Relazione per il progetto di Sistemi Embedded ed IoT "Smart Dam"

A.A 2020/2021

Alessandro Becci(alessandro.becci@studio.unibo.it)
Matricola 0000873910

Università degli studi di Bologna
Campus di Cesena
Corso di Ingegneria e Scienze Informatiche

Scelte libere studente

Per quanto riguarda il sistema "remote-hydrometer" ho utilizzato una architettura piuttosto semplice, in quanto il task che l'assignment richiedeva non era di elevata complessità.
Ho scelto di utilizzare la libreria "Ticker" per generare delle chiamate a funzione periodiche e di non creare un sistema più complesso.

Per il formato delle comunicazioni ho utilizzato quasi sempre il formato JSON, per riuscire ad avere i dati necessari nella forma più semplice possibile. Solo in un caso (Modalità Manuale), quando la comunicazione era molto semplice, ho usato delle semplici stringhe.

Il sistema "Dam Dashboard" è stato realizzato in Java, sfruttando la libreria VertX per l'implementazione di un WebClient.

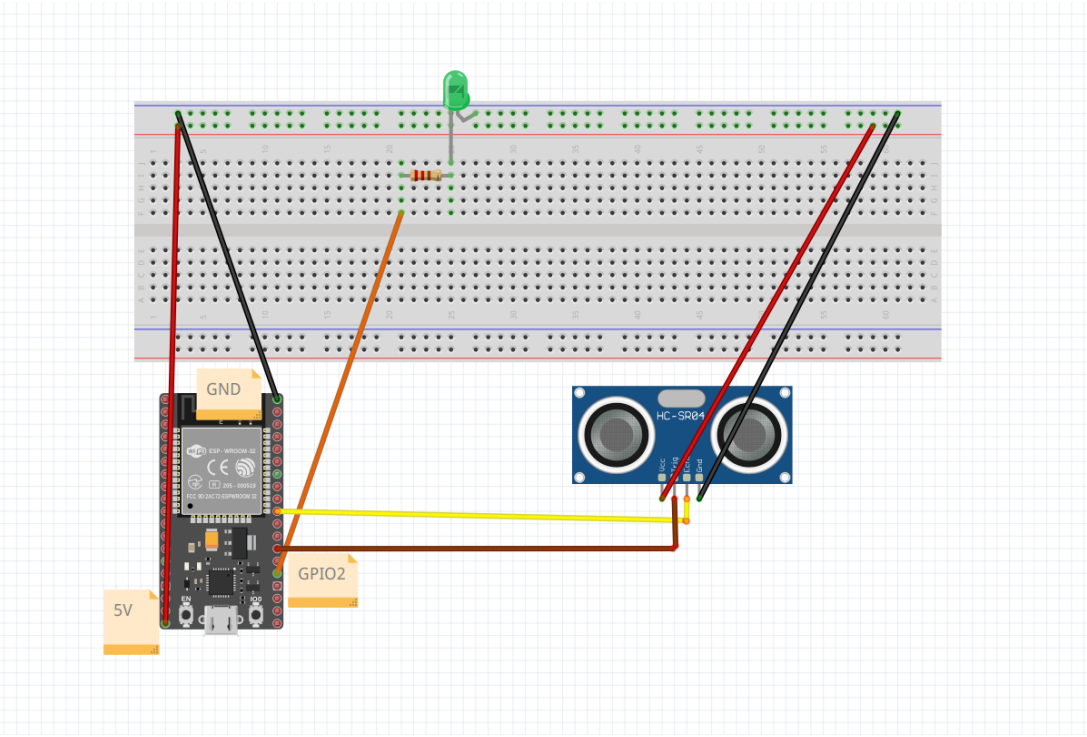
Questa parte dell'assignment:
“Quando il sistema si trova in stato PRE-ALLARME, oltre allo stato, viene visualizzato l’andamento del livello L nel tempo, considerando le ultime Nril rilevazioni.”

