**DEC Design Document**

Chao Zhang

Email: [czhang28@buffalo.edu](mailto:czhang28@buffalo.edu)

**1. Data Structure:**

The data structure for the graph:

Node Structure:

typedef struct vertexStruct {

char event[12];

struct edgeStruct\* edges;

struct vertexStruct\* nodeNext;

int visited;

} Node;

Edge Structure:

typedef struct edgeStruct {

struct vertexStruct\* connectsTo;

struct edgeStruct\* edgeNext;

} Edge;

NodeList Structure:

typedef struct nodeListStruct {

struct vertexStruct\* nodeFront;

struct vertexStruct\* nodeRear;

} NodeList;

EdgeList Structure:

typedef struct edgeListStruct {

struct edgeStruct\* edgeFront;

struct edgeStruct\* edgeRear;

} EdgeList;



I use a linkedlist NodeList to store all the nodes. And use a linkedlist EdgeList to store the edges. Every node in the NodeList has a EdgeList. The EdgeList stores all the adjacent nodes. When executing an insert command, 2 nodes will be inserted into the NodeList, and then a directed edge will be added between these 2 nodes.

In order to find a path from 2 nodes, I use recursive DFS algorithm to find the path.

Input Command Analyze:

The program will analyze a whole line. Then it will split the line into words by space, every time it encounters a semicolon, it will treat the previous part as a complete command.

**2. Appendix 1: README**

#1: run: make. It will generate the execute file: dec\_server

#2: run: make client. It will generate the execute file: client

#3: run: ./dec\_server. Then run: ./client

**3. Appendix 2: Code**

**Client.c:**

#include <stdio.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <unistd.h>

#define perr(msg) { perror(msg); exit(1); }

int main(int argc, char \*argv[]) {

static const char help[] =

"\nDistributed Event Coordination System - Server:\n\n"

" -h - Print this usage summary with all options.\n\n"

" -s server-host - Connect to the specified host. The default server is localhost.\n\n"

" -p port-number - Connect to the server at the given port. The default port is 9090.\n\n";

int c;

int sockfd;

int len;

int done;

struct sockaddr\_in address;

int client;

int bytes;

int portNo = 9090;

char line[256];

char buffer[256];

char \*serverAddr = "127.0.0.1";

bzero(line, 256);

bzero(buffer, 256);

FILE\* file;

fd\_set ready;

opterr = 0;

while((c = getopt(argc, argv, "hs:p:")) != -1) {

switch(c) {

case 'h':

printf("%s", help);

exit(0);

break;

case 's':

serverAddr = optarg;

break;

case 'p':

portNo = atoi(optarg);

if (portNo == 0 && optarg != '0') {

perr("option -p argument type error! \n");

}

break;

case '?':

if (optopt == 's') {

perr("option -s argument type error! \n");

} else if (optopt == 'p') {

perr("option -p argument type error! \n");

}

return 1;

}

}

sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

address.sin\_family = AF\_INET;

address.sin\_addr.s\_addr = inet\_addr(serverAddr);

address.sin\_port = htons(portNo);

len = sizeof(address);

client = connect(sockfd, (struct sockaddr \*)&address, len);

if (client == -1) {

perror("client init error!\n");

exit(1);

}

file = stdin;

while(!done) {

/\*fgets(line, 256, file);\*/

FD\_ZERO(&ready);

FD\_SET(sockfd, &ready);

FD\_SET(fileno(stdin), &ready);

if (select((sockfd + 1), &ready, 0, 0, 0) < 0) {

perror("select");

exit(1);

}

if (FD\_ISSET(fileno(stdin), &ready)) {

if((bytes = read(fileno(stdin), line, 256)) <= 0) {

done++;

}

write(sockfd, line, 256);

}

if (FD\_ISSET(sockfd, &ready)) {

if ((bytes = read(sockfd, line, 256)) <= 0) {

done++;

}

write(fileno(stdout), line, 256);

}

bzero(line, sizeof(line));

}

close(client);

exit(0);

}

**Server.c:**

#include <stdio.h>

#include <strings.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <netdb.h>

#include <unistd.h>

#include <time.h>

#include <sys/stat.h>

#include <arpa/inet.h>

#include <dirent.h>

#include "server.h"

#include "io.h"

pthread\_mutex\_t mutex;

sem\_t sem;

char error[128];

char response[128];

/\*

\* start the server, and listen to the given port

\*/

int initServer(int port) {

int sock\_id;

struct sockaddr\_in saddr;

sock\_id = socket(AF\_INET, SOCK\_STREAM, 0);

if (sock\_id == -1 ) {

perr("Create socket error!");

