QSD Monitoring Network

Communication over UDP Ethernet

T. Barrett

July 2018

Communication over UDP Ethernet

The sensor measurements acquired by each microcontroller node must be aggregated by a "Collector" device, and handled in one location for archiving and for real-time monitoring. Therefore, a reliable, low-maintenance method of transmitting data around the Accelerator building is needed. It makes sense for this purpose to use the building's internal private local area network (LAN), which connects all the green-labelled sockets to each other via a dedicated ethernet switch (*HP JL256A Aruba 2930F*) in the Meeting Room area. This network is completely isolated form the external university internet network, and so can be used freely for our lab purposes without conflicts. Wired ethernet is preferable here for data transmission, as a wireless solution could face intermittent connection issues due to signal strength around the building, and additionally could lead to interference of the microwave frequencies with the experiments

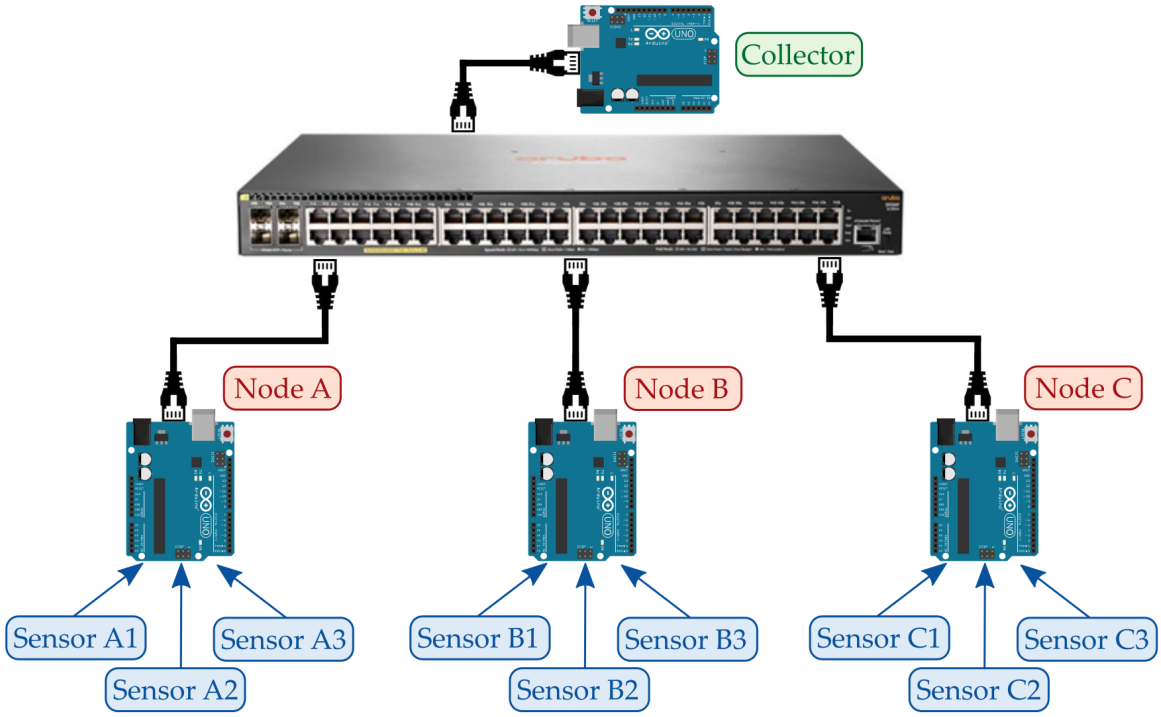

FIGURE 1: Schematic of data collection: There are many microcontrollers located in different areas and rooms around the building - labelled "Nodes". Multiple sensors attached to each node then collect measurements (temperature, laser powers, etc.) in that particular local vicinity. Making use of a dedicated ethernet switch and a private internal LAN, a "Collector" device periodically requests the data from each node sequentially via UDP communication, ready for processing and archiving.