Ho interpretato questa parte salvando i dati ogni qualvolta il sonar rileva uno stato di PreAllarme/Allarme in un database. Quando il sistema passa dallo stato Normale allo stato di PreAllarme vengono messe a schermo le ultime N rilevazioni.

Per il sistema "Dam Controller" ho utilizzato una macchina a stati asincrona ad eventi.

Remote Hydrometer

Schema Elettrico



Il pin di ESP32 eroga 3.3V, e quindi la resistenza ideale per il funzionamento del led sarebbe quella da 68 ohm. Utilizzando la resistenza da 220 ohm la luce risulta comunque sufficentemente luminosa.

Per quanto riguarda l'alimentazione del sonar ho utilizzato il pin di ESP che fornisce una alimentazione a 5 volt.

Architettura del Sistema

Il sistema utilizza come parte fondante del proprio funzionamento la libreria "Ticker".
Essa consente di utilizzare i 4 timer hardware presenti sul microcontrollore, assegnando delle funzioni di callback che eseguono quindi periodicamente.
Le funzioni che ho utilizzato sono state 3:

1. stateDet : La funzione setta una variabile booleana per la lettura del sonar e la determinazione dello stato.

2. msgNotAlarm : La funzione controlla lo stato, e in base ad esso setta una variabile booleana per decidere l'invio dei dati(PreAllarme) o dello stato(Normale).
3. MsgAlarm : La funzione controlla lo stato e se ci si trova in stato di allarme setta una variabile booleana per l'invio dei dati(Allarme).

La gestione del Led viene eseguita nel loop dell'applicazione, come tutta la parte di messaggistica e rilevazione. Le funzioni cambiano soltanto lo stato di alcune variabili, in modo tale che la loro esecuzione sia la più rapida possibile.

I messaggi vengono mandati attraverso una connessione HTTP, e in formato JSON.
Ho utilizzato la libreria ArduinoJSON per questa parte.
Il messaggio nel caso dei "Data"(inviati in Allarme/PreAllarme) è composto da type, state e distance.

Nel caso degli "State" vengono inviati solo type e state.
Ho posto una particolare attenzione all'invio delle frequenze(tempi) di aggiornamento, in parole povere la frequenza di invio dei messaggi in modalità di Allarme, e in modalità di non Allarme.

Il JSON dei tempi è strutturato da type, prealarm_time e alarm_time.

Quando il sistema riconoscerà che è presente una connessione attiva, manderà le frequenze, se la connessione dovesse interrompersi, quando diventerà di nuovo attiva, verranno reinviate.
Questo è stato pensato affinchè tutto il sistema conosca questi dati, che vengono decisi su ESP. In questo modo, la Dashboard riesce a chiedere i dati alla frequenza corretta, riuscendo a raggiungere un grado di reattività migliore.

