A. Header file

a. This is the header file sensor.h

```
#ifndef SensorsH
#define SensorsH
//---------------------------------------------------------------------------
#include <Classes.hpp>
#include "Signals.h"
//---------------------------------------------------------------------------
#define SIGNAL_SPEED        1500
#define MAX_SENSORS          10000
//structure of the main packet (control packet)
struct TControlPacket {
public:
        short int iSensor_ID;
        short int iCluster_ID;
        short int CH_ID;
        short int iTarget_ID;                   // if iTarget_ID = -1 then broadcasting
        unsigned short int iDataLength;          // if iDataLength = 0 then no data packet exists
        float fEnergy;
        int iDistance;                          // Distance in msec
        short int iPacket_ID;
};
struct TDataPacket {
public:
        char *data;
};
class TSensorThread;  //
//structure of the data packet
struct TPacketData {
public:
        TControlPacket cp;
        // if inside control packet 'cp', iTarget_ID is -1, then data packet 'dp' is not needed.
        TDataPacket dp;
        TSensorThread *SourceSensor;
        TSensorThread *TargetSensor;
        double dReceiveConsumption;
};
void __fastcall InitLog();
```

```
void __fastcall WriteLog(UnicodeString s);
void __fastcall CloseLog();
void __fastcall WriteLog1(UnicodeString s);

//implementation of TSensorThread
class TSensorThread : public TThread
{
private:
        int iBufferSize;
        int iDataPacketNum;
        int iarLastID[MAX_SENSORS];
        char *buffer;
        __int64 liComputerFreq;
        __int64 __fastcall CustomTickCount();
        __int64 __fastcall CustomTickCounter();

//      CRITICAL_SECTION lock_SignalsCS;
protected:
        list<TSensorThread*> *sens_list;
        void __fastcall ExSleep(DWORD dwTime);
        void __fastcall SendSignal(TPacketData sig);
        __int64 __fastcall ReceiveSignal(TPacketData &pd);
        void __fastcall Send_Data();
        __int64 __fastcall CheckBuffer(TPacketData &pd);
        __int64 __fastcall WaitForACK();
        __int64 __fastcall WaitForData();
        void __fastcall GatherData();
        BOOL __fastcall TransmitData();
        void __fastcall Communicate();
        void __fastcall Clustering();
        void __fastcall Sink();
        void __fastcall UnderwaterSensor();
        void __fastcall Execute();
public:
        DWORD dwGathDataInterval, dwSuspTime;
        int iRetriesCount;
        int TimeSlot;
        int SleepPeriod;
```

```
        int MAX_TRANSMIT_DISTANCE;
        float MIN_CLUST_DISTANCE;
        float MAX_CLUST_DISTANCE;
        float EXTRA_DIST;
        int TRANSMISION_RATE;
        float PACKET_RATIO_ERRORS;
        float PACKET_RATIO_LOST;
        int riBufferSize;
        CRITICAL_SECTION lock_PositionCS;
        double dSignalFrequency;
        BOOL bHead;
        int iHeadNodeID;
        int ibHeadNodeID;
        int iHeadNodeDist;
        TSignals<TPacketData> sens_signals;
        float Battery_Level; // W * h
        float Battery_init; //initial value of battery
        /* The total energy consumption of a sensor after the clustering.
        It contains the energy needed to communicate with the head sensor + the
        sum of the energy spent to forward all the data of a sensor (cluster member) */
        float fLastSignalConsumption;
        float fRxLastSignalConsumption;
        float fTxConsumption;
        float fRxConsumption;
////////////////////////////////////COUNTERS/////////////////////////////////////////////////
        int iSuccTxPackets;
        int iLostTxPackets;
        int iTxTotalData;
        int iAckOrCtrl;
        int iRxDataPackForSensor;
        int iRxDataPackDiscard;
        int iRxAckOrCtrlForSensor;
        int iRxAckOrCtrlDiscard;
        int iRxErrorPackForSensor;
        int iRxErrorPackDiscard;
////////////////////////////////////COUNTERS/////////////////////////////////////////////////
        int iTxOrphan;     //orphan is the sensor node without connection
        int iTxACKs;       // to count the ACKs+Ctrl_packets
```

```
        int iTxRetries;
        int iSuccRxPackets;
        int iRxErrorPackets; // for future reference
        int iRxOrphanDiscard; // for future reference
        int iRxOrphanReceived; // for future reference

        // Pointer to the head sensor. Needed to communicate and send data to it.