The most common way of sending pieces of data (known as packets) over a network is the *transmission control protocol* (TCP), which is used in the transport layer of the *internet protocol* (IP) suite. This means that information is sent to devices identified by an IP address. TCP is known as a connection-oriented protocol, which means that a connection is set up between two devices in order for them to be able to talk. The protocol includes a form of *handshaking* - a process which occurs between two devices before full communication begins, for example

to indicate that both are active and ready to send/receive data, etc. Bits of data are numbered, allowing them to be re-ordered correctly if they are received in the wrong order, as well as being retransmitted fully if no acknowledgement is sent back.

All these error-checking features make TCP a very reliable protocol, and is the reason it is widely used for most data transmission on the internet (for example being used between a browser and a server for loading web pages). However, the downside is that it is therefore slower, more complex, and has significant overheads associated with it. As an alternative, another widely used method is known as *user datagram protocol* (UDP). In contrast to TCP, UDP is known as a *connection-less protocol*, and basically throws out all the error-checking features of TCP. This means that messages are sent out, but there is no guarantee of it being delivered. The analogy is that UDP is like posting a standard letter - there is no way for the sender to know if the letter arrived. The advantage is that this protocol is much simpler, minimalistic, and faster way of sending packets, with appreciably less overhead in the system. It is becoming more popular now with the rise of *internet of things* (IoT) systems, because it is ideal for collecting sensor data in low power devices, where it is not a problem if a packet is dropped every now and again. It is more suitable in situations where it is more preferable to drop a packet, than to hold up the entire data collection process due to retransmission attempts of packets. UDP is suitable for the sensor network in the Accelerator building, because there is very little traffic anyway on the internal network - indeed, a test was performed in which 10,000 packets were sent between two devices over the network, and in fact not a single packet was dropped.

0.1 Hardware Options for UDP with Arduino

For ethernet communication with Arduino, there are several options in terms of hardware. At the time of writing, none of the offical Arduino boards have built-in ethernet connectivity, and so external hardware needs to be bought. The most commonly-available adapters are based on either one of two ethernet controller chips - the *Microchip ENC28J60*, or the *WIZNet W5100* (or sometimes *W5500*). The main difference is that the W5100 chip includes on-board a fully hardwired IP stack (a set of software networking protocols needed for communication, including both TCP and UDP). In contrast, the ENC28J60 does not, and so requires much more to be implemented on the Arduino itself, leading to the libraries and sketches taking up much more memory, and being more bloated. The W5100 is also compatible with the official Arduino *Ethernet.h* library, whereas the ENC28J60 requires one of several third-party libraries. After extensive tests using both types of ethernet controller chips, it was found that the W5100 was far more reliable, as well as being simpler to implement. For this reason, the QSD Monitoring Network should always have sensor nodes that are based on the W5100.

There are several ways to add connectivity with the WIZnet chips, some of which are shown in Fig. 2. The official *Arduino Ethernet 2 Shield* is shown in Fig. 2 a), which works nicely but is a little more expensive (∼£30) and difficult to get hold of. Another option is to use the *Freetronics EtherTen* (or its big brother, the *EtherMega*), shown in Fig. 2 b). These boards are based on the Uno and Mega, but with additional upgrades and ethernet connectivity already built-in. These again work perfectly well, but are more expensive - £40 for the EtherTen and £90 for the EtherMega. Finally, a nice option is the *Mini W5100 Ethernet Adapter Module*, shown in Fig. 2 c),

