**Topic 1: Performance Study of CSMA/CD Protocol through Simulation**

Algorithm:

1. Initialize the network environment and configure the nodes.

2. Set the simulation parameters such as node density, traffic load, and network conditions.

3. Begin the simulation:

a. For each node, check the channel for any ongoing transmission (Carrier Sense).

b. If the channel is idle, the node transmits the frame.

c. Monitor the channel for collisions during transmission (Collision Detection).

d. If a collision occurs, the node waits for a random backoff period before retransmitting.

e. Measure the throughput, latency, and collision rate during the simulation.

4. Analyze the simulation results and calculate performance metrics.

5. Repeat the simulation with different parameters and configurations to observe their impact on performance.

Code (C++):

#include <iostream>

#include <random>

using namespace std;

void simulateCSMACD(int nodeCount, int simulationTime) {

std::random\_device rd;

std::default\_random\_engine generator(rd());

std::uniform\_real\_distribution<double> distribution(0.0, 1.0);

int collisions = 0;

int successfulTransmissions = 0;

for (int time = 0; time < simulationTime; time++) {

bool channelBusy = (distribution(generator) < 0.2

for (int node = 0; node < nodeCount; node++) {

if (!channelBusy) {

successfulTransmissions++;

bool collisionDetected = (distribution(generator) < 0.1);

if (collisionDetected) {

collisions++;

}

}

}

}

double throughput = successfulTransmissions / (double)simulationTime;

double collisionRate = collisions / (double)successfulTransmissions;

cout << "Simulation Results:\n";

cout << "Successful Transmissions: " << successfulTransmissions << endl;

cout << "Collision Rate: " << collisionRate << endl;

cout << "Throughput: " << throughput << endl;

}

int main() {

int nodeCount = 10;

int simulationTime = 1000;

simulateCSMACD(nodeCount, simulationTime);

return 0;

}

**Topic 2: Performance Study of Token Bus and Token Ring Protocols through Simulation**

Algorithm:

1. Initialize the network topology and configure nodes for the desired protocol (Token Bus or Token Ring).

2. Set simulation parameters such as node density, traffic load, and network conditions.

3. Begin the simulation:

a. Simulate token circulation in the network according to the protocol's rules.

b. Nodes transmit data only when they possess the token.

c. Measure the throughput, latency, and network utilization during the simulation.

4. Analyze the simulation results and calculate performance metrics.

5. Repeat the simulation with different parameters and configurations to observe their impact on performance.

Code (C++):

#include <iostream>

#include <vector>

using namespace std;

void simulateTokenBus(int nodeCount, int simulationTime) {

std::vector<bool> tokens(nodeCount, true);

int successfulTransmissions = 0;

for (int time = 0; time < simulationTime; time++) {

for (int node = 0; node < nodeCount; node++) {

if (tokens[node]) {

successfulTransmissions++;

}

tokens[node] = false;

}

}

double throughput = successfulTransmissions / (double)simulationTime;

cout << "Simulation Results:\n";

cout << "Successful Transmissions: " << successfulTransmissions << endl;

cout << "Throughput: " << throughput << endl;

}

int main() {

int nodeCount = 10;

int simulationTime = 1000

simulateTokenBus(nodeCount, simulationTime);

return 0;

}

**Topic 3: Implementation of Error Detection and Correction Algorithms**

Algorithm (Checksum):

1. Take input data to be transmitted.

2. Calculate the checksum by summing all the data bytes.

3. Append the checksum to the data.

4. Transmit the data with the checksum.

5. On the receiver side, extract the received data and checksum.

6. Calculate the checksum of the received data.

7. Compare the calculated checksum with the received checksum.

8. If they match, the data is error-free. Otherwise, an error occurred during transmission.

Code (C++):

#include <iostream>

#include <vector>

using namespace std;

int calculateChecksum(const std::vector<int>& data) {

int sum = 0;

for (int byte : data) {

sum += byte;

}

return sum;

}

bool detectErrorsWithChecksum(const std::vector<int>& data, int checksum) {

int calculatedChecksum = calculateChecksum(data);

return (calculatedChecksum == checksum);

}

int main() {

vector<int> data = { 1, 2, 3, 4, 5 };

int checksum = calculateChecksum(data);

cout << "Data: ";

for (int byte : data) {

cout << byte << " ";

}

cout << endl;

cout << "Checksum: " << checksum << endl;

bool isErrorDetected = detectErrorsWithChecksum(data, checksum);

if (isErrorDetected) {

cout << "Error detected during transmission." << endl;

