Network Of Embeeded Devices

Iakovidis Ioannis

October 9, 2019

Contents

1 Code

In this link there is a github repository with the code used. Code

2 Introduction

In this document we explain the implementation of a message exchange system, based on a embedded real time system that controls the exchanges.

The system is tested on a adhoc network of Raspberry pi Zero W devices. With a limited operated system (not straight Rasbian). First all devices are set to create an wifi network with the same SSID and name. So if they are in each others wifi's range the they belong in the same network and can communicate. We implement a C application using TCP sockets that handles the communication in each device (node).

The specifications of the system are that each node stores a list (S_{buf}) of all (the N latest) the messages that it has produced and all the messages that it has received from the other nodes. Each node produces a new message each T seconds for a random node in the network. The node also sends that list to all devices in the network. The list have to be maintained such that there are no multiplicities.

We have divide our code into server and client code. In the following paragraphs we will explain the implementation of the two portions and we will present performance measurements of the system.

3 Program Structure and Helper Functions

First of all the way we divide our code is to server and client code is that we launch two threads one executing client and one server code. The server handles reading incoming messages and the client sends messages. With this choice we have made our message-list a shared variable as it is updated when a new message is generated, when we send a message and when we recieve a message that it's not in the list.

In the message list (buffer) for testing and functionality purposes stores for each message the producer, the receiver, the time of creation and the list of devices that it has been send to. For maintaining the list we use a global variable head and each time we write in the list we write in the head position and we increament it modulo the size of the buffer. Also because the buffer is shared we use a mutex lock every time we access it.
More precise:

```
#define BUFFER_SIZE 2000
int head=0;
struct buffer_elem * buffer;
struct buffer_elem{
   int* devs;// list of devices devs[i]==1 means the i'th device has recieved this message
   char message[300];// producer/recieve/time/message
};
void writeElem(struct buffer_elem *buffer,char * message); // write the message string in the buffer
int Isinbuffer(struct buffer_elem *buffer,char * message);//Isinbuffer(buffer,message)==1 means the
    message string is in the buffer
```

As concerning the communication for sending a message we first send the size in bytes of the message with a fixed 3 byte message and then we send the message. When in a connection we finish sending messages we send the end of messaging indicator ("EOM", 3 bytes). In the proccess of reading messages we do the same proccedure each time we loop reading the length (3 bytes) end then the message if the length is equal to "EOM" we finish reading.

4 Server

In the server thread we bind our ip to the server and we open sockets accepting incoming requests. Because we want to accept requests constantly (and accept is a blocking call) we make one more thread that processes the sockets the server has opened. And so in the server thread we just accept requests and save their file descriptors in a buffer and the working thread processes them. So following C-phedocode shows the structure of the server code the structure

```
void* server_routine(void* arg){
  /*----------------Server Code----------------------*/
  socketbuf=(int *)calloc(MAX_PENDING_REQS, sizeof(int));
  int server_fd, new_socket;
  struct sockaddr_in address;
  int addrlen = sizeof(address);
  server_fd = Socket(AF_INET, SOCK_STREAM, 0)) == 0)
  address.sin_family = AF_INET;
  address.sin_addr.s_addr = INADDR_ANY;
  address.sin_port = htons( PORT );
  Bind(server_fd, (struct sockaddr*)&address,sizeof(address))
  if (listen(server_fd, 3) < 0)
  pthread_t pid;
  pthread_create(&pid,NULL,pthread_routine,NULL);
  while (1) {
  if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
              (socklen_t*)&addrlen))<0)
  {
     perror("accept");
     exit(EXIT_FAILURE);
  }
  socketbuf_insert(socketbuf,new_socket);
  }
}
```

For exiting the server thread we use the signal SIGUSR1 and pthread_kill() from the client thread.

In the working thread we remove an element form the socket buffer and we makes the communication. The communication consists of reading incomming messages and updating the message buffer if the incoming message is not in the buffer and we are not the destined receiver of this message.

5 Client

In the client thread (main) we loop over the devices opening sockets for each one and when this is successful we make the communication. In a communication we send all the messages from the list that haven't yet been sent to the device. Also in the client code we keep the tine to generate new messages and exit the program when the input total time has passed. In the following C-phedocode we see the structure of the client code.