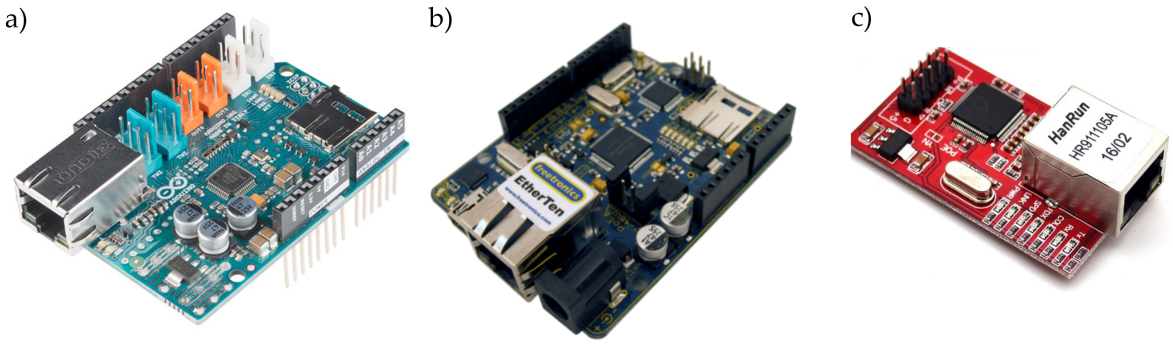

FIGURE 2: Some hardware options, all based on WIZnet controller chips, for adding ethernet functionality to the Arduino: a) The official Arduino Ethernet 2 Shield. b) The EtherTen from Freetronics. c) The Mini W5100 Ethernet Adapter Module.

which can be picked up easily for ~£4 on eBay, and communicate with any model of Arudino board using digital pins over the SPI bus interface.

Using the Mini W5100 Adapter Module

Due to the extremely low cost, easy availability, and simplicity of use, the preferred adapter would usually be the Mini W5100 Adapter Module. To use this module, six connections are required: in addition to +5 V power, GND, and a chip select (SS) pin, there are three SPI pins needed. These are MOSI (*Master Out Slave In*), MISO (*Master In Slave Out*), and SCK (*Clock*). The Arduino pins which are capable of providing these signals are different depending on the model of Arduino being used, and are shown for the Mega, Uno, and Leonardo in Table 1 below.

W5100	Mega	Uno	Leonardo
+5V	+5V	+5V	+5V
GND	GND	GND	GND
SS	10	10	10
MOSI	51	11	ICSP-4
MISO	50	12	ICSP-1
SCK	52	13	ICSP-3

TABLE 1: SPI pins for the Arduino Mega, Uno, and Leonardo.

The pin-out arrangement on the ethernet adapter board is shown in Fig. 3 a), and the connections needed for using it with an Arduino Mega are shown in the Fritzing diagram in Fig. 3 b). Note that even though the hardware SS pin on the Mega is specified to be Pin 53 (see Arduino SPI documentation web page for this), it is actually easier to use Pin 10 because this is defined in the *Ethernet.h* library already for convenience.However, Pin 53 should still be set as an output in the sketch anyway, to ensure the SPI communication works properly.

The Arduino should be loaded with a sketch containing the libraries *SPI.h*, *Ethernet.h*, and *EthernetUdp.h* in order to work. The most important commands needed in order to achieve UDP communication with these libraries are given in Table 2 below.

