Kalpesh Kagresha

UBIT: kkagresh

Person#: 50061160

CSE 421/521: Project 3 Report

Programming Task: To Implement a Distributed Event Coordination System

The objective of this project is to implement a distributed event coordination system (DEC) system in C/C++ on a UNIX-based platform. This system will consist of a DEC server which can interact and communicate with multiple DEC clients at the same time.

1. Important Components

a. dec_server: The main objective of the server to listen on a certain port number for incoming requests from DEC clients.
b. dec_client: It is responsible for sending the following commands such as insert, query and reset.

2. Data Structure Used

In order to maintain event ordering system, directed acyclic graph is required. To implement this, a sophisticated data structure such as adjacency list is used.

The following data structure is used at the server side implementation.

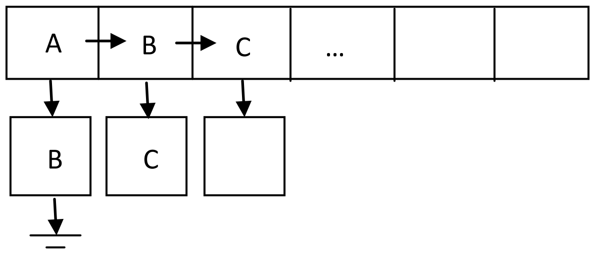
```
typedef struct vertex{
        char element;
        struct vertex *connVertices;
}vertex;

typedef struct graph{
        char element;
        struct graph *next;
        struct vertex *vertices;
}graph;
```

2.a struct graph: This structure stores nodes/vertices in graph. Each node is a vertex linked by 'next' pointer. List of connected vertices to this node is linked by the 'vertices' pointer.

2.b struct vertex: This structure stores the vertices connected to node in graph. It acts as linked list of all vertices connected to a given node in graph.

3. Visual Representation of Directed Acyclic Graph (Using Adjacency List)



The above figure illustrates the following idea:

1. Horizontal Linked List represents the nodes/ vertices of graph while vertical linked list highlights the vertices connected to a particular node.

2. The node in horizontal list contains the pointer 'vertices' in 'graph' struct pointing the first vertex, therefore all vertices are connected through 'connVertices' in struct 'vertex'.

3. For e.g. as shown A->B, B->C are edges from A to B and B to C respectively

4. Methods/ Functions Used at Server side

1. **void initConfigParams(char *argv[]):** This method takes as input the arguments from command line and initializes the parameters such as usageFlag, log_file and portno.

2. **int checkPath(graph **g, char source, char dest):** It takes as input the double pointer to graph, source and destination vertex. It is a recursive function and its main objective to detect path from source to destination. Basically we start from source and traverse all the connected nodes. For each path/edge, we make a recursive call by passing current vertex and same destination. If there exists a path, it returns 1 else 0;

3. **int undoInsert(graph**g,char event1,char event2):** This method takes input two events namely event1 and event2. It removes event2 from list of event1 i.e. it removes connected vertex from node event1. This situation occurs in case if there is conflict while inserting events in graph (For e.g. events A->B B->C are already present in graph and inserting C->A forms a cycle which is not allowed).

4. **int removeUnRefNodes(graph **g):** This method helps to remove all unreferenced nodes from the graph. These nodes exists when we remove all its connected vertices and no other node is connected this node.

5. **int addVertex1(graph**g, char event1, char event2):** This function is heart of the system adding vertex and edges in graph. It is also responsible for detecting the cycle in graph. Cycle is detected by passing the input string C->A in reverse order i.e. we check if there exists path from A to C. If yes, method returns appropriate message and stops the further addition of events in graph.

6. **char * query(graph **g, char event1, char event2):** This method finds relationship between two events event1 and event2. There are three possible cases i.e. 'Event not found', 'happened before' and 'concurrent events'.

7. **void reset(graph **g):** Resets the graph by de-allocating the memory.

8. **char * parseInput(char *input, graph **g)**: Another important method which scans the input from left to right to check for commands (such as insert, query and reset), syntax(

for e.g. insert A->B; should contain event1 followed by ->, Event2 and ;) and illegal characters (such as numbers etc). After parsing the input string, it calls addVertex1() method to insert the events in the graph. In case of conflict, we make a call to undoInsert() and removeUnRefNodes(). Depending on the command (insert, query or reset), a call is made to appropriate methods defined above.

5. Working:

The server starts its execution by initializing the configuration parameters. Thereafter, it creates a socket, binds the socket with server address and portno. It then starts listening on this socket for incoming requests. It continuously accepts the connection in while loop. After reading incoming request i.e. insert, query and reset command from the client, it parses the input string for any syntax errors. If found, returns the appropriate message to client. If parsed successfully, it calls methods such as addVertex, query and reset depending on the command in input string. In case of conflict, client is notified and further processing of input string is stopped.

On the other side, client also initializes the config parameters such as usage summary, server IP address and portno. Similarly to server, it creates socket and connects to the server. After establishing connection, it starts sending input string and waits for the response.

Conclusion:

The project is tested and executed successfully with different inputs.