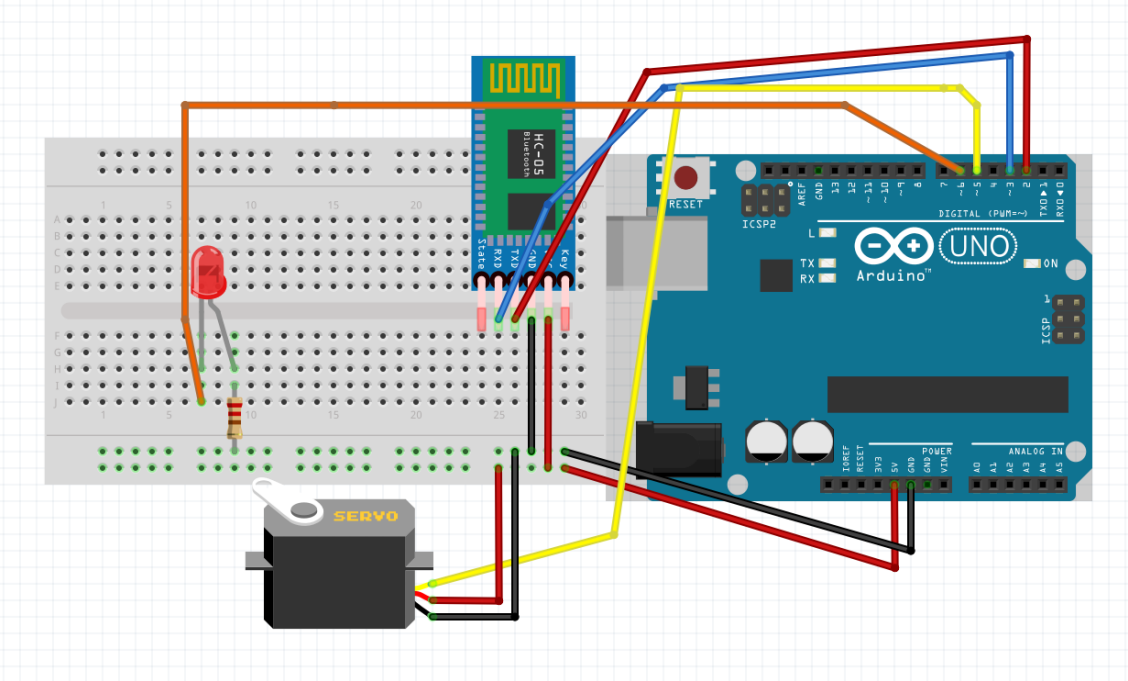
Considerazioni Personali

Ho utilizzato frequenze di funzionamento molto inferiori a quelle specificate nell'assignment, perchè mi consentivano di testare più velocemente le varie parti.
Il vero limite l'ho trovato nel Sonar, che ha bisogno di essere operato ad una frequenza non troppo bassa. Calcolo quindi lo stato

ogni 0.5 secondi, che comunque nel caso di un fiume ho ritenuto essere più che sufficiente.
Avrei potuto creare un sistema più complesso, basato su macchine a stati finiti coordinate da uno scheduler, ma ho preferito optare per la semplicità, sfruttando le potenzialità hardware(4 Timer) di ESP32.
Avrei potuto utilizzare un timer in meno se fossi riuscito a cambiare la frequenza con la quale viene eseguita la funzione per il PreAllarme/Allarme.
Non ho utilizzato Ngrok per avere un indirizzo pubblico, ma ho preferito utilizzare l'indirizzo IP locale del mio pc. Questo mi ha consentito di aggirare le limitazioni che Ngrok mette a chi lo usa gratuitamente(40 messaggi al minuto).

Dam Controller

Schema Elettrico



Architettura del Sistema

Il sistema si basa su un modello di macchina a stati finiti asincrona, in questo caso triggerata da Eventi.
Gli Eventi vengono generati dalle classi deputate alla ricezione dei messaggi dalla porta Seriale, e dal modulo Bluetooth.
Una volta che un evento viene generato, la macchina asincrona viene notificata e l'evento viene gestito.

Sono partito dai template forniti dal prof. Ricci per lo sviluppo delle macchine a stati.
Nel loop() principale del sistema ogni macchina asincrona controlla se ci sono eventi da gestire.

Ho utilizzato due macchine asincrone.
La *MyAsyncFSM* si occupa del funzionamento di tutto il sistema, fatta eccezione per il blinking del led.