}

bzero(&saddr, sizeof(saddr));

saddr.sin\_family = AF\_INET;

saddr.sin\_port = htons(port);

saddr.sin\_addr.s\_addr = INADDR\_ANY;

if (bind(sock\_id, (struct sockaddr \*) &saddr, sizeof(saddr)) != 0) {

perr("Server binding error!");

}

if (listen(sock\_id, 5)) {

perr("Server listen error!");

}

printf("Listening to the port: %d \n", port);

return sock\_id;

}

void handleRequest(void\* arguments) {

Args\* args = arguments;

int clientId = args->sock\_fd;

void\* nodeList = args->nodeList;

char\* path = args->logPath;

char\* clientName = args->clientName;

int logFlg = args->logFlg;

char buffer[128];

bzero(buffer, sizeof(buffer));

List\* cmdList[48];

int size;

int i;

FILE\* logFile;

while (read(clientId, buffer, sizeof(buffer))) {

if (logFlg == 1) {

logFile = fopen(path, "a+");

fprintf(logFile, "%s", buffer);

fclose(logFile);

}

bzero(error, sizeof(error));

bzero(response, sizeof(response));

memset(cmdList, 0, sizeof(cmdList));

size = AnalyzeInput(cmdList, buffer);

printf("size: %d\n", size);

if (size > 0) {

for (i = 0; i < size; i++) {

pthread\_mutex\_lock(&mutex);

ExecuteCmd(cmdList[i], nodeList, clientName);

/\*printf("traverse\n");\*/

/\*Traverse(nodeList);\*/

pthread\_mutex\_unlock(&mutex);

write(clientId, error, strlen(error));

write(clientId, response, strlen(response));

if (logFlg == 1) {

logFile = fopen(path, "a+");

fprintf(logFile, "%s", error);

fprintf(logFile, "%s", response);

fclose(logFile);

}

bzero(error, sizeof(error));

bzero(response, sizeof(response));

}

} else if (size == -1 || size == -3) {

write(clientId, error, strlen(error));

if (logFlg == 1) {

logFile = fopen(path, "a+");

fprintf(logFile, "%s", error);

fclose(logFile);

}

} else {

continue;

}

bzero(buffer, sizeof(buffer));

}

close(clientId);

}

**Server.h:**

#include <pthread.h>

#include <semaphore.h>

#define perr(msg) { perror(msg); exit(1); }

extern pthread\_mutex\_t mutex;

extern sem\_t sem;

/\* Individual job struct. Accept function, arguments and job size. \*/

typedef struct JobStruct {

void \*(\*function)(void \*arg);

void \*arg;

struct JobStruct \*next;

struct JobStruct \*prev;

}Job;

/\* Queue struct. \*/

typedef struct QueueStruct {

Job \*front;

Job \*rear;

int currentSize;

}Queue;

/\* Thread pool struct. \*/

typedef struct PoolStruct {

pthread\_t \*threads;

int threadsNum;

Queue \*queue;

}ThreadPool;

/\* Arguments struct. Store the args which will passed into every job. \*/

typedef struct ArgStruct {

int sock\_id;

int sock\_fd;

char\* logPath;

int logFlg;

void\* nodeList;

char\* clientName;

}Args;

/\* start the server \*/

int initServer();

void handleRequest(void \*);

**Graph.c:**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "graph.h"

Node\* InitNode() {

Node\* node = (Node \*)malloc(sizeof(Node));

memset(node, 0, sizeof(Node));

return node;

}

NodeList\* InitNodeList() {

NodeList\* nodeList = (NodeList \*)malloc(sizeof(NodeList));

memset(nodeList, 0, sizeof(NodeList));

return nodeList;

}

void AddNode(NodeList\* nodeList, char\* c) {

Node\* node = InitNode();

strcpy(node->event, c);

/\*node->event = c;\*/

if (nodeList->nodeFront == NULL) {

nodeList->nodeFront = nodeList->nodeRear = node;

} else {

nodeList->nodeRear->nodeNext = node;

nodeList->nodeRear = node;

}

}

/\* add edge to edgeList, two vertexes of the edge is edgeNode1 and edgeNode2\*/

void AddEdge(NodeList\* nodeList, char\* startChar, char\* endChar) {

Edge\* edge = (Edge \*)malloc(sizeof(Edge));

Node\* edgeNode1 = GetNode(nodeList, startChar);

Node\* edgeNode2 = GetNode(nodeList, endChar);

if (edgeNode1->edges == NULL) {

edgeNode1->edges = (Edge \*)malloc(sizeof(Edge));

edgeNode1->edges->connectsTo = edgeNode2;