        TSensorThread *Parent;
        /* List of pointers to the members (children) of that sensor (that are in the same cluster).  If
        the list has 0 size, then that sensor node is not a head sensor, otherwise it has members
        equal to the size of the list.*/
        list<TSensorThread*> sens_childs;
        // The sensor node ID. The header has always 0.
        int SensorID;
        // The cluster ID that this sensor node belongs to.
        int Cl_id;
        // The X, Y, Z coordinates of the sensor node.
        int Sensor_Left;
        int Sensor_Top;
        int Sensor_Depth;
        int Sensor_Level;
        __fastcall TSensorThread(bool CreateSuspended, list<TSensorThread *> &sl);
        __fastcall ~TSensorThread();
        float __fastcall TransmitEnergyConsumption(int iMaxDist, int iPackSize);
        float __fastcall ReceiveEnergyConsumption(double dCons);
        void __fastcall lock_positionCS();
        void __fastcall unlock_positionCS();
};
//TMessagesThread *MessageThrd;
//---------------------------------------------------------------------------
#endif
// ---------------------------------------------------------------------------------------------------------------------------
```

B. Source code file

a. This is the source file sensor.cpp

```
// ---------------------------------------------------------------------------
#include <vcl.h>
#pragma hdrstop
#include "Sensors.h"
#include "Unit1.h"
#pragma package(smart_init)
// ---------------------------------------------------------------------------
TFileStream *LogFile;
CRITICAL_SECTION lock_LogCS;
void __fastcall InitLog() {
        InitializeCriticalSection(&lock_LogCS);
        LogFile = new TFileStream("log.txt", fmCreate | fmOpenWrite);
}
void __fastcall WriteLog(UnicodeString s) {
        s= IntToStr((int)GetTickCount()) + " - " + s + "\r\n";
        EnterCriticalSection(&lock_LogCS);
        LogFile->Seek(0, soFromEnd);
        LogFile->Write(s.t_str(), s.Length());
        LeaveCriticalSection(&lock_LogCS);
}
void __fastcall CloseLog() {
        DeleteCriticalSection(&lock_LogCS);
        delete LogFile;
}
__int64 __fastcall TSensorThread::CustomTickCount() {
        _LARGE_INTEGER li;
        QueryPerformanceCounter(&li);
        return li.QuadPart;
}
 __int64 __fastcall TSensorThread::CustomTickCounter() {
        _LARGE_INTEGER li;
        QueryPerformanceCounter(&li);
        return li.QuadPart;
 }
void __fastcall TSensorThread::SendSignal(TPacketData sig) {
        list<TSensorThread*>::iterator iter;
```

```
        TSensorThread *sens;
        int dx;
        int dy;
        int dz;
        int iTargSensID;
        lock_positionCS();
        int iTop = Sensor_Top;
        int iLeft = Sensor_Left;
        int iDepth = Sensor_Depth;
        int iSensID = SensorID;
        int SensPrId;
        double NewCx;
        unlock_positionCS();
//keep track every time a signal is send
        __int64 liCurTime = CustomTickCount();
//calculate the packet size
        int iPacketSize;
        if (sig.cp.iDataLength == 0) iPacketSize = sizeof(TControlPacket) * 8;
        if (sig.cp.iDataLength != 0)
                iPacketSize = (sizeof(TControlPacket) + sig.cp.iDataLength) * 8;
//calculate the transmission distance with an extra distance for safety reasons
        int iSendDist = (iHeadNodeDist * (100 + EXTRA_DIST) / 100);
//in iHeadNodeDist the exact distance between a sensor and a head sensor is stored
        if (sig.cp.iDataLength == 0)
                iSendDist = (int)MAX_TRANSMIT_DISTANCE;          //this is the max distance used
mainly when an ACK or a control packet is send.
//calculate the energy consumption when a packet is received
sig.dReceiveConsumption = TransmitEnergyConsumption(iSendDist, iPacketSize) / 5;

        if (Terminated) {
                return;
        }

        if (sig.cp.iDataLength == 0) {
                iAckOrCtrl++;
        } else {
                iTxTotalData++;
        }
```

```
//          run through sensor list
            int iS = sens_list->size();
            for (iter = sens_list->begin(); iter != sens_list->end(); iter++) {
                        sens = *iter;
                        if (sens != this) {
                          sens->lock_positionCS();
//deploy the sensor nodes according to their coordinates
                                    dx = iLeft - sens->Sensor_Left;
                                    dy = iTop - sens->Sensor_Top;
                                    dz = iDepth - sens->Sensor_Depth;
                                    iTargSensID = sens->SensorID;
                                    sens->unlock_positionCS();

                                    double dDist;
                                    double dTTR;
//calculate the destination sensor and the source sensor
                        dDist = sqrt(pow((double)dx,2.)+pow((double)dz,2.)+pow((double)dy,2.));
                                    dTTR = dDist * 1000 / SIGNAL_SPEED;

                                    __int64 liTTR = (liComputerFreq * dTTR) / 1000 + liCurTime;

                                    if (dDist <= iSendDist) {
                                                sig.SourceSensor = this;            //source sensor
