /* calloc(), etc */

#define INET_ADDRSTRLEN 16

/* this is simulation project, so a very flexible type system is used */

typedef enum {
  false, true
} bool;

/* binary search tree types */
typedef void *element_t;

struct node;
typedef struct node node_t;
typedef node_t *position_t;
typedef node_t *tree_t;

struct node {
  element_t e;
  tree_t left;
  tree_t right;
};
```

```
/* status of the process */
#define PROC_ACTIVE       1
#define PROC_SUSPENDED          0
/* network related types */
typedef int pid_t; /* to encapsulate process identifier */
typedef int stat_t; /* status of process [active,suspended] */
struct proc_struct {
  pid_t pid;
  /* which address the process is communicating */
  char ip_dst[INET_ADDRSTRLEN];/* conforming to <netinet/in.h> */
  stat_t status; /* status of the process, to be triggered */
  int n_port; /* in which port this process is attached */
};
typedef struct proc_struct proc_t;

#define PORT_UP           1               /* port status */
#define PORT_DOWN         0               /* port status */
#define INVALID_PORT      -1              /* port number */
/* type for port */
struct port_struct {
  int n_port; /* port number [0 indexed] */
  stat_t status; /* status of the port */
  tree_t procs; /* list of processes on this port */
  int nprocs; /* how many processes currently attached with this port */
};
typedef struct port_struct port_t;
typedef port_t socket_t;

/* the whole network controller*/
#define NPORTS 6
#define INVALID_NETWORK -1
struct network_struct {
  int nid; /* network identifier, we may have several networks [0 indexed] */
  int nports; /* number of ports for the network controller */
  port_t *ports; /* the network has 'nports' ports, terminated by n_port=-1  */
  tree_t proc_tree; /* process tree for the network */
  /* for load balancing */
  int port_to_use; /* when new process arrives, attach it with this port */
};
typedef struct network_struct network_t;
```

```
/* types for input */

/* process initialization */
const char INIT[] = "INIT";             /*  process initiation  */
#define INIT_VAL 0
const char COMP[] = "COMP";                 /*  process completion  */
#define COMP_VAL 1
const char DOWN[] = "DOWN";                 /*  port is down        */
#define DOWN_VAL 2
const char REST[] = "REST";           /*  port is restored    */
#define REST_VAL 3
const char SHOW[] = "SHOW";                 /*  show network status */
#define SHOW_VAL 4
const char EXIT[] = "EXIT";           /*  exit from program   */
#define EXIT_VAL 4
#define BADT_VAL 5                          /*  bad type val        */
struct input_struct {
  char type[10]; /* type of input [one of the above] */
  int typeval;
  /* data for possible inputs */
  pid_t pid; /* process ID for initiation */
  char ip_dst[INET_ADDRSTRLEN];/* ip destination address for initiation */
  int n_port; /* which port is down/restored [0 indexed ]*/
};
typedef struct input_struct input_t;


/* public functions */
void run_simulator(void);
/* first obtain a network */
/* uses static variable, not thread-safe */
network_t *init_network(void);
/* take actoin on network according to input, return the result */
int action_on_network(network_t *net, input_t *input);
/* we're done, destroy the network */
void destroy_network(network_t *net);
void show_network_status(network_t *net);
void dump_proc_tree(tree_t t, bool verbose);
void print_proc_info(proc_t *proc, bool verbose);
void update_port_to_use(network_t *net);
```

```
/* for thread-safety, use the following routines */
void network_init(network_t *net, int nid, int nports); /* initialize network */
void network_free(network_t *net);
/* now manage the obtained network */
/* high level routine that calls other functions */

int proc_init(network_t *net, proc_t *proc); /* initialize process in net */
int proc_comp(network_t *net, pid_t pid); /* process completed */
int port_down(network_t *net, int n_port); /* port is down */
int port_rest(network_t *net, int n_port); /* port is restored*/
void trigger_interrupt(tree_t t, stat_t status); /* set status for all in t */
/*
 * macro equivalent of this function is available
 * int element_compare(element_t *e1, element_t *e2);
 */

/* ********* TREE FUNCTIONS ********* */
tree_t      tree_destroy(tree_t t, bool free_element);/* free element? */
position_t  tree_find(element_t e, tree_t t);
position_t  tree_find_min(tree_t t);
position_t  tree_find_max(tree_t t);
tree_t      tree_insert(element_t e, tree_t t);
tree_t      tree_delete(element_t e, tree_t t, bool free_element);
/*
 * macro equivalent of this function is available
 * element_t   tree_get_element(position_t p);
 */


/* these functions will be called several times, so using macros */
#define element_compare(e1, e2) (((proc_t*)(e1))->pid - ((proc_t *)(e2))->pid)
#define find_suitable_port(net) ((net)->port_to_use)
#define tree_get_element(p) ((p)->e)