} else {

edge = edgeNode1->edges;

while (edge->edgeNext != NULL) {

edge = edge->edgeNext;

}

edge->edgeNext = (Edge \*)malloc(sizeof(Edge));

edge->edgeNext->connectsTo = edgeNode2;

}

}

Node\* GetNode(NodeList\* nodeList, char\* c) {

Node\* node = nodeList->nodeFront;

for (; node != NULL; node = node->nodeNext) {

if (strcasecmp(node->event, c) == 0) {

return node;

}

}

return NULL;

}

int EdgeExists(NodeList\* nodeList, char\* startChar, char\* endChar) {

Node\* edgeStartNode = GetNode(nodeList, startChar);

Node\* edgeEndNode = GetNode(nodeList, endChar);

Edge\* edge = edgeStartNode->edges;

for(; edge != NULL; edge = edge->edgeNext) {

if (edge->connectsTo == edgeEndNode) {

return 1;

}

}

return 0;

}

int SubPathExists(Node\* startNode, Node\* endNode) {

if (startNode->visited == 1) {

return 0;

} else if (startNode == endNode) {

return 1;

} else {

startNode->visited = 1;

}

Edge\* edge = startNode->edges;

for(; edge != NULL; edge = edge->edgeNext) {

if (SubPathExists(edge->connectsTo, endNode)) {

return 1;

}

}

return 0;

}

int PathExists(NodeList\* nodeList, char\* startChar, char\* endChar) {

Node\* node = nodeList->nodeFront;

Node\* startNode = GetNode(nodeList, startChar);

Node\* endNode = GetNode(nodeList, endChar);

for(; node != NULL; node = node->nodeNext) {

node->visited = 0;

}

return SubPathExists(startNode, endNode);

}

int IsConflicted(NodeList\* nodeList, char\* startChar, char\* endChar) {

if (PathExists(nodeList, endChar, startChar) == 1) {

return 1;

}

return 0;

}

void Traverse(NodeList\* nodeList) {

Node\* node = nodeList->nodeFront;

printf("step1\n");

Edge\* edge;

while (1) {

if (node != NULL) {

printf("%s: ", node->event);

edge = node->edges;

while (1) {

if (edge != NULL) {

printf("%s, ", edge->connectsTo->event);

edge = edge->edgeNext;

} else {

printf("\n");

break;

}

}

node = node->nodeNext;

} else {

break;

}

}

}

**Graph.h:**

typedef struct vertexStruct {

char event[12];

struct edgeStruct\* edges;

struct vertexStruct\* nodeNext;

int visited;

} Node;

typedef struct edgeStruct {

struct vertexStruct\* connectsTo;

struct edgeStruct\* edgeNext;

} Edge;

typedef struct nodeListStruct {

struct vertexStruct\* nodeFront;

struct vertexStruct\* nodeRear;

} NodeList;

typedef struct edgeListStruct {

struct edgeStruct\* edgeFront;

struct edgeStruct\* edgeRear;

} EdgeList;

Node\* InitNode();

NodeList\* InitNodeList();

void AddNode(NodeList\*, char\*);

void AddEdge(NodeList\*, char\*, char\*);

Node\* GetNode(NodeList\*, char\*);

int EdgeExists(NodeList\* nodeList, char\*, char\*);

int SubPathExists(Node\*, Node\*);

int PathExists(NodeList\*, char\*, char\*);

int IsConflicted(NodeList\*, char\*, char\*);

void Traverse(NodeList\*);

**io.c:**

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <ctype.h>

#include "io.h"

/\*#include "graph.h"\*/

/\*#include "server.h"\*/

void AddCmd(List\* list, char\* str) {

Cmd\* cmd = (Cmd \*)malloc(sizeof(Cmd));

memset(cmd, 0, sizeof(Cmd));

strcpy(cmd->val, str);

if (list->front == NULL) {

list->front = cmd;

list->rear = cmd;

} else {

list->rear->next = cmd;

list->rear = cmd;

}

}

int AnalyzeInput(List\* cmdList[], char\* input) {

/\*FILE\* file = stdin;\*/

char line[128];

char\* word;

char\* tmp;

char semicolonWord[12];

bzero(line, sizeof(line));

strcpy(line, input);

/\*fgets(line, 128, file);\*/

if (strlen(line) == 0) {

return -2;

}

if (strcmp(line, "\n") == 0) {

printf("please input the command.\n");

strcat(error, "please input the command.\n");

return -3;

}

/\* The input command don't end with ";" \*/

if (line[strlen(line) - 2] != ';') {

printf("input should end with \";\" \n");

strcat(error, "response from server: input should end with \";\" \n");

return -1;