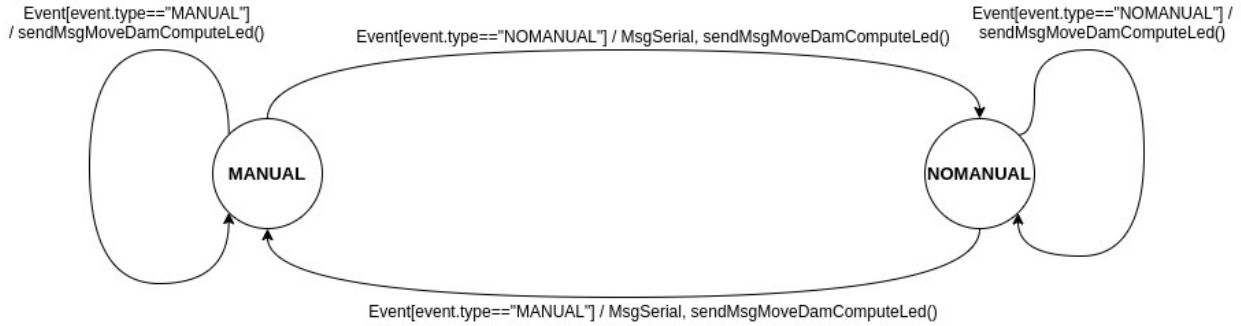


Diagramma degli stati della MyAsyncFSM

All'interno della funzione *sendMsgMoveDamComputeLed()* vengono gestiti diversi aspetti.
In particolare:

- Accensione Led(On, Off, Blinking).
- Gestione dell'apertura della diga(servo) in NOMANUAL e MANUAL
- Aggiornamenti bluetooth al Dam Monitor

Per quanto riguarda il Servo ho deciso di utilizzarlo tramite la libreria "ServoTimer2" in modo da evitare conflitti con Timer1.

Quando si entra in modalità NOMANUAL/MANUAL viene notificato il Dam Service tramite la porta seriale, come quando si cambia l'apertura della diga.
I messaggi inviati in questi casi non sono stati codificati tramite JSON per via della loro semplicità.

I messaggi dalla porta seriale che possono arrivare al Dam Controller sono tutti relativi allo stato, alla distanza, e all'apertura della diga. In questo caso viene utilizzato il formato JSON per via dei dati più "complessi".
Ho utilizzato la funzione *SerialEvent* per avere i messaggi nel momento della loro ricezione: tale funzione viene operata automaticamente da Arduino alla fine di ogni loop().

Nel caso invece dei messaggi ricevuti sul modulo bluetooth ho usato la libreria SoftwareSerial per utilizzare la comunicazione seriale su pin digitali.
Ho utilizzato una funzione *BtSerialEvent* simile a quella utilizzata per la seriale, ma in questo caso ho dovuto inserirla manualmente alla fine del loop(). I dati verso il Dam Monitor sono stati inviati in formato JSON, mentre quelli ricevuti non lo sono(motivi analoghi a quelli già spiegati).

L'altra macchina a stati finiti utilizzata è stata la *LedAsyncFSM*, che si occupa del blinking del led.
Vengono mandati eventi di tipo *LedEvent* da un timer(implementato con Timer1), che viene comandato tramite la *MyAsyncFSM*.
Esso si può disattivare, quando il sistema non è in allarme, e può essere reattivato quando ce n'è bisogno.
Ogni volta che arriva un evento sulla macchina asincrona, il led viene spento se è acceso, e viceversa.
In questo modo, avendo eventi inviati regolarmente dal timer, viene creato il blinking del led.

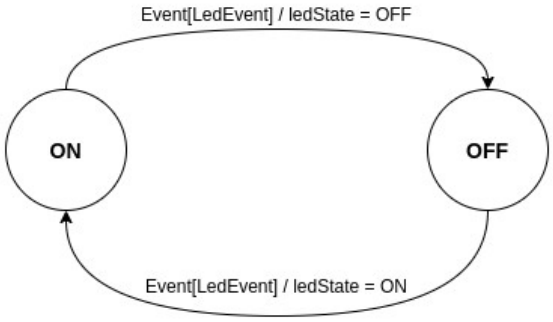


Diagramma degli stati della LedAsyncFSM

Dam Service

Ho gestito questa applicazione prevalentemente utilizzando la libreria VertX, creando differenti verticle a seconda degli usi.

In particolare ho creato un verticle per:
- Comunicazione Seriale(*SerialVerticle*)
- Comunicazione HTTP(*InternetVerticle*)

Il verticle dedicato alla comunicazione seriale utilizza la classe *SerialCommChannel*(fornita dal prof.Ricci) per ricevere e mandare messaggi attraverso la porta seriale.