                                                sig.TargetSensor = sens;            //destination sensor

/*UnicodeString sLog = IntToStr((int)dwCurTime) + " - Sensor " + IntToStr(SensorID) + " to " +
IntToStr(sens->SensorID) + ". ";
sLog += "TTR: " + IntToStr((int)dTTR) + ", dDis: " + IntToStr((int)dDist);
WriteLog(sLog);
 */
 //sens: signal list for all sensors
                                                sens->sens_signals.insert(sig, liTTR);

                        }
            }
}
   }
```

```
__fastcall TSensorThread::TSensorThread(bool CreateSuspended, list<TSensorThread *> &sl) :
TThread
        (CreateSuspended) {
        int i;
        for (i = 0; i < MAX_SENSORS; i++) {
                iarLastID[i] = 0;
        }
        InitializeCriticalSection(&lock_PositionCS);
        iDataPacketNum = 0;
        fTxConsumption = 0;
        fRxConsumption = 0;
        iSuccTxPackets = 0;
        iLostTxPackets = 0;
        iAckOrCtrl = 0;
        iTxTotalData = 0;
        iRxDataPackForSensor = 0;
        iRxDataPackDiscard = 0;
        iRxAckOrCtrlForSensor = 0;
        iRxAckOrCtrlDiscard = 0;
        iRxErrorPackForSensor = 0;
        iRxErrorPackDiscard = 0;

        iTxACKs = 0;
        iTxOrphan = 0;
        iTxRetries = 0;
        iSuccRxPackets = 0;
        iRxErrorPackets = 0;
        iRxOrphanDiscard = 0;
        iRxOrphanReceived = 0;
        iBufferSize =0;
        riBufferSize =0;
        iHeadNodeID = -1;
        sens_list = &sl;
        sens_list->push_back(this);
}

__fastcall TSensorThread::~TSensorThread() {
        DeleteCriticalSection(&lock_PositionCS);
```

```
}
// Routine that returns an energy consumption for the sensor 'Source'.
// Returns: the energy spend to send data from the sensor 'Source' to the sensor 'Target'.
float __fastcall TSensorThread::TransmitEnergyConsumption(int iMaxDist, int iPackSize) {
        double d, a, TL, SL, It;     // d = distance between sensors (m)
        Battery_init=Battery_Level;                     //keep the initial value of battery
        d= iMaxDist;                    //this is equal to the iHeadNodeDist (exact distance)
        if(d!=0)
        {
//calculation of the underwater environment parameters (a,TL,SL,It)
a= 0.1 * pow(dSignalFrequency, 2.0) / (1 + pow(dSignalFrequency, 2.0)) + 40 * pow(dSignalFrequency,
2.0) / (4100 + pow(dSignalFrequency, 2.0)) + 2.75 / 10000 * pow(dSignalFrequency, 2.0) +0.003;

TL=20*log10(d)+a*d/1000;
SL=TL+70;
It=pow(10.,SL/10)*0.67*pow(10.,-18);
        }
        double P = 4 * 3.14159 * d * d * It;
        double PTT = (double)iPackSize / TRANSMISION_RATE;
        double fEnergyConsumption = P * PTT;

//When the battery level is 0 terminate the sensor node
if ((Terminated) || (Battery_Level - fEnergyConsumption<= 0)) return fEnergyConsumption;

//      Store the energy consumption every time a packet is transmitted
        fTxConsumption =fTxConsumption + fEnergyConsumption;
//Store the energy consumption of the last packet be transmitted
        fLastSignalConsumption = fEnergyConsumption;

        if ( (Battery_Level-fLastSignalConsumption<=0) && (SensorID != 0) ) {
                Terminate();

                Battery_Level = Battery_Level - fEnergyConsumption;
                return fEnergyConsumption;
        } else Battery_Level = Battery_Level - fEnergyConsumption;
        return fEnergyConsumption;
}
```

```
// Routine that returns an energy consumption for the sensor 'Source'.
// Returns: the energy spend to receive data from the sensor 'Target' to the sensor 'Source'.
float __fastcall TSensorThread::ReceiveEnergyConsumption(double dCons) {
	if ((Terminated) || (Battery_Level - dCons<= 0)) return 0.0;

//	fEnergyConsumption += dCons;
	fRxLastSignalConsumption = dCons;
	if ( (Battery_Level - dCons <= 0)  && (SensorID != 0) ) {
		Terminate();
		return 0.0;
	}
	fRxConsumption = fRxConsumption + dCons;
	Battery_Level = Battery_Level - dCons;
	return 0.0;
}
void __fastcall TSensorThread::ExSleep(DWORD dwTime) {
	TPacketData pd;
	__int64 liStart = CustomTickCount();
	__int64 liEnd = CustomTickCount();

	while ((!Terminated) && (((liEnd - liStart) * 1000) / liComputerFreq < dwTime)) {
		BOOL bOK;
		while (sens_signals.pop_front(CustomTickCount(), pd) != -1) ;
		Sleep(5);
		liEnd = CustomTickCount();
	}
}
__int64 __fastcall TSensorThread::ReceiveSignal(TPacketData &pd) {
	__int64 iTime = sens_signals.pop_front(CustomTickCount(), pd);
		__int64 liStart = CustomTickCount();
	if ((liStart%TimeSlot)<(TimeSlot-SleepPeriod))  {
	if (iTime != -1) {

		if (pd.cp.iDataLength == 0) {
			if (pd.cp.iTarget_ID == SensorID) {
				iRxAckOrCtrlForSensor++;

			}
```

```
                    else{
                        iRxAckOrCtrlDiscard++;
                    }

                } else {
                    if (pd.cp.iTarget_ID == SensorID) {
                        iRxDataPackForSensor++;
                    }
                    else
                    {
                        iRxDataPackDiscard++;
                    }
                }
                float fRand = (rand() % 10000) + 1;
                float iRand = fRand / 100;
//              if (iRand <= SIGNAL_NO_RECEIVED) {
//              }
                if (iRand > (PACKET_RATIO_ERRORS + PACKET_RATIO_LOST)) {
                    // received the correct packet.