}

word = strtok(line, " \t\n");

int i = 0;

while (word != NULL) {

if (cmdList[i] == NULL) {

cmdList[i] = (List \*)malloc(sizeof(List));

memset(cmdList[i], 0, sizeof(List));

}

tmp = strchr(word, ';');

if (tmp != NULL) {

strncpy(semicolonWord, word, tmp-word);

semicolonWord[tmp-word] = '\0';

AddCmd(cmdList[i], semicolonWord);

i++;

} else {

AddCmd(cmdList[i], word);

}

word = strtok(NULL, " \t\n");

}

/\*fclose(file);\*/

return i;

}

int IsLegalInsertCmd(Cmd\* cmd) {

if (strlen(cmd->val) != 4) {

return 0;

}

if (!isalpha(cmd->val[0]) || !isalpha(cmd->val[3])) {

return 0;

}

if (cmd->val[1] != '-' || cmd->val[2] != '>') {

return 0;

}

return 1;

}

int IsLegalQueryCmd(Cmd\* cmd) {

if (strlen(cmd->val) != 1) {

return 0;

}

if (!isalpha(cmd->val[0])) {

return 0;

}

return 1;

}

void InsertNode(List\* list, NodeList\* nodeList) {

/\* skip the fisrt command: insert \*/

Cmd\* cmd = list->front->next;

char startChar[2];

char endChar[2];

bzero(startChar, sizeof(startChar));

bzero(endChar, sizeof(endChar));

/\* check if there is any illegal input, if exists, terminate the whole command \*/

for (; cmd != NULL; cmd = cmd->next) {

if (!IsLegalInsertCmd(cmd)) {

printf("insert command input error.\n");

strcat(error, "response from server: insert command input error.\n");

}

}

/\* reset the cmd pointer to the list->front->next; \*/

cmd = list->front->next;

/\*printf("print nodelist backup:\n");\*/

/\*Traverse(nodeListBackup);\*/

for (; cmd != NULL; cmd = cmd->next) {

strncpy(startChar, &cmd->val[0], 1);

strncpy(endChar, &cmd->val[3], 1);

if ((GetNode(nodeList, startChar) != NULL) && (GetNode(nodeList, endChar) != NULL)) {

if (IsConflicted(nodeList, startChar, endChar)) {

printf("CONFLICT DETECTED. INSERT FAILED.\n");

printf("%s->%s and %s->%s cannot be true at the same time!\n", startChar, endChar, endChar, startChar);

strcat(error, "response from server: CONFLICT DETECTED. INSERT FAILED.\n");

strcat(error, startChar);

strcat(error, "->");

strcat(error, endChar);

strcat(error, " and ");

strcat(error, endChar);

strcat(error, "->");

strcat(error, startChar);

strcat(error, " cannot be true at the same time!\n");

return;

}

} else {

if (GetNode(nodeList, startChar) == NULL) {

AddNode(nodeList, startChar);

}

if (GetNode(nodeList, endChar) == NULL) {

AddNode(nodeList, endChar);

}

if (!PathExists(nodeList, startChar, endChar)) {

AddEdge(nodeList, startChar, endChar);

}

}

}

printf("INSERT DONE\n");

strcat(response, "response from server: INSERT DONE\n");

}

void QueryNode(List\* list, NodeList\* nodeList) {

/\* skip the fisrt command: query \*/

Cmd\* cmd = list->front->next;

char startChar[2];

char endChar[2];

bzero(startChar, sizeof(startChar));

bzero(endChar, sizeof(endChar));

for (; cmd != NULL; cmd = cmd->next) {

if (IsLegalQueryCmd(cmd) == 0) {

strcat(error, "response from server: query command input error.\n");

}

}

cmd = list->front->next;

Cmd\* cmd2 = cmd->next;

strncpy(startChar, &cmd->val[0], 1);

strncpy(endChar, &cmd2->val[0], 1);

if (GetNode(nodeList, startChar) == NULL) {

printf("Event not found: %s.\n", startChar);

strcat(error, "response from server: Event not found: ");

strcat(error, startChar);

strcat(error, ".\n");

}

if (GetNode(nodeList, endChar) == NULL) {

printf("Event not found: %s.\n", endChar);

strcat(error, "response from server: Event not found: ");

strcat(error, endChar);

strcat(error, ".\n");

return;

}

if (PathExists(nodeList, startChar, endChar)) {

printf("%s happened before %s.\n", startChar, endChar);

strcat(response, "response from server: ");

strcat(response, startChar);

strcat(response, " happened before ");

strcat(response, endChar);

strcat(response, ".\n");