La comunicazione HTTP è invece affidata al *InternetVerticle* che riceve i dati, ma si mette anche in condizione di ricevere delle richieste GET da parte della Dam Dashboard.

Le informazioni ricevute sono poi memorizzate all'interno del *Model*, che si occupa anche di notificare la porta seriale e di popolare il database.
È stato creata appunto una base di dati per memorizzare in maniera persistente i dati in modalità di PREALLARME/ALLARME.
Il database viene gestito attraverso librerie di VertX ed è stato realizzato utilizzando MYSQL.
Allo scopo di mantenere tutte queste parti collegate fra loro, ho utilizzato la classe *MainController*.

Dam Dashboard

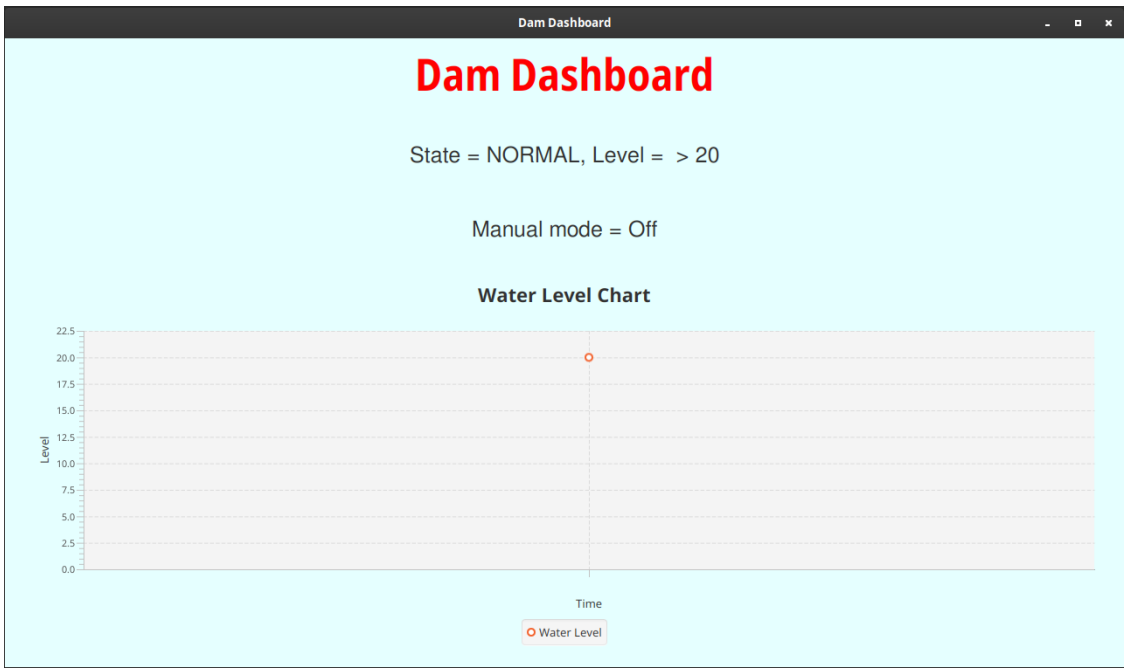
Ho realizzato la dashboard in Java, utilizzando la libreria grafica JavaFX.