                    ReceiveEnergyConsumption(pd.dReceiveConsumption);
                } else {
                    iTime = -1;
                    if (iRand <= PACKET_RATIO_ERRORS) {
                        // Received packet with error
                        ReceiveEnergyConsumption(pd.dReceiveConsumption);
                        if (pd.cp.iTarget_ID == SensorID)
                            iRxErrorPackForSensor++; else iRxErrorPackDiscard++;
                    }
                    // Otherwise, the packet was not received at all.
                }
                if (Terminated) {
                    iTime = -1;
                }

            } //if iTime
            }//if %
            return iTime;
}
```

```
void __fastcall TSensorThread::lock_positionCS() {
        EnterCriticalSection(&lock_PositionCS);
}

void __fastcall TSensorThread::unlock_positionCS() {
        LeaveCriticalSection(&lock_PositionCS);
}
//routine for the SINK sensor node
void __fastcall TSensorThread::Sink() {
        // Typical delay to wait member (child)sensors to initialize themselves.
//      Sleep(5000);
        iHeadNodeID = -10000;
        while (!Terminated) {
                WaitForData();
                Sleep(1);
        }
}
void __fastcall TSensorThread::Send_Data() {
        if (iBufferSize == 0) return;
        TPacketData snd;
        snd.cp.iPacket_ID = iDataPacketNum;
        snd.cp.iDataLength = iBufferSize;
        snd.cp.iSensor_ID = SensorID;
        snd.cp.iTarget_ID = iHeadNodeID;
  riBufferSize= riBufferSize + iBufferSize;

  /*    UnicodeString sLog = ";Sensor " + IntToStr(SensorID) + "; sends data to ;" +
IntToStr(iHeadNodeID);
        WriteLog1(sLog);    */

        SendSignal(snd);
//      iBufferSize = 0;
}
__int64 __fastcall TSensorThread::CheckBuffer(TPacketData &pd) {
// Returns TRUE if control packet received, else FALSE.
//      TPacketData rcv;

        while (!Terminated) {
```

```
                __int64 liTime = ReceiveSignal(pd);
                        __int64 liStart = CustomTickCount();
                if (liTime == -1) return -1;
                if (pd.cp.iDataLength == 0) {
                        return liTime;
                }

                if (pd.cp.iTarget_ID == SensorID) {
                  if ((liTime%TimeSlot)<(TimeSlot-SleepPeriod))
                  {
// TODO: Store the data packet to memory.
                        if (pd.cp.iPacket_ID > iarLastID[pd.cp.iSensor_ID]) {
                iBufferSize = iBufferSize + pd.cp.iDataLength + sizeof(TControlPacket);
                                iarLastID[pd.cp.iSensor_ID] = pd.cp.iPacket_ID;
                        }
                        iSuccRxPackets++;
        }
                        // Send a Data ACK
                        TPacketData snd;
                        snd.cp.iDataLength = 0;
                        snd.cp.iSensor_ID = SensorID;
                        snd.cp.iTarget_ID = pd.cp.iSensor_ID;
                        iTxACKs++;
                        SendSignal(snd);

/*      UnicodeString sLog1 = "Sensor " + IntToStr(SensorID) + " received data from " +
IntToStr(pd.cp.iSensor_ID);
                        sLog1 += " Sending ACK";
                        WriteLog(sLog1);  */

                        return -1;
                }  //if (pd.cp.iTarget_ID == SensorID)

                Sleep(1);
          }
          return -1;
}
__int64 __fastcall TSensorThread::WaitForACK() {
```

```
		TPacketData pd;
		__int64 liTime;
		while (!Terminated) {
			liTime = CheckBuffer(pd);
			if (liTime == -1) return -1;
			if (pd.cp.iTarget_ID == SensorID) {
/*				UnicodeString sLog = "Sensor " + IntToStr(SensorID) + " received ACK from "
+ IntToStr(pd.cp.iSensor_ID);
				WriteLog(sLog);*/

				return liTime;
			}
			if ((pd.cp.iTarget_ID == -1) && (bHead) && (iHeadNodeID != -1) &&
(pd.cp.iSensor_ID != iHeadNodeID)) {
				iRxOrphanReceived++;

				// Send the ACK.