} else {

cout << "No errors detected during transmission." << endl;

}

return 0;

}

**Topic 4: Implementation and Study of 1-Bit Sliding Window (Stop and Wait) Protocol**

Algorithm:

1. Initialize the sender and receiver.

2. Sender:

a. Read data to be transmitted.

b. Send a frame with the data to the receiver.

c. Start a timer to wait for an acknowledgment.

d. If the acknowledgment is received within the timeout period, move to the next frame.

e. If the acknowledgment is not received within the timeout period, retransmit the frame.

f. Repeat steps b to e until all frames are transmitted.

3. Receiver:

a. Receive a frame from the sender.

b. Send an acknowledgment back to the sender.

c. Process the received data.

d. Repeat steps a to c for each frame received.

Code (C++):

#include <iostream>

using namespace std;

void sender() {

int frameCount = 5;

for (int frame = 0; frame < frameCount; frame++) {

cout << "Sending Frame " << frame << endl;

bool isAcknowledged = false;

while (!isAcknowledged) {

isAcknowledged = true;

}

}

}

void receiver() {

int frameCount = 5;

for (int frame = 0; frame < frameCount; frame++) {

cout << "Sending Acknowledgment for Frame " << frame << endl;

}

}

int main() {

sender();

receiver();

return 0;

}

**Topic 5: Implementation and Study of Go-Back-N Protocol**

Algorithm:

1. Initialize the sender and receiver.

2. Sender:

a. Divide the data into fixed-size frames.

b. Send multiple frames, one after another, without waiting for individual acknowledgments.

c. Maintain a sliding window to keep track of sent frames and acknowledgments received.

d. Start a timer for the first frame in the window.

e. If an acknowledgment is received within the timeout period, move the window forward.

f. If a timeout occurs, retransmit all frames in the window.

g. Repeat steps b to f until all frames are transmitted.

3. Receiver:

a. Receive frames from the sender.

b. Send cumulative acknowledgments for received frames.

c. Process the received data.

d. Repeat steps a to c for each frame received.

Code (C++):

#include <iostream>

#include <vector>

using namespace std;

void sender() {

int frameCount = 10;

int windowSize = 4;

vector<bool> acknowledgments(frameCount, false);

int nextFrameToSend = 0;

int base = 0;

while (base < frameCount) {

for (int i = base; i < base + windowSize && i < frameCount; i++) {

cout << "Sending Frame " << i << endl;

}

for (int i = base; i < base + windowSize && i < frameCount; i++) {

acknowledgments[i] = true;

}

while (base < frameCount && acknowledgments[base]) {

base++;

}

}

}

void receiver() {

int frameCount = 10;

for (int i = 0; i < frameCount; i++) {

cout << "Sending Acknowledgment for Frame " << i << endl;

}

}

int main() {

sender();

receiver();

return 0;

}

**Topic 6: Implementation and Study of Selective Repeat Protocol**

Algorithm:

1. Initialize the sender and receiver.

2. Sender:

a. Divide the data into fixed-size frames.

b. Send multiple frames, one after another, without waiting for individual acknowledgments.

c. Maintain a sending window to keep track of sent frames and acknowledgments received.

d. Start a timer for each frame in the window.

e. If an acknowledgment is received within the timeout period, mark the frame as acknowledged.

f. If a timeout occurs for a specific frame, retransmit only that frame.

g. Repeat steps b to f until all frames are transmitted.

3. Receiver:

a. Receive frames from the sender.

b. Send individual acknowledgments for received frames.

c. Process the received data.

d. Repeat steps a to c for each frame received.

Code (C++):

#include <iostream>

#include <vector>

using namespace std;

void sender() {

int frameCount = 10;

int windowSize = 4;

vector<bool> acknowledgments(frameCount, false);

int base = 0;

int nextFrameToSend = 0;

while (base < frameCount) {

for (int i = base; i < base + windowSize && i < frameCount; i++) {

cout << "Sending Frame " << i << endl;

}

for (int i = base; i < base + windowSize && i < frameCount; i++) {

acknowledgments[i] = true;

}

while (base < frameCount && acknowledgments[base]) {

base++;

}

}

}

void receiver() {

int frameCount = 10;

for (int i = 0; i < frameCount; i++) {

cout << "Sending Acknowledgment for Frame " << i << endl;

}

}

int main() {

sender();

receiver();

return 0;