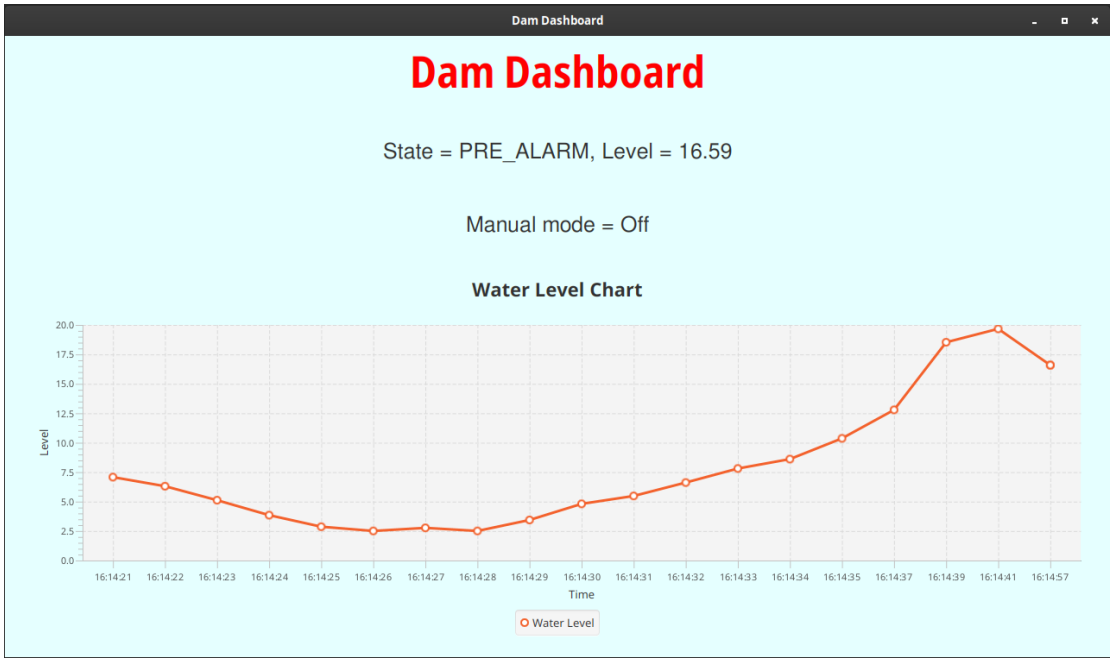
Il sistema richiede i dati al Dam Service(in particolare, al server HTTP *InternetVerticle*) con una certa frequenza, e quindi aggiorna l'interfaccia grafica.

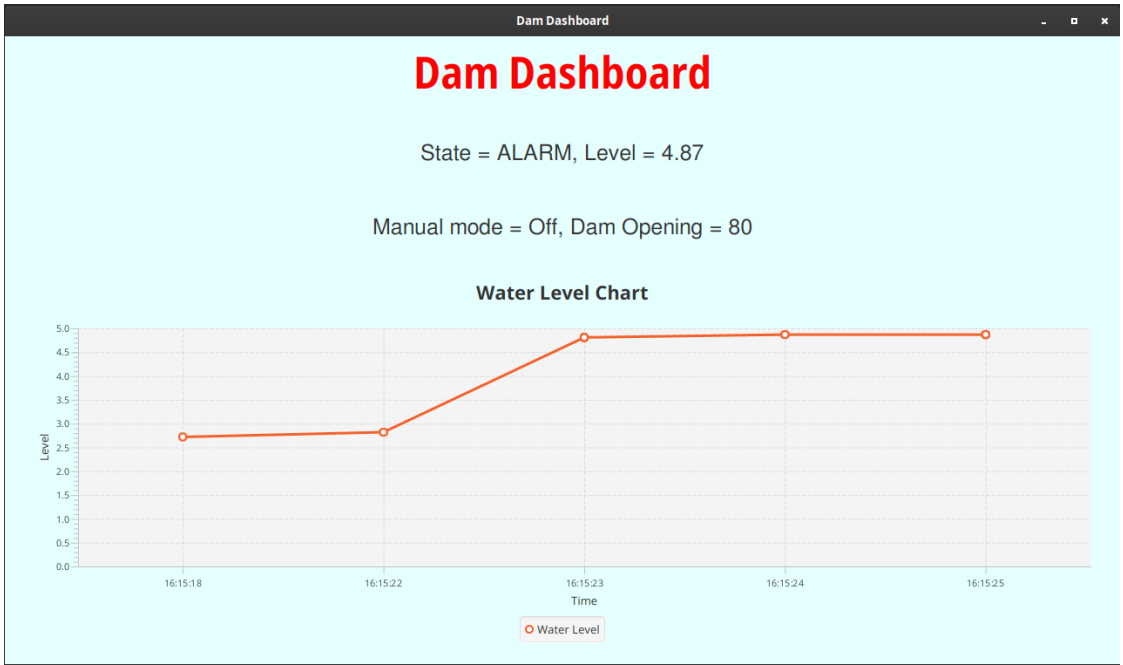
Nello specifico, quando la dashboard viene avviata, viene fatta una richiesta GET per ottenere le frequenze dal Dam Service: finchè esse non vengono ricevute il sistema non esegue richieste.
Grazie alle frequenze ottenute, la dashboard esegue delle richieste GET regolate coi tempi di invio dei dati da parte di ESP negli stati di NOT_ALARM/ALARM.
Allo scopo di eseguire tali richieste ho utilizzato un Thread, nello specifico uno ScheduledExecutor.