#endif /* !SNC_H */
```

CODE LISTING

snc.c

```
/*
 * snc.c
 * simple-network-controller
 * Copyright (c) Ayub <mrayub@gmail.com><ayub@technovilla.com>
 * Techno Villa [www.technovilla.com]
 */
#include "snc.h"

int main(int argc, char **argv)
{
  if (argc > 1) { /* read input from file */
    freopen(argv[1], "r", stdin);
  }
  if (argc > 2) { /* write output to file */
    freopen(argv[2], "w", stdout);
  }

  run_simulator();

  return 0;
}


/* ================ ****** BLACK BOX ****** ================== */
void run_simulator(void)
{
  bool input_ok = false;
  char buf[80]; /* input buffer */

  input_t in;
  network_t *net;

  net = init_network();
```

```
/* input format:
 * <type> [other data according to type]
 */
while (fgets(buf, sizeof(buf), stdin)) {
  if (buf[0] == '#' || buf[0] == '\n') { /* ignore comment, empty line */
    continue; /* ignore */
  }
  memset(&in, 0, sizeof(input_t));
  input_ok = false;
  sscanf(buf, "%s", in.type);

  if (!strcmp(in.type, INIT)) {
    in.typeval = INIT_VAL;
    sscanf(buf, "%s%d%s", in.type, &in.pid, in.ip_dst);
    if (in.pid > 0 && strlen(in.ip_dst) > 0) {
        input_ok = true;
    } else {
        fprintf(stderr, "%s <PID> <IP> expected\n", INIT);
    }
  } else if (!strcmp(in.type,COMP)) {
    in.typeval = COMP_VAL;
    sscanf(buf, "%s%d", in.type, &in.pid);
    if (in.pid > 0) {
        input_ok = true;
    } else {
        fprintf(stderr, "%s <PID> expected\n", COMP);
    }
  } else if (!strcmp(in.type,DOWN)) {
    in.typeval = DOWN_VAL;
    sscanf(buf, "%s%d", in.type, &in.n_port);
    if (in.n_port > 0) {
        in.n_port--; /* 0 indexed */
        input_ok = true;
    } else {
        fprintf(stderr, "%s <#PORT> expected\n",DOWN);
    }
  } else if (!strcmp(in.type,REST)) {
    in.typeval = REST_VAL;
    sscanf(buf, "%s%d", in.type, &in.n_port);
    if (in.n_port > 0) {
```

```
            in.n_port--; /* 0 indexed */
            input_ok = true;
        } else {
            fprintf(stderr, "%s <#PORT> expected\n",REST);
        }
    } else if (!strcmp(in.type,SHOW)) {
        in.typeval = SHOW_VAL;
        show_network_status(net);
    } else if (!strcmp(in.type,EXIT)) {
        break;
    }
    else {
        in.typeval = BADT_VAL;
        fprintf(stderr, "Unknown type of input [%s]\n", in.type);
    }

    if (input_ok == true) {
        action_on_network(net, &in);
        update_port_to_use(net); /* update net->port_to_use variable */
    }
  }

  destroy_network(net);
}

/* 0: success
 * failure otherwise
 */
int action_on_network(network_t *net,input_t *input)
{
  proc_t *proc; /* for init_process */
  int ret = -1;
  int n_port = INVALID_PORT;

  switch (input->typeval) {
    case INIT_VAL:
      n_port = find_suitable_port(net); /* find up andless loaded one */
      if (n_port == INVALID_PORT) {
            fprintf(stderr, "ERROR: All ports are DOWN\n");
            break;
      }
```

```
      proc = malloc(sizeof(proc_t));
      if (!proc) {
            perror("malloc");
            exit(1); /* fatal error */
      }
      memset(proc, 0, sizeof(proc_t));
      proc->n_port = n_port;
      proc->pid = input->pid;
      strcpy(proc->ip_dst, input->ip_dst);
      proc->status = PROC_ACTIVE;
      ret = proc_init(net, proc);
      break;
    case COMP_VAL:
      ret = proc_comp(net, input->pid);
      break;
    case DOWN_VAL:
      ret = port_down(net, input->n_port);
      break;
    case REST_VAL:
      ret = port_rest(net, input->n_port);
      break;
    default:
      ret = -1;
      break;
  }

  return ret;
}

/* allocates a network and calls network_init() to initialize */
/*
 * NULL: failure
 * new network otherwise
 */
network_t *init_network(void)
{
  static int nid;

  network_t *temp;

  temp = (network_t *) malloc(sizeof(network_t));
```

```
  if (!temp) {
    perror("malloc");
    exit(1); /* fatal error */
  }

  memset(temp, 0, sizeof(network_t));
  network_init(temp, nid++, NPORTS);

  return temp;
}

void destroy_network(network_t *net)
{
  /* the network may be marely a variable which is not albcated, so we shoud
   * be confirm whether we really need to free the network itself
   * this is useful for multithreaded applications
   */
  network_free(net);
  free(net); /* it was allocated by 'calloc' */
}