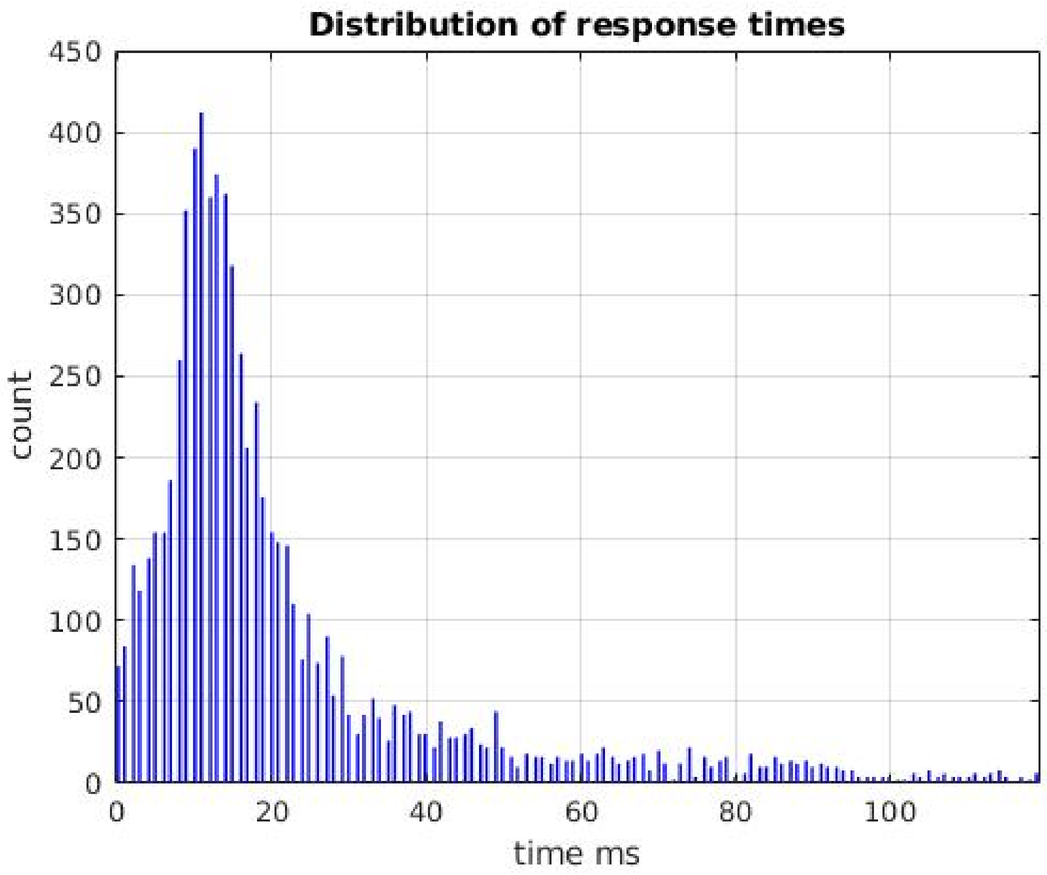
```
 /*Get ips from the device  list */
char** ip=(char** )malloc(n*sizeof(char*));
getIP( devlist ,ip);
int sock = 0;
struct sockaddr_in serv_addr;
/*Until time runs out*/
while(s<N){
```

```
/*For all devices*/
   for(int i=0;i<n;i++){
   gettimeofday(&now, NULL);
   if(diference(now,prev)>period){
      gettimeofday(&prev, NULL);
      char* mes;
      generateMessage(&mes,devlist);
      pthread_mutex_lock(&lock);
      writeElem(buffer,mes);
      pthread_mutex_unlock(&lock);
      s++;
   }
   /*Connect to the i'th Device*/
   if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0) { a=1;}
   if(a==0){
      /*Send messages from the list that have not been sent to the i'th device*/
      char* out;
      pthread_mutex_lock(&lock);
      Send_missing_messages(buffer,sock);
      pthread_mutex_unlock(&lock);
      /*Send End of Messaging*/
      send(sock,"EOM",strlen("EOM"),0);
    }
   close(sock);}}
   pthread_kill(server, SIGUSR1);
   pthread_join(server, NULL);
```

We use this way of keeping time for generation of the messages because if we had used timers and signals because for the a substantial amount of time the execution is in locks for the integrity of the list we can't use insert it to the list. So we use this simplified approach that works fine as far as performance.

6 Performance Measurements

In order to successfully implement such a system we need to choose the right performance metrics and improve the application based on the results. An important performance metric is the time from message generation until its received fo rm the right node (T_{total}). Also for the system to be real time we have to ensure that the message generation is periodic with period T. But to measure time we need to have synchronized clocks among the nodes of the network otherwise our measurements could be off or not make any meaning. So first of all we synchronize the clocks for the processes that we have in our network and then we proceed with the main execution. Below we see the measurement of binning T_{total} for a 2 hour session with 3 nodes on the left and a two .



For the 3 node session we see that the measurements satisfy a nearly exponential distribution with the count decreasing as the response time increases. Also the average response time for this session is 19.3 ms. Also for 2 node executions we see the same results.

7 Validation

The implementation is valid if the end result is right the messages reach nodes they suppose to. To check that we store all generated messages from each node and all received messages that we are the named receiver. And after the communication we send each message from the receiver back to the source. The source checks all if all its generated messages have come back and if so we have a successful communication. For the validation communication we use straight communication from receiver to server. We could also use the same connection network doing the same procedure described above but exchanging the receiver and producer tags in the message.