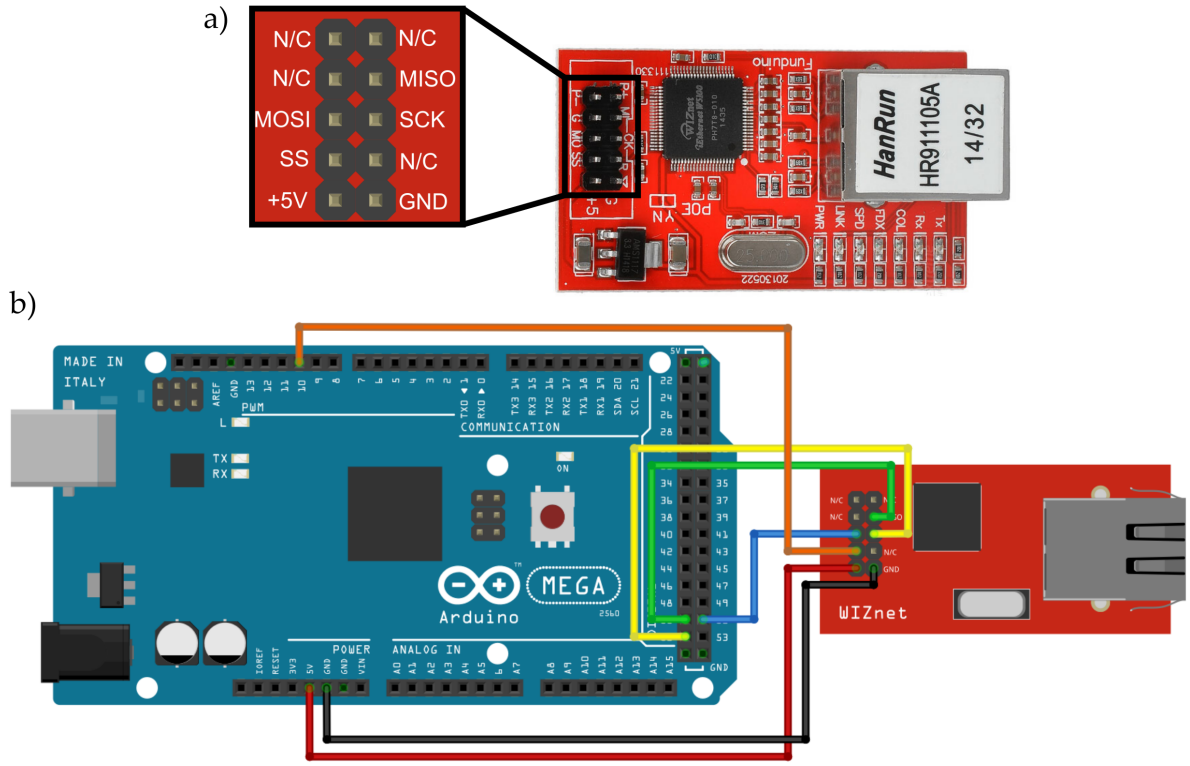

FIGURE 3: Wiring for the Mini W5100 Adapter Module. a) The pin out arrangement and labelling of the module itself. b) Connections required specifically for use with the Arduino Mega.

Command	**Purpose**
EthernetUDP Udp	Create an instance of EthernetUDP to work with.
Ethernet.begin(mac,ipAddress)	Initialise the Ethernet library and settings (ensure that the *mac* is a unique hexadecimal on the network).
Udp.begin(localPort)	Initialise the EthernetUdp library and it settings. The parameter *localPort* is the port on which to listen.
Udp.beginPacket(ipRemote,localPort)	Starts a connection to write UDP data to the remote connection.
Udp.print(string)	Write a string to the remote device (should always called after *Udp.beginPacket*).
Udp.endPacket()	Ends the remote connection (should be called after *Udp.print*).
Udp.parsePacket()	Checks for the presence of a UDP packet, and reports the size.
UDP.read(packetBuffer, MaxSize)	Reads UDP data into the buffer *packetBuffer* (can only be called after *Udp.parsePacket*).

TABLE 2: Most important commands for UDP ethernet communication with Arduino.

0.2 A Minimal Working Example: Ping-Pong with UDP

For testing the ethernet UDP communication, a simple example is provided here. The setup is shown in Fig. 4 a), which shows two Arduino devices - called *client* and *server* - which are each connected to their own Mini W5100 Adapter Module for communication with each other. Both Arduino boards are also connected to a PC via USB, to be able to read out diagnostic information over two independent serial ports - just to check everything is working. A photograph of the arrangement can be seen in Fig. 4 c). The scheme sees the client sending a message to the server - which in this example is the string ”Ping” once every three seconds. The server is constantly listening and checking for new packets arriving over UDP, and if it receives something it will immediately reply with the string ”Pong”. The sketches uploaded to each of the Arduino client and server boards are given in the sections **Arduino Client Code** and **Arduino Server Code** below, respectively. A screenshot of the output seen on the serial monitors for each device is shown in Fig. 4 b), indicating the messages ”Ping” and ”Pong” being passed back and forth between the nodes via UDP. The entire monitoring network in the building is based on a scaled-up version of this basic scheme, in which a client board (”Collector”) sends out commands to multiple server boards, requesting that they reply with their sensor measurement data.