Una volta che i dati sono stati ottenuti, viene notificato il controller della View che quindi aggiorna i dati visualizzati a schermo.

Seguono alcuni screenshot dell'interfaccia grafica.


Dam Dashboard
Dam Dashboard
State = NORMAL, Level = > 20
Manual mode = Off
Water Level Chart
Level
Time
Water Level

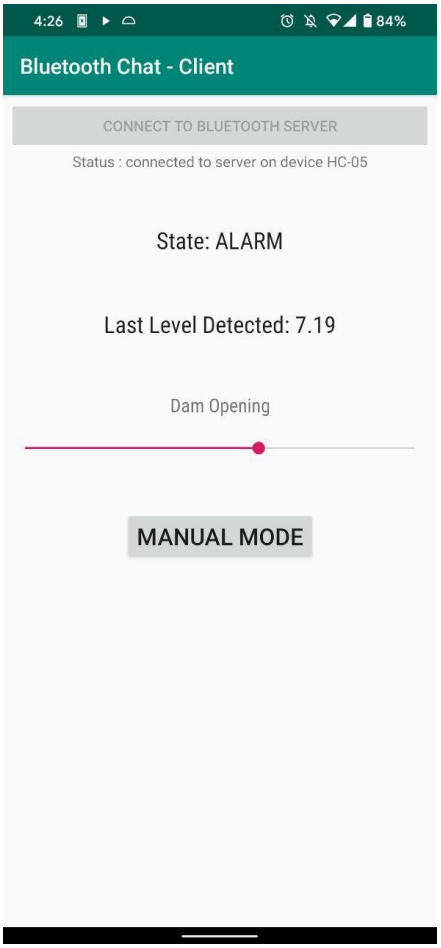

Dam Dashboard
Dam Dashboard
State = PRE_ALARM, Level = 16.59
Manual mode = Off
Water Level Chart
Level
Time
Water Level


Dam Dashboard
Dam Dashboard
State = ALARM, Level = 4.87
Manual mode = Off, Dam Opening = 80
Water Level Chart
Level
Time
Water Level

Dam Mobile

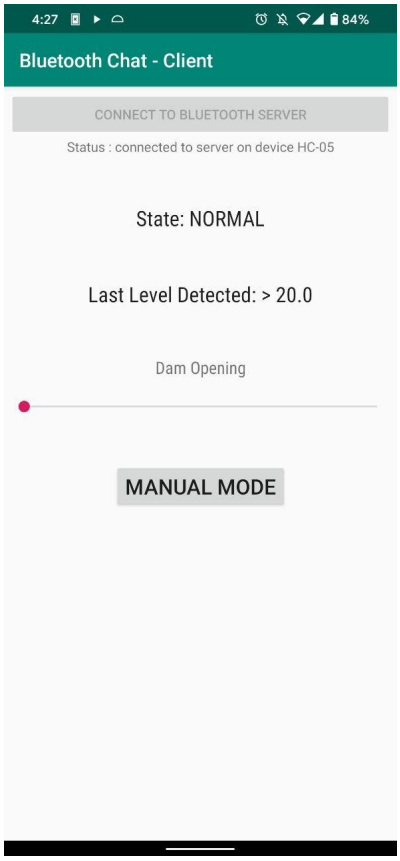
L'applicazione android è stata creata partendo dai template forniti dal prof. Croatti.