				TPacketData snd;
				snd.cp.iDataLength = 0;
				snd.cp.iSensor_ID = SensorID;
				snd.cp.iTarget_ID = pd.cp.iSensor_ID;

/* UnicodeString sLog = "ACK from Sensor " + IntToStr(SensorID) + " to target " +
IntToStr(pd.cp.iSensor_ID);
				WriteLog(sLog);
*/
				iTxACKs++;
				SendSignal(snd);
			} else iRxOrphanDiscard++;
			Sleep(1);
		}
		return -1;
}
__int64 __fastcall TSensorThread::WaitForData() {
		TPacketData pd;
		__int64 liTime;

		while (!Terminated) {
```

```
                liTime = CheckBuffer(pd);
                if (liTime == -1) return -1;
                if ((pd.cp.iTarget_ID == -1) && (bHead) && (iHeadNodeID != -1) &&
(pd.cp.iSensor_ID != iHeadNodeID)) {
                        iRxOrphanReceived++;

                        // Send the ACK.
                        TPacketData snd;
                        snd.cp.iDataLength = 0;
                        snd.cp.iSensor_ID = SensorID;
                        snd.cp.iTarget_ID = pd.cp.iSensor_ID;

/* UnicodeString sLog = "ACK from Sensor " + IntToStr(SensorID) + " to target " +
IntToStr(pd.cp.iSensor_ID);
                        WriteLog(sLog);
*/
                        iTxACKs++;
                        SendSignal(snd);
                        return -1;
                } else iRxOrphanDiscard++;
                Sleep(1);
        }
        return liTime;
}
void __fastcall TSensorThread::GatherData() {
        while (!Terminated) {
                __int64 liSampleTimer = CustomTickCount();
                // Sensing environmental data.
                // We will just increase the buffer. This value can be changed.
                iBufferSize = iBufferSize + 5;

                while (!Terminated) {
                        if (iBufferSize > 50 ) return ;
                        WaitForData();
                        Sleep(1);
                        __int64 liEnd = CustomTickCount();
                        if ((liEnd - liSampleTimer) * 1000 / liComputerFreq > dwGathDataInterval)
break;
```

```
    }
        }
}
BOOL __fastcall TSensorThread::TransmitData() {
        __int64 liSendTime;
        int iRetries;
        BOOL bAck;
        iRetries = 0;
        iDataPacketNum++;
        while (!Terminated) {
                liSendTime = CustomTickCount();
                Send_Data();
//creation of the TDMA schedule
  if ((liSendTime%TimeSlot)<(TimeSlot-SleepPeriod)){
                while (!Terminated) {
                        __int64 liCur = WaitForACK();

                        if (liCur != -1) bAck = TRUE; else bAck = FALSE;
                        __int64 liCurrent = CustomTickCount();
if ((liCurrent - liSendTime) * 1000 / liComputerFreq > 2 * (200 + 1000 * MAX_TRANSMIT_DISTANCE /
SIGNAL_SPEED) )
 {
UnicodeString sLog = "Send time: " + IntToStr(liSendTime) + ". Sensor " + IntToStr(SensorID) + " to
head " + IntToStr(iHeadNodeID) + " (data) ";
sLog += "Timeout ACK: " + IntToStr(liCurrent) + ", Diff: " + IntToStr((liCurrent -
liSendTime)*1000/liComputerFreq)+ " Data " + IntToStr(riBufferSize);
                                WriteLog(sLog);

                                iLostTxPackets++;
                                iRetries++;
                                if (iRetries >= iRetriesCount) {
                                        iBufferSize = 0;
                                        return FALSE;
                                } else {
                                        iTxRetries++;
                                        break;
                                }
                        } else {
```

```
                                        if ((bAck) && ((liCur - liSendTime) > 0)) {
                                                iBufferSize = 0;
                                                iSuccTxPackets++;

// Calculate the round trip time.