/* initialize the given network, assuming 'net' is not initialized */
void network_init(network_t *net, int id, int nports)
{
  int i;

  if (net == NULL)
    return;

  net->nid = id;
  net->nports = nports;

  net->ports = (port_t *) malloc(sizeof(port_t) * nports);
  if (net->ports == NULL) {
    perror("malloc");/* program should exit immediately */
    exit(1); /* fatal error */
  }

  for (i = 0; i < net->nports; i++) {
    net->ports[i].n_port = i; /* initialize the port */
    net->ports[i].status = PORT_UP; /* initially all are up */
```

```
    net->ports[i].procs = NULL; /* no processes assigned initially */
    net->ports[i].nprocs = 0; /* no processes attached yet */
  }
  net->proc_tree = NULL; /* empty process tree */
}

/* always call 'false' first, then 'true' */
void network_free(network_t *net)
{
  int i;

  net->nid = INVALID_NETWORK;
  tree_destroy(net->proc_tree,false);
  for (i = 0; i < net->nports; i++) {
    tree_destroy(net->ports[i].procs,true); /* each proc has one instance */
  }
  free(net->ports);
}

/* initialize process in net */
/* 0: success
 * nonzero otherwise
 */
int proc_init(network_t *net, proc_t *proc)
{
  tree_t t;

  /* print_proc_info(proc, true); */

  t = net->proc_tree;
  t = tree_insert(proc, t);
  if (t) {
    net->proc_tree = t;
  } else {
    return -1;
  }

  t = net->ports[proc->n_port].procs;
  t = tree_insert(proc, t);
  if (t) {
    net->ports[proc->n_port].procs = t;
```

```
  } else {
    return -2;
  }
  net->ports[proc->n_port].nprocs++;

  return 0;
}

/*
 * 0: success
 * nonzero otherwise
 */
/* process completed */
int proc_comp(network_t *net, pid_t pid)
{
  tree_t t;
  position_t p;
  proc_t tmp_proc;
  proc_t *proc;
  int n;

  memset(&tmp_proc, 0, sizeof(proc_t));
  tmp_proc.pid = pid; /* pid is enough for finding, it's used for comparison */

  t = net->proc_tree;
  p = tree_find(&tmp_proc, t);
  if (!p) {
    return -1;
  }

  proc = (proc_t *) tree_get_element(p);
  /* we have the proc entry now */

  net->ports[proc->n_port].nprocs--;

  t = net->proc_tree;
  t = tree_delete(proc, t, false);
  net->proc_tree = t;

  n = proc->n_port; /* after freeing we can't access the element, so saving */
  t = net->ports[n].procs;
```

```
  t = tree_delete(proc, t, true);
  net->ports[n].procs = t;

  return 0;
}

/* port is down */
int port_down(network_t *net, int n_port)
{
  net->ports[n_port].status = PORT_DOWN;
  trigger_interrupt(net->ports[n_port].procs, PROC_SUSPENDED);

  return 0;
}

/* port is restored */
int port_rest(network_t *net, int n_port)
{
  net->ports[n_port].status = PORT_UP;
  trigger_interrupt(net->ports[n_port].procs, PROC_ACTIVE);

  return 0;
}

/* updates net->port_to_use variable */
void update_port_to_use(network_t *net)
{
  int minval, i;
  /* now update net->port_to_use for next process attachment */

  /* only probe which ports are UP, we can't attach to a DOWN port */
  minval = 0;
  net->port_to_use = INVALID_PORT;
  for (i = 0; i < net->nports; i++) {
    if (net->ports[i].status == PORT_UP) {
      minval = net->ports[i].nprocs;
      net->port_to_use = i;
      break;
    }
  }
```

```
  if (net->port_to_use == INVALID_PORT) { /* no port is UP */
    return;
  }

  for (i = 0; i < net->nports; i++) {
    if (net->ports[i].nprocs< minval && net->ports[i].status == PORT_UP) {
      minval = net->ports[i].nprocs;
      net->port_to_use = i;
    }
  }
}

void show_network_status(network_t *net)
{
  int i;
  bool verbose = false; /* we want verbose output ? */