} else if (PathExists(nodeList, endChar, startChar)) {

printf("response from server: %s happened before %s.\n", endChar, startChar);

strcat(response, "response from server: ");

strcat(response, endChar);

strcat(response, " happened before ");

strcat(response, startChar);

strcat(response, ".\n");

} else {

printf("response from server: %s concurrent to %s.\n", startChar, endChar);

strcat(response, "response from server: ");

strcat(response, startChar);

strcat(response, " concurrent to ");

strcat(response, endChar);

strcat(response, ".\n");

}

}

void ResetNodeList(NodeList\* nodeList) {

memset(nodeList, 0, sizeof(nodeList));

printf("response from server: RESET DONE.\n");

strcat(response, "RESET DONE.\n");

}

void ExecuteCmd(List\* list, NodeList\* nodeList, char\* clientName) {

printf("request received from %s: ", clientName);

Cmd\* cmd = list->front;

for(; cmd != NULL; cmd = cmd->next) {

printf("%s ", cmd->val);

}

printf("\n");

cmd = list->front;

if (strcmp(cmd->val, "insert") == 0) {

InsertNode(list, nodeList);

} else if (strcmp(cmd->val, "query") == 0) {

QueryNode(list, nodeList);

} else if (strcmp(cmd->val, "reset") == 0) {

ResetNodeList(nodeList);

} else {

printf("input error!\n");

strcat(error, "input error!\n");

}

}

**io.h:**

#include "graph.h"

extern char error[128];

extern char response[128];

typedef struct listStruct {

struct cmdStruct\* front;

struct cmdStruct\* rear;

} List;

typedef struct cmdStruct {

char val[12];

struct cmdStruct\* next;

} Cmd;

void AddCmd(List \*, char \*);

int AnalyzeInput(List\* [], char \*);

int IsLegalInsertCmd(Cmd \*);

int IsLegalQueryCmd(Cmd \*);

void InsertNode(List \*, NodeList \*);

void QueryNode(List \*, NodeList \*);

void ResetNodeList(NodeList \*);

void ExecuteCmde(List \*, NodeList \*, char \*);

**main.c:**

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

#include <string.h>

#include <unistd.h>

#include <netinet/in.h>

#include <sys/types.h>

#include <sys/socket.h>

#include <sys/stat.h>

#include "server.h"

#include "io.h"

int main(int argc, char \*argv[]) {

static const char help[] =

"\nDistributed Event Coordination System - Server:\n\n"

" -h - Print this usage summary with all options.\n\n"

" -p port-number - Listen on the given port. The default port is 9090.\n\n"

" -l file - Log all requests and responses to the given file. If not, print all to stdout.\n\n";

int c;

int sock\_id, sock\_fd;

int portNo = 9090;

pthread\_t threads[20];

int logFlg = 0;

char\* path = NULL;

int threadsCount = 0;

opterr = 0;

while((c = getopt(argc, argv, "hp:l:")) != -1) {

switch(c) {

case 'h':

printf("%s", help);

exit(0);

break;

case 'p':

portNo = atoi(optarg);

if (portNo == 0 && optarg != '0') {

perr("option -p argument type error! \n");

}

break;

case 'l':

logFlg = 1;

path = optarg;

break;

case '?':

if (optopt == 'l') {

perr("option -l needs an argument. \n");

} else if (optopt == 'p') {

perr("option -p needs an argument. \n");

}

return 1;

}

}

pthread\_mutex\_init(&mutex, NULL);

NodeList\* nodeList = InitNodeList();

sock\_id = initServer(portNo);

while(1) {

struct sockaddr\_in caddr;

socklen\_t clen = sizeof(caddr);

bzero(&caddr, sizeof(caddr));

Args\* args;

char clientName[1024];

args = (Args \*)malloc(sizeof(Args));

memset(args, 0, sizeof(args));

sock\_fd = accept(sock\_id, (struct sockaddr \*) &caddr, &clen);

if( sock\_fd == -1) {

perr("Server accept error!");

}

getnameinfo(&caddr, clen, clientName, sizeof(clientName), NULL, 0, 0);

/\*printf("client name: %s\n", clientName);\*/

args->sock\_fd = sock\_fd;

args->nodeList = nodeList;

args->logFlg = logFlg;

args->logPath = path;

args->clientName = clientName;

pthread\_create(&threads[threadsCount], NULL, (void \*) handleRequest, (void \*) args);

threadsCount++;

}

int i;

for (i = 0; i < threadsCount; i++) {

pthread\_join(threads[i], NULL);

}

close(sock\_fd);

close(sock\_id);

return 0;

}