                                        int iSigTime = ((liCur - liSendTime) * 1000 / liComputerFreq) / 2;
//calculate the distance between a sensor and a head sensor
                                        iHeadNodeDist = (SIGNAL_SPEED * iSigTime) / 1000;
UnicodeString sLog = "Send time: " + IntToStr(liSendTime) + " - Sensor " + IntToStr(SensorID) + " to
head " + IntToStr(iHeadNodeID) + " (data) ";
sLog += "Received ACK: " + IntToStr(liCur) + ", Diff: " + IntToStr((liCur -
liSendTime)*1000/liComputerFreq)+ " Data " + IntToStr(riBufferSize);
                                                WriteLog(sLog);
                                                return TRUE;
                                        }  //if
                                }  //else
                        }  //while
    }//if timeslot
            } //while
            return FALSE;
}
void __fastcall TSensorThread::Communicate() {
            BOOL b;
            do {
                        GatherData();
                        if (Terminated) return;
                        b = TransmitData();
                        Sleep(1);
            } while (b && (!Terminated));
}
void __fastcall TSensorThread::Clustering() {
            TPacketData snd, rcv;
            snd.cp.iSensor_ID = SensorID;

            __int64 liSendTime;
            int iRetries;

            while (!Terminated) {
```

```
                    iRetries = 0;
                    liSendTime = CustomTickCount();
                    snd.cp.iDataLength = 0;
                    snd.cp.iTarget_ID = -1;
                    iTxOrphan++;
                    SendSignal(snd);
/*  UnicodeString sLog = "Send time: " + IntToStr((int)dwSendTime) + " - Sensor " + IntToStr(SensorID)
+ " requests clustering";
                    WriteLog(sLog);
*/
                    while ( 2 > 1 ) {
                              if (Terminated) return ;
                              __int64 liRecv = ReceiveSignal(rcv);
                              if ( (liRecv != -1) &&
                                        (rcv.cp.iTarget_ID == SensorID) &&
                                        (rcv.cp.iDataLength == 0) && (liRecv - liSendTime)>0 ) {
                                        __int64 liSigTime = (liRecv - liSendTime) / 2;
                    int iDist = (SIGNAL_SPEED * (liSigTime * 1000 / liComputerFreq)) / 1000;
//calculate the head sensors -the battery level must be more than the 50% of the initial battery value-
          if ( (iDist >= (MAX_TRANSMIT_DISTANCE * MIN_CLUST_DISTANCE / 100)) && (iDist <=
(MAX_TRANSMIT_DISTANCE * MAX_CLUST_DISTANCE / 100)) && (Battery_Level  >=
(0.5*Battery_init)))
                                                  bHead = TRUE; else bHead = FALSE;

                                        iHeadNodeDist = iDist;
 /* UnicodeString sLog = "Sensor " + IntToStr(SensorID) + " got clustering ACK from " +
IntToStr(rcv.cp.iSensor_ID);
sLog += ". Time: " + IntToStr((int)dwSigTime) + ", Dist: " + IntToStr((int)(iDist));
                                        WriteLog(sLog);
 */
                                        iHeadNodeID = rcv.cp.iSensor_ID;
                                        return ;
                              }
                              Sleep(1);
                              __int64 liCurTime = CustomTickCount();
                              if ((liCurTime - liSendTime) * 1000 / liComputerFreq > 2 * (200 + 1000 *
MAX_TRANSMIT_DISTANCE / SIGNAL_SPEED) ) {
/* UnicodeString sLog = ";Sensor " + IntToStr(SensorID) + "; got no clustering response.;";
```

```
            sLog += "; Time passed: ;" + IntToStr((int)(liCurTime - liSendTime));
                                    WriteLog1(sLog);
*/
                                    iRetries++;
                                    if (iRetries >= iRetriesCount) {
/* UnicodeString sLog = "Sensor " + IntToStr(SensorID) + " tried 3 times with no luck. Will sleep.";
                                            WriteLog(sLog);
 */
                                            ExSleep(dwSuspTime);
                                            break;
                                    } else {
                                            break;
                                    }
                            }
                    }
            }
}
void __fastcall TSensorThread::UnderwaterSensor() {
            while (!Terminated) {
                    iHeadNodeID = -1;
/* UnicodeString sLog = "Sensor " + IntToStr(SensorID) + " is orphan. Starting clustering.";
                    WriteLog(sLog);
*/
                    Clustering();
                    if (Terminated) return;
                    Communicate();
                    Sleep(1);
            }
}
// ---------------------------------------------------------------------------
void __fastcall TSensorThread::Execute() {
            _LARGE_INTEGER li;
            Randomize();
            QueryPerformanceFrequency(&li);
            liComputerFreq = li.QuadPart;
            iHeadNodeDist = 0;
            fLastSignalConsumption = 0;
            if (SensorID == 0) {
```

```
                bHead = TRUE;
                Sink();
        } else {
                bHead = FALSE;
                UnderwaterSensor();
        }
}
// ---------------------------------------------------------------------------------------------------------------------
```

C. <u>The following source code gives a solution for the mobile sensor problem</u>

```
void __fastcall TSensorThread::Clustering() {
        TPacketData snd, rcv;
        snd.cp.iSensor_ID = SensorID;
        __int64 liSendTime;
        int iRetries;
        while (!Terminated) {
                iRetries = 0;
                liSendTime = CustomTickCount();