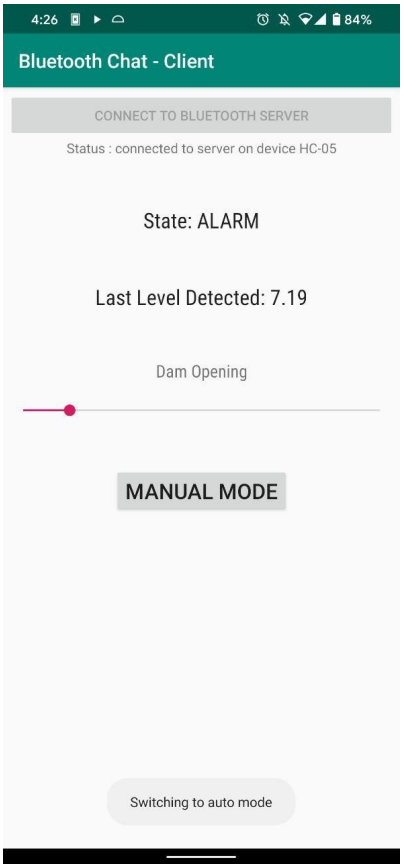
La sua architettura è la medesima, ho solo aggiunto delle TextView per la visualizzazioni dei dati, e una SeekBar che funge sia da visualizzazione che da selezione dell'apertura della diga.

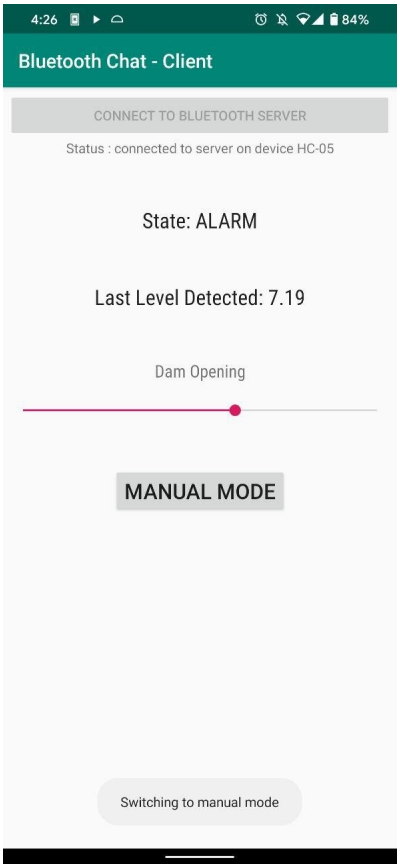


Quando viene selezionata la ManualMode, la SeekBar si "sblocca" e si può selezionare l'apertura. Quando la modalità è quella automatica, la barra è bloccata e serve solo a rappresentare l'apertura.
Viene mostrato un Toast alla pressione del tasto.

Seguono alcuni screenshot dell'applicazione.


4:27
84%
Bluetooth Chat - Client
CONNECT TO BLUETOOTH SERVER
Status : connected to server on device HC-05
State: NORMAL
Last Level Detected: > 20.0
Dam Opening
MANUAL MODE


4:26
84%
Bluetooth Chat - Client
CONNECT TO BLUETOOTH SERVER
Status : connected to server on device HC-05
State: ALARM
Last Level Detected: 7.19
Dam Opening
MANUAL MODE
Switching to auto mode


4:26
84%
Bluetooth Chat - Client
CONNECT TO BLUETOOTH SERVER
Status : connected to server on device HC-05
State: ALARM
Last Level Detected: 7.19
Dam Opening
MANUAL MODE
Switching to manual mode