Smart Dam

Relazione terzo Assignment SEIOT

Luca Fabri 0000892878

23 maggio 2021

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Indice

1 Aspetti generali **2**
1.1 Scelte effettuate 2

2 Remote Hydrometer **3**
2.1 Architettura e sviluppo 3
2.1.1 I Task 3
2.2 Schema circuito 5

3 Dam Service **6**
3.1 Architettura e sviluppo 6

4 Dam Controller **8**
4.1 Architettura e sviluppo 8
4.2 Schema circuito 10
4.3 Considerazioni 10

5 Dam Dashboard **11**
5.1 Sviluppo 11

6 Dam Mobile **13**
6.1 Sviluppo 13
6.1.1 Connessione bluetooth 13
6.1.2 Display, ricezione messaggi 13

Capitolo 1

Aspetti generali

1.1 Scelte effettuate

Per gli aspetti riguardo la comunicazione ho deciso di utilizzare tra Remote Hydrometer e Dam Service il protocollo HTTP. Questo perchè l'assignment è stato pensato per contenere un unico Dam Service e un protocollo publish/subscribe come MQTT risulterebbe poco valorizzato in quanto in questo caso ci sarebbe un unico subscriber. A discapito di ciò, Remote Hydrometer risulterà meno efficiente ma comunque adatto alle nostre esigenze.

Dam Mobile comunica solamente via Bluetooth con Dam Controller, che tra le altre cose inoltra i messaggi provenienti da Dam Mobile a Dam Service e viceversa.

Per favorire l'interoperabilità, tutti i messaggi scambiati tra i sottosistemi sono in formato json.

Capitolo 2

Remote Hydrometer

2.1 Architettura e sviluppo

Per Remote Hydrometer ho utilizzato un'architettura a task basati su macchine a stati finiti sincrone.

Per gestire i periodi dei vari task ho scelto la libreria Ticker, disponibile per il chip ESP8266 a mia disposizione. In particolare, il `tick()` di ogni task viene eseguito ad ogni ciclo del rispettivo timer.

2.1.1 I Task

I task sono:

- `ControllerTask`: Per quanto riguarda le letture delle distanze dal sonar, queste vengono effettuate ogni secondo da questo task, indipendentemente dalla frequenza di invio delle rilevazioni; in questo modo, i cambiamenti di stato vengono catturati più velocemente e, a fronte di questo, `ControllerTask` imposta un flag a true che indica che un messaggio è pronto per essere spedito. Il flag per l'invio del messaggio via HTTP a Dam Service è necessario poichè `ControllerTask`, eseguendo il `tick()` all'interno di un ISR, risulterebbe appesantito da un eventuale HTTP POST. Perciò, l'idea è che appena un messaggio è pronto per essere spedito, viene settato a true il flag `msgReady` e il `loop()` provvederà all'invio. Al cambiamento di stato, oltre a settare il flag a true, questo task gestisce gli altri, facendone l'attach o il detach quando necessario.

  Gli stati della sua FSM sono: NORMAL, PRE_ALARM, ALARM, e cambiano a fronte della distanza letta dal sonar.

- `BlinkLedTask`: questo task, come dal nome, gestisce il blink del led. Il suo attach viene effettuato da `ControllerTask` quando il suo stato diventa PRE_ALARM. Gli stati di questo task sono: BL0, BL1, BL2. BL0 corrisponde al task quando è detached e gli altri due quando è attached, in BL1 il led è off, in BL1 on.

- `SendTask`: questo task, come `BlinkLedTask`, viene gestito da `ControllerTask`. Il suo compito è settare il flag `msgReady` a true ad ogni `tick()`. Ne viene fatto l'attach quando `ControllerTask` è in stato PRE_ALARM o ALARM, e ne viene impostato il periodo sempre da quest'ultimo, 10s e 5s rispettivamente.

Per inserire la marcatura temporale della rilevazione da inviare, ho utilizzato un client NTP. Questo client reperisce il timestamp dal server NTP pool.ntp.org a fronte dell'invio di un messaggio.

Di seguito la FSM di Remote Hydrometer:

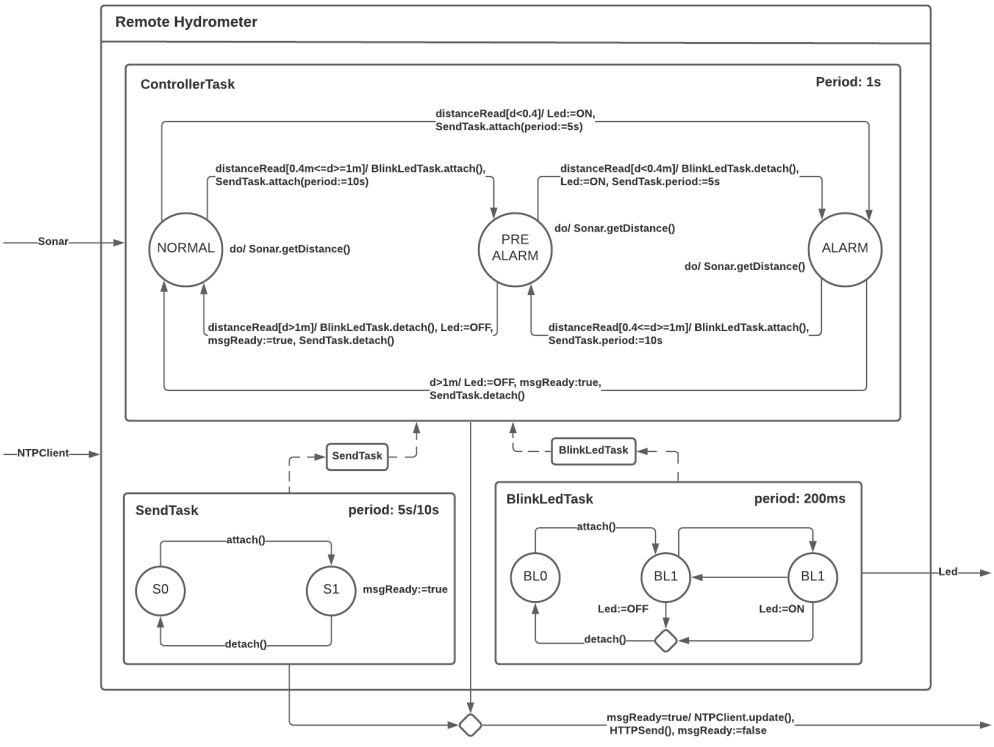

Figura 2.1: Remote Hydrometer FSM

2.2 Schema circuito

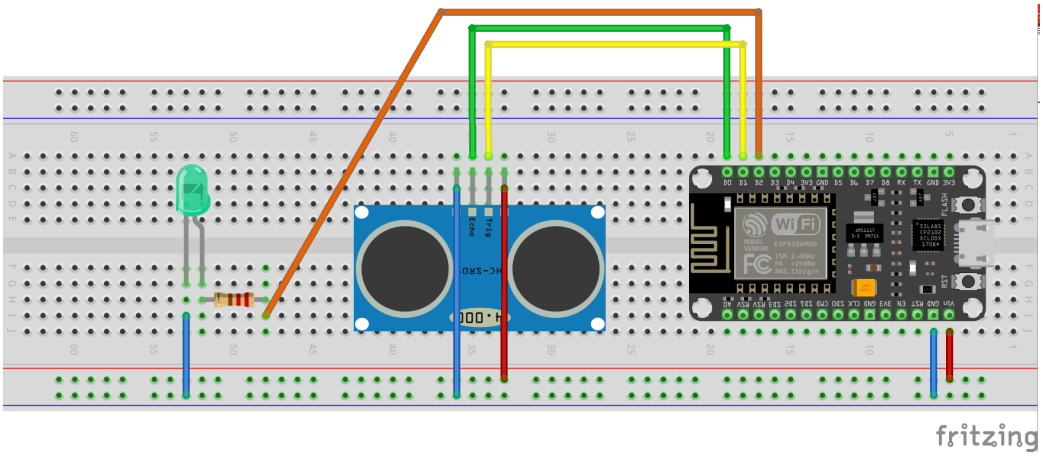

Figura 2.2: Schema del circuito realizzato con Fritizing

Capitolo 3

Dam Service

3.1 Architettura e sviluppo

Per il Dam Service ho utilizzato il pattern MVC:

- Model: unica classe `Data` che mappa la tabella di RiverData DBMS in un oggetto. Utilizzato come parametro per l'inserimento di una riga
- View: reperisce la path assoluta del file da restituire a DD e ne determina il content-type. Viene utilizzata da `HTTPServerController`
- Controllers: sono tre

  1. `HTTPServerController` implementa `AbstractVerticle` di Vert.x e rappresenta il server, memorizza infatti i dati provenienti da RH e restituisce a DD i file per visualizzare la pagina. Contiene inoltre gli handler per l'API e l'oggetto `CommChannel` per inoltrare i messaggi via seriale.

     I messaggi json che vengono spediti a DC non contengono sempre lo stesso quantitativo di informazioni. Ad esempio quando lo stato del fiume è normale il livello dell'acqua non viene calcolato e perciò non viene spedito. Comunque, dal json ricevuto da RH viene determinata l'apertura della diga se la modalità è automatica, il livello dell'acqua se lo stato non è normale e la modalità della diga (poichè può essere cambiata da manuale a automatica se lo stato non è più allarme). Allo stesso tempo viene eliminato il timestamp poichè non è necessaria come informazione. Il json completo (es. stato allarme = 2, modalita automatica = 0) è della forma:

     ```
     {"S":2,"L":4.931,"M":0,"DO":100}
     ```

2. `SerialCommChannelControllerRunnable`. implementa invece un `Runnable` e si occupa di rispondere ai messaggi provenienti via seriale da DC attraverso una `BlockingQueue`
3. `DBMSController`, contiene la connessione al DBMS e il suo scopo è eseguire le query SQL nella tabella Data. Come driver ho utilizzato jdbc.