                snd.cp.iDataLength = 0;
                snd.cp.iTarget_ID = -1;
                iTxOrphan++;
                SendSignal(snd);

/*UnicodeString sLog = "Send time: " + IntToStr((int)dwSendTime) + " - Sensor " + IntToStr(SensorID) +
" requests clustering";
WriteLog(sLog);
*/
        while ( 2 > 1 ) {
                if (Terminated) return ;
                __int64 liRecv = ReceiveSignal(rcv);
                if ( (liRecv != -1) &&
                        (rcv.cp.iTarget_ID == SensorID) &&
                        (rcv.cp.iDataLength == 0) && (liRecv - liSendTime)>0 ) {
                        __int64 liSigTime = (liRecv - liSendTime) / 2;
                        int iDist = (SIGNAL_SPEED * (liSigTime * 1000 / liComputerFreq)) / 1000;
//calculate the head sensors -the battery level must be more than the 50% of the initial battery value-
```

```
if ( (iDist >= (MAX_TRANSMIT_DISTANCE * MIN_CLUST_DISTANCE / 100)) && (iDist <=
(MAX_TRANSMIT_DISTANCE * MAX_CLUST_DISTANCE / 100)) && (Battery_Level  >=
(0.5*Battery_init)))
                                bHead = TRUE; else bHead = FALSE;

                                iHeadNodeDist = iDist;

/* UnicodeString sLog = "Sensor " + IntToStr(SensorID) + " got clustering ACK from " +
IntToStr(rcv.cp.iSensor_ID);
        sLog += ". Time: " + IntToStr((int)dwSigTime) + ", Dist: " + IntToStr((int)(iDist));
                                        WriteLog(sLog);     */
                                iHeadNodeID = rcv.cp.iSensor_ID;
                                return ;
                                }
                                Sleep(1);
                                __int64 liCurTime = CustomTickCount();
if ((liCurTime - liSendTime) * 1000 / liComputerFreq > 2 * (200 + 1000 * MAX_TRANSMIT_DISTANCE /
SIGNAL_SPEED) ) {
/*          UnicodeString sLog = ";Sensor " + IntToStr(SensorID) + "; got no clustering response.;";
                sLog += "; Time passed: ;" + IntToStr((int)(liCurTime - liSendTime));
                WriteLog1(sLog);   */
                                        iRetries++;
                                        if (iRetries >= iRetriesCount) {
 /*UnicodeString sLog = "Sensor " + IntToStr(SensorID) + " tried 3 times with no luck. Will sleep.";
                                                WriteLog(sLog);   */
                                                ExSleep(dwSuspTime);
                                                break;
                                        } else {
                if(this->bHead)     this->bHead = false;      //  If CH ability exists remove it
                                                list<TSensorThread*>::iterator iter;
        TSensorThread *sens;    // TSensorThread type pointer for storing iterator's objects below
        TSensorThread *TheNextCH;   //  Pointer to store the Next CH used, only below
        double NextCHDist = MAX_TRANSMIT_DISTANCE;     // The distance of the sensor relative to
the orphan (sensor without connection)
        for (iter = sens_list->begin(); iter != sens_list->end(); iter++) {
                                                sens = *iter;
         if (sens != this) {      // not the object from the iterator
                double dx = Sensor_Left - sens->Sensor_Left;
```

```
                double dy = Sensor_Top - sens->Sensor_Top;      // Distance between
orphan(mobile sensor) and the sensor into cluster
                double dz = Sensor_Depth - sens->Sensor_Depth;
                double dDist = sqrt(pow((double)dx,2.)+pow((double)dz,2.)+pow((double)dy,2.));  //
3D space distance formula
                NextCHDist = dDist;
                if(dDist <= NextCHDist){    // Sensors distance shorter than the previous one
                                NextCHDist = dDist;         // Save distance
                                TheNextCH = *iter;          // Save sensor's pointer
                                                             }
                                                    }
                                                }   //iterator
        if(NextCHDist < MAX_TRANSMIT_DISTANCE && NextCHDist > 0.){   // If the shorter distance
object found above is in range then turn it to head
                TheNextCH->bHead = true;            // turn the sensor you found into CH
                                                }
                                  //        break;
                                }  // else end
                        }
                }
        }
}
// ---------------------------------------------------------------------------------------------------------------------------
```

D. <u>The following source codes give a solution for the CH failure problem</u>

a. <u>Backup_CH procedure</u>

```
void __fastcall TSensorThread::Backup_CH() {
        TPacketData snd, rcv;
        snd.cp.iSensor_ID = SensorID;
        __int64 liSendTime;
        int iRetries;
        while (!Terminated) {
                iRetries = 0;
                liSendTime = CustomTickCount();
                snd.cp.iDataLength = 0;
                snd.cp.iTarget_ID = -1;
                iTxOrphan++;
                SendSignal(snd);
```

```
/*UnicodeString sLog = "Send time: " + IntToStr((int)dwSendTime) + " - Sensor " + IntToStr(SensorID) +
" requests Backup_CH";
WriteLog(sLog);
*/
        while ( 2 > 1 ) {
                if (Terminated) return ;
                __int64 liRecv = ReceiveSignal(rcv);
                if ( (liRecv != -1) &&
                        (rcv.cp.iTarget_ID == SensorID) &&
                        (rcv.cp.iDataLength == 0) && (liRecv - liSendTime)>0 ) {
                        __int64 liSigTime = (liRecv - liSendTime) / 2;
                        int iDist = (SIGNAL_SPEED * (liSigTime * 1000 / liComputerFreq)) / 1000;