}

**Topic 7: Get the MAC or Physical Address of system using Address Resolution Protocol**Algorithm:

* Start the program.
* Execute the command "arp -a" to retrieve the system's ARP table.
* Read and capture the output of the command.
* Parse the command output to extract the MAC or physical address.
* Print the MAC or physical address.
* End the program

Code (C++):  
#include <iostream>

#include <string>

#include <vector>

#include <fstream>

#include <sstream>

#include <cstdlib>

using namespace std;

string executeCommand(const string& command) {

array<char, 128> buffer;

string result;

shared\_ptr<FILE> pipe(popen(command.c\_str(), "r"), pclose);

if (!pipe) {

throw runtime\_error("Failed to run command.");

}

while (!feof(pipe.get())) {

if (fgets(buffer.data(), 128, pipe.get()) != nullptr) {

result += buffer.data();

}

}

return result;

}

string extractMACAddress(const string& arpOutput) {

istringstream iss(arpOutput);

string line;

vector<string> tokens;

while (getline(iss, line)) {

istringstream issLine(line);

string token;

while (getline(issLine, token, ' ')) {

tokens.push\_back(token);

}

}

for (const std::string& token : tokens) {

if (token.find(':') != std::string::npos) {

return token;

}

}

return "";

}

int main() {

string arpCommand = "arp -a";

string arpOutput = executeCommand(arpCommand);

string macAddress = extractMACAddress(arpOutput);

if (!macAddress.empty()) {

cout << "MAC Address: " << macAddress << endl;

} else {

cout << "Failed to retrieve MAC address." << endl;

}

return 0;

}  
  
**Topic 8: Implementation of distance vector routing algorithm**Algorithm:

1. Start the program.
2. Define the network topology using an adjacency matrix, representing the connections and distances between nodes.
3. Initialize a distance vector routing table with the same dimensions as the adjacency matrix. Set all distances to infinity except for the distances between directly connected nodes, which are set to their corresponding values in the adjacency matrix.
4. Perform the distance vector routing algorithm:
   1. Iterate over all nodes in the network.
   2. For each node, iterate over all possible destinations.
   3. For each destination, iterate over all neighboring nodes of the current node.
   4. Check if the distance from the current node to the destination, passing through the neighboring node, is shorter than the current known distance.
   5. If the new distance is shorter, update the distance in the distance vector routing table.
5. Repeat step 4 until there are no further changes to the distance vector routing table.
6. Print the final distance vector routing table, showing the shortest distances between nodes.
7. End the program.

Code (C++):#include <iostream>

#include <vector>

using namespace std;

vector<vector<int>> calculateDistanceVectorRouting(const vector<vector<int>>& adjacencyMatrix) {

int numNodes = adjacencyMatrix.size();

vector<vector<int>> distanceVectorRoutingTable(numNodes, vector<int>(numNodes, INT\_MAX));

for (int i = 0; i < numNodes; ++i) {

for (int j = 0; j < numNodes; ++j) {

if (adjacencyMatrix[i][j] != 0) {

distanceVectorRoutingTable[i][j] = adjacencyMatrix[i][j];

}

}

}

for (int k = 0; k < numNodes; ++k) {

for (int i = 0; i < numNodes; ++i) {

for (int j = 0; j < numNodes; ++j) {

if (distanceVectorRoutingTable[i][k] != INT\_MAX && distanceVectorRoutingTable[k][j] != INT\_MAX) {

int newDistance = distanceVectorRoutingTable[i][k] + distanceVectorRoutingTable[k][j];

if (newDistance < distanceVectorRoutingTable[i][j]) {

distanceVectorRoutingTable[i][j] = newDistance;

}

}

}

}

}

return distanceVectorRoutingTable;

}

int main() {

vector<vector<int>> adjacencyMatrix = {

{0, 2, 0, 5, 0},

{2, 0, 4, 0, 0},

{0, 4, 0, 6, 3},

{5, 0, 6, 0, 1},

{0, 0, 3, 1, 0}

};

vector<vector<int>> distanceVectorRoutingTable = calculateDistanceVectorRouting(adjacencyMatrix);

cout << "Distance Vector Routing Table:" << endl;

for (int i = 0; i < distanceVectorRoutingTable.size(); ++i) {

for (int j = 0; j < distanceVectorRoutingTable[i].size(); ++j) {

cout << distanceVectorRoutingTable[i][j] << " ";

}

cout << endl;

}

return 0;

}