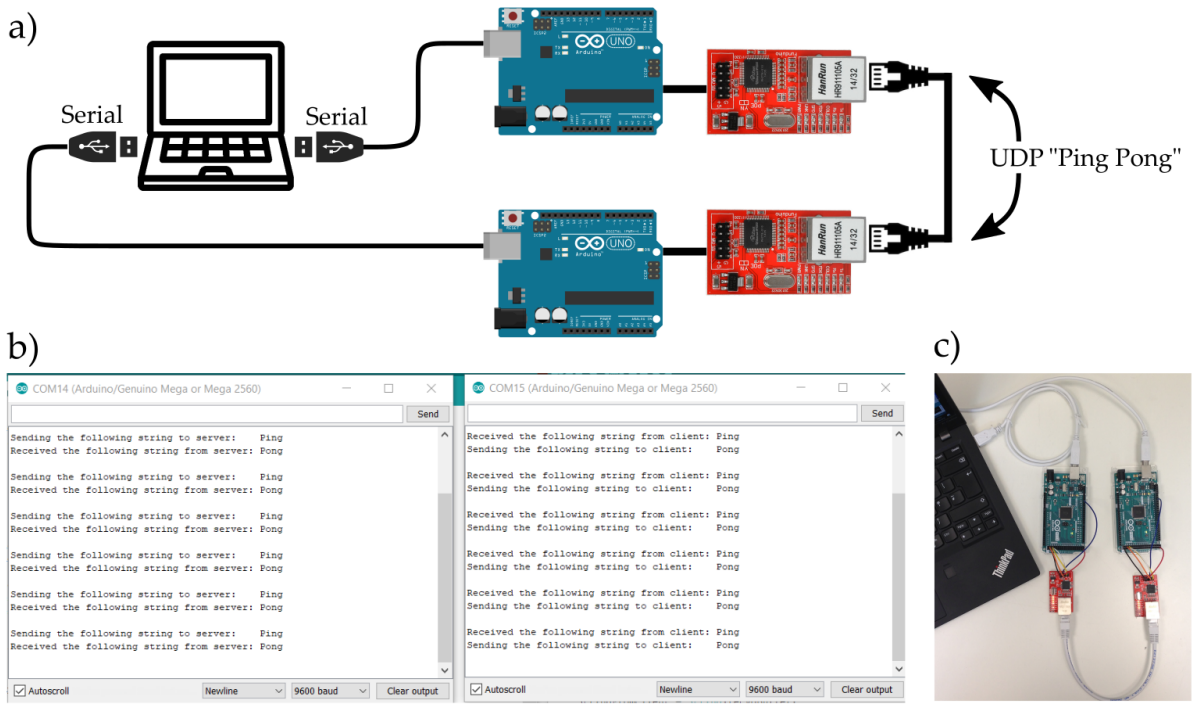

FIGURE 4: Demonstration of ”Ping Pong” communication using UDP. a) Schematic of the setup - two Arduino boards communicate via UDP through the W5100 adapter modules, with USB connection to a PC for monitoring and debugging. b) Screenshot of the results from the serial port monitors for each device, showing that the messages ”Ping” and ”Pong” are successfully passed back and forth. c) Photograph of the setup.