  /*
  printf("*** Network Controller Status ***\n");
  printf("Network ID: %d\n", net->nid);
  printf("Number of ports: %d\n", net->nports);
  */

  for (i = 0; i < net->nports; i++) {
    printf("Port #%d: %s", i+1,
           net->ports[i].status == PORT_UP ? "Active" : "Inactive");
    if (net->ports[i].nprocs> 0) {
      printf(", Processes#");
    } else {
      printf(", No processes");
    }
    dump_proc_tree(net->ports[i].procs, verbose);
    printf("\n");
  }
  printf("... ... ...\n"); /* just for easy readability */
}

void dump_proc_tree(tree_t t, bool verbose)
{
  proc_t *proc;
```

```
  if (t) {
    dump_proc_tree(t->left, verbose);
    dump_proc_tree(t->right, verbose);
    proc = (proc_t *) t->e;
    print_proc_info(proc, verbose);
  }
}

void print_proc_info(proc_t *proc,bool verbose)
{
  if (verbose == false) {
    printf(" %d", proc->pid);
    /* fflush(stdout); */
  } else {
  printf("\tpid: %d\n"
      "\tip_dst: %s\n"
      "\tstatus: %d\n"
      "\tport: %d\n",
      proc->pid, proc->ip_dst, proc->status, proc->n_port);
  }
}

/* set status of all processes in the tree */
void trigger_interrupt(tree_t t, stat_t status)
{
  proc_t *proc;

  if (t) {
    trigger_interrupt(t->left,status);
    trigger_interrupt(t->right,status);
    proc = (proc_t *) t->e;
    proc->status = status;
  }
}

/* ============ TREE ================ */

/* binary search tree functions */
tree_t tree_destroy(tree_t t,bool free_element)
{
  if (t) {
```

```
    tree_destroy(t->left, free_element);
    tree_destroy(t->right, free_element);
    if (free_element == true) {
      free(t->e);
    }
    free(t);
  }
  return NULL;
}

position_t tree_find(element_t e, tree_t t)
{
  int result = 0;

  if (!t)
    return NULL;

  result = element_compare(e, t->e);
  if (result < 0)
    return tree_find(e, t->left);
  if (result > 0)
    return tree_find(e, t->right);

  /* else */
  return t;
}

position_t tree_find_min(tree_t t)
{
  if (!t)
    return NULL;

  if (!t->left)
    return t;

  return tree_find_min(t->left);
}

position_t tree_find_max(tree_t t)
{
  if (!t) {
```

```
    while (t->right) {
      t = t->right;
    }
  }

  return t;
}

tree_t tree_insert(element_t e, tree_t t)
{
  int result = 0;

  if (!t) {
    /* create one-node tree */
    t = malloc(sizeof(node_t));
    if (!t) {
      perror("malloc");
      exit(1); /* we can't progress, no memory available */
    } else {
      t->e = e;
      t->left = t->right = NULL;
    }
  } else {
    result = element_compare(e, t->e);
    if (result < 0) {
      t->left = tree_insert(e, t->left);
    } else if (result > 0) {
      t->right = tree_insert(e, t->right);
    } else {
      /* otherwise element is in the tree, do nothing, error */
      fprintf(stderr, "*** ERROR: Existing node\n");
      return NULL;
    }
  }

  return t;
}

tree_t tree_delete(element_t e, tree_t t, bool free_element)
{
  int result = 0;
```

```
  position_t tmp;

  if (!t) {
    fprintf(stderr, "Tree empty!\n");
  } else {
    result = element_compare(e, t->e);

    if (result < 0) { /* go left */
      t->left = tree_delete(e, t->left, free_element);
    } else if (result > 0) { /* go right */
      t->right = tree_delete(e, t->right, free_element);
    } else { /* found element */
      if (t->left && t->right) { /* both children exist */
          tmp = tree_find_min(t->right);
          t->e = tmp->e;
          t->right = tree_delete(t->e, t->right, free_element);
      } else { /* one or zero children */
          tmp = t;
          if (!t->left) {
            t = t->right;
          } else if (!t->right) {
            t = t->left;
          }
          if (free_element == true) {
            free(tmp->e);
          }
          free(tmp);
      }
    }
  }

  return t;
}

/* we have macro for these, so functions are not needed */
#ifdef __no__
element_t tree_get_element(position_t p)
{
  if (p) {
    return p->e;
  }
```

```
  return NULL;
}
#endif

#ifdef __no__
/* we have macro for this, so not using the function */
int find_suitable_port(network_t *net)
{
  return (net->port_to_use);
}
#endif

#ifdef __no__
/* we have macro for this, so not using function */
/* element_t == proc_t * */
int element_compare(element_t e1,element_t e2)
{
  proc_t *p1 = (proc_t *) e1;
  proc_t *p2 = (proc_t *) e2;

  return (p1->pid - p2->pid);
}
#endif
```