//find the sensors close to the CH -the battery level must be more than the 50% of the initial battery
value-
if ( (iDist <= (MAX_TRANSMIT_DISTANCE * MIN_CLUST_DISTANCE / 100)) && (Battery_Level  >=
(0.5*Battery_init)))
backupHead = TRUE; else  backupHead = FALSE;
                        iHeadNodeDist = iDist;
                        iHeadNodeID = rcv.cp.iSensor_ID;
                        return ;
                        }
                        Sleep(1);
                        __int64 liCurTime = CustomTickCount();
if ((liCurTime - liSendTime) * 1000 / liComputerFreq > 2 * (200 + 1000 * MAX_TRANSMIT_DISTANCE /
SIGNAL_SPEED) ) {
//the backup head will try to connect with the sensor head of the upper tier cluster
                                iRetries++;
                                if (iRetries >= iRetriesCount) {
 /*UnicodeString sLog = "Sensor " + IntToStr(SensorID) + " tried 3 times with no luck. Will sleep.";
                                        WriteLog(sLog);   */
                                        ExSleep(dwSuspTime);
                                        break;
                                } else {
                                        list<TSensorThread*>::iterator iter;
        TSensorThread *sens;    // TSensorThread type pointer for storing iterator's objects below
        TSensorThread *TheNextCH;   //  Pointer to store the Next CH used, only below
        double NextCHDist = MAX_TRANSMIT_DISTANCE;     // The distance of the sensor (upper
tier) relative to the backup head (used only below)
```

```
        for (iter = sens_list->begin(); iter != sens_list->end(); iter++) {
                                        sens = *iter;
        if (sens != this) {      // not the object from the iterator
                double dx = Sensor_Left - sens->Sensor_Left;
                double dy = Sensor_Top - sens->Sensor_Top;      // Distance between backup head
and the sensor into the upper tier cluster
                double dz = Sensor_Depth - sens->Sensor_Depth;
        double dDist = sqrt(pow((double)dx,2.)+pow((double)dz,2.)+pow((double)dy,2.));    // 3D
space distance formula
                NextCHDist = dDist;
                if(dDist <= NextCHDist){    // Sensors distance shorter than the previous one
                                NextCHDist = dDist;         // Save distance
                                TheNextCH = *iter;          // Save sensor's pointer
                                                }
                                        }
                                }    //iterator
        if(NextCHDist < MAX_TRANSMIT_DISTANCE && NextCHDist > 0.){    // If the shorter distance
object found above is in range then turn it to head
                TheNextCH->bHead = true;            // turn the sensor you found into CH
                                        }
                                //      break;
                                } // else end
                        }
                }
        }
}
// ------------------------------------------------------------------------------------------------------------------------
```

b. <u>Failure detection procedure</u>

i. <u>This procedure inform the CH for the BackupCH's status</u>

```
void __fastcall TSensorThread::Send_Status() {
        TPacketData snd;
        snd.cp.iPacket_ID = iDataPacketNum;
        snd.cp.iDataLength = Battery_Level;
        snd.cp.iSensor_ID = SensorID;                   //the backup CH id
        snd.cp.iTarget_ID = iHeadNodeID;
  /*    UnicodeString sLog = ";Sensor " + IntToStr(SensorID) + "; sends battery status to ;" +
IntToStr(iHeadNodeID);
        WriteLog1(sLog);    */
```

```
        SendSignal(snd);
}
```

ii. <u>This procedure inform the BackupCH for the CH's status</u>

```
 void __fastcall TSensorThread::Send_CH_Status() {
        TPacketData snd;
        snd.cp.iPacket_ID = iDataPacketNum;
        snd.cp.iDataLength = Battery_Level;
        snd.cp.iSensor_ID = SensorID;                    //the CH id
        snd.cp.iTarget_ID = iBackupCHid;
  /*    UnicodeString sLog = ";Sensor " + IntToStr(SensorID) + "; sends battery status to ;" +
IntToStr(iBackupCHid);     WriteLog1(sLog);   */
        SendSignal(snd);
}
```

c. Recovery activation

i. Check the status of backup CH. If critical, trigger the Backup CH procedure.

```
void __fastcall TSensorThread::check_B_CH() {
                __int64 liSampleTimer = CustomTickCount();

                while (!Terminated) {
                        if (Battery_Level >= (0.3*Battery_init ) ){
                        Communicate();
                        Send_CH_Status ();
                        } else  Backup_CH();     //Battery level is lower than 30%.  Find a new
backup CH

    }
        }
// ---------------------------------------------------------------------------------------------------------------------------
```

ii. Check the status of CH. If critical, the backup CH will appointed as a new head of the cluster group.

```
void __fastcall TSensorThread::check_CH() {
                while (!Terminated) {
                        if (Battery_Level >= (0.3*Battery_init ) ){
                        Communicate();
                        Send_Status ();
                        } else  {   bHead=True;
```

```
                                    iHeadNodeID = Sensor_ID;            //the head sensor id is the
backup CH id
                                    snd.cp.iTarget_ID = iHeadNodeID;  //send the new CH id to the
cluster members                     }
                                }
        }
// ------------------------------------------------------------------------------------------------------------------------
```