Arduino Client Code

```
/* UDP Ping Pong (Client Board Code)
 *
 * Code sends the string "Ping" to a remote server board,
 * and then prints the reply it receives.
*/

#include <SPI.h>
#include <Ethernet.h>
#include<EthernetUdp.h>

// Initialise Ethernet Parameters ////////////////////////////
static uint8_t mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xEE };
IPAddress ipClient(169, 254, 128, 105);
IPAddress ipServer(169, 254, 128, 106);
const int localPort = 1396;
char recvdBuffer[10]; // A buffer to hold incoming strings
int recvdPacketSize;
EthernetUDP Udp;
#define UDP_TX_PACKET_MAX_SIZE 50 // Increase UDP size
String StringToServer = "Ping";
String StringFromServer;

void setup() {

  delay(1000);
  while (!Serial) {}
  Serial.begin(9600);

  // Pin 10 should be kept as output for CS, even when using using MEGA (10 is
  // defined in Ethernet.h library already). Pin 53 should still be set as
  // output if using a Mega, for SPI to work properly.
  pinMode(10, OUTPUT);
  pinMode(53, OUTPUT);

  // Begin ethernet and UDP
  Ethernet.begin(mac, ipClient);
  Udp.begin(localPort);
  delay(1000);
}

void loop() {

  // Send the string to the remote server board
  Serial.print("Sending the following string to server:    ");
  Serial.println(StringToServer);
  Udp.beginPacket(ipServer, localPort);
  Udp.print(StringToServer);
  Udp.endPacket();
  delay(10); // Allow time for reply to come through

  recvdPacketSize = 0;
  recvdPacketSize = Udp.parsePacket();
  // Check if packet received, and if so read it
  if (recvdPacketSize > 0) {
    Udp.read(recvdBuffer, UDP_TX_PACKET_MAX_SIZE);
    StringFromServer = String(recvdBuffer);
    Serial.print("Received the following string from server: ");
  } else {
    StringFromServer = "No string received from server";
  }
  Serial.println(StringFromServer);
  Serial.println("");

  // Clear received buffer, ready for next one
  memset(recvdBuffer, 0, sizeof(recvdBuffer));
  delay(3000);
}
```

Arduino Server Code

```
/* UDP Ping Pong (Server Board Code)
 *
 * Code receives a string from a remote client board,
 * and then sends back the string "Pong".
*/

#include <SPI.h>
#include <Ethernet.h>
#include<EthernetUdp.h>

// Initialise Ethernet Parameters ////////////////////////////
static uint8_t mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
IPAddress ipClient(169, 254, 128, 105);
IPAddress ipServer(169, 254, 128, 106);
const int localPort = 1396;
char recvdBuffer[10];
int recvdPacketSize;
EthernetUDP Udp;
#define UDP_TX_PACKET_MAX_SIZE 50 // Increase UDP size

String StringToClient = "Pong";
String StringFromClient;

void setup() {

  delay(1000);
  while (!Serial) {}
  Serial.begin(9600);

  // Pin 10 should be kept as output for CS, even when using using MEGA (10 is
  // defined in Ethernet.h library already). Pin 53 should still be set as
  // output if using a Mega, for SPI to work properly.
  pinMode(10, OUTPUT);
  pinMode(53, OUTPUT);

  // Begin ethernet and UDP
  Ethernet.begin(mac, ipServer);
  Udp.begin(localPort);
  delay(1000);
}

void loop() {

  recvdPacketSize = Udp.parsePacket();

  if (recvdPacketSize > 0) {

    Udp.read(recvdBuffer, UDP_TX_PACKET_MAX_SIZE);
    StringFromClient = String(recvdBuffer);
    Serial.print("Received the following string from client: ");
    Serial.println(StringFromClient);

    Serial.print("Sending the following string to client:    ");
    Serial.println(StringToClient);
    Serial.println("");

    Udp.beginPacket(ipClient, localPort);
    Udp.print(StringToClient);
    Udp.endPacket();

    memset(recvdBuffer, 0, sizeof(recvdBuffer));
  }
}
```