Per lanciare Dam Service, importare prima sqljdbc4.jar e mysql-connector-java-8.0.21.jar, andare nel package view ed eseguire `RunDamService`. Di seguito il diagramma UML dell'architettura.

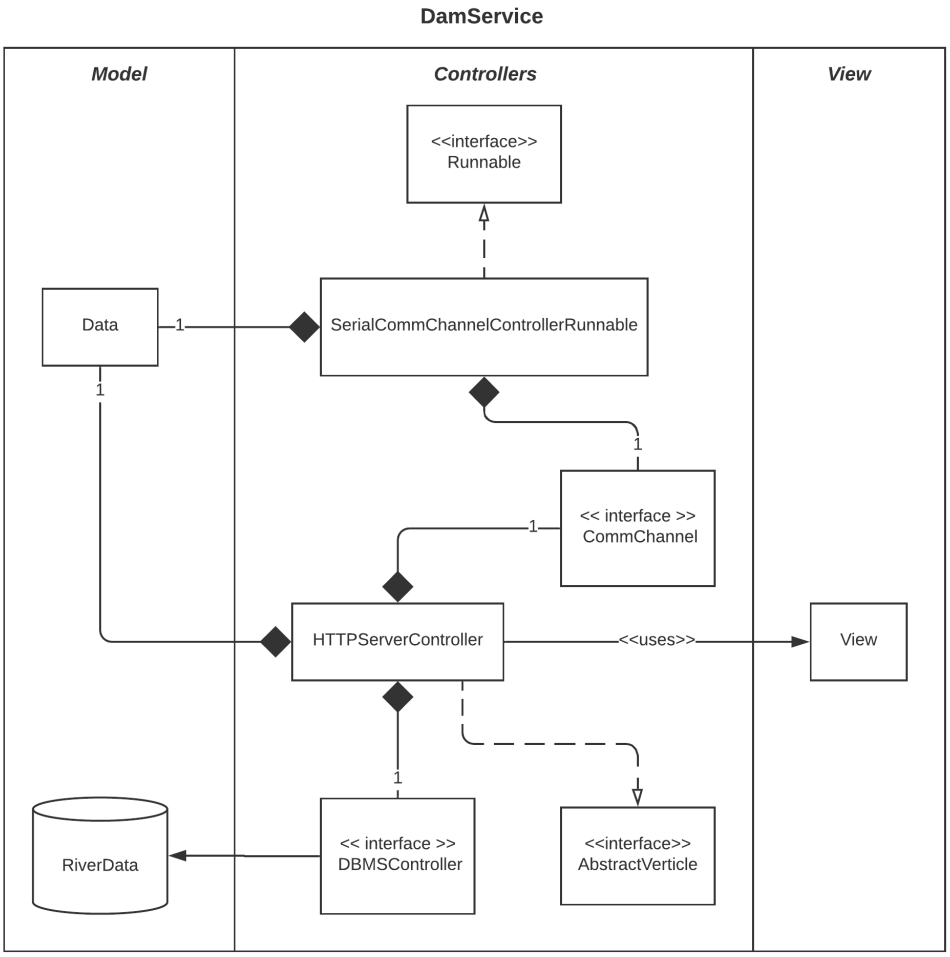

Figura 3.1: Dam Service: diagramma UML

Capitolo 4

Dam Controller

4.1 Architettura e sviluppo

Per Dam Controller ho deciso di utilizzare un approccio basato sull'Asynchronous FSM. Gli eventi corrispondono ai messaggi che arrivano via Serial (da DS) e SoftwareSerial (da DM), che sono gli EventSource; in particolare solamente quelli provenienti da Dam Service fanno cambiare di stato la macchina in quanto corrisponde al "cervello" del sistema.

Siccome Serial e SoftwareSerial non sono in grado di generare interrupts, allora viene effettuato del polling per controllare se un messaggio è arrivato ed eventualmente viene generato l'evento.

Come la funzione built-in di Arduino `serialEvent()`, anche lato bluetooth con SoftwareSerial viene controllato alla fine del `loop()` se c'è un messaggio nel buffer.

Gli stati della FSM sono: NORMAL, PRE_ALARM e ALARM e le tipologie di evento sono: SERIAL_MSG_RECV, e BT_MSG_RECV. Negli stati NORMAL e PRE_ALARM, se avviene un evento BT_MSG_RECV, non interessa il contenuto e viene direttamente inoltrato a DS, mentre se lo stato è ALARM e questo contiene l'apertura della diga, allora viene in aggiunta mosso il servo motore.

Notare che lo spostamento del servo motore quando il sistema è in modalità manuale non viene comunicato da DS dopo aver accettato l'apertura del DM, ma viene automaticamente effettuato quando comunicato da DM, poichè se lo stato è ALARM allora è allarme anche lo stato del fiume.

A seguire lo schema UML della macchina a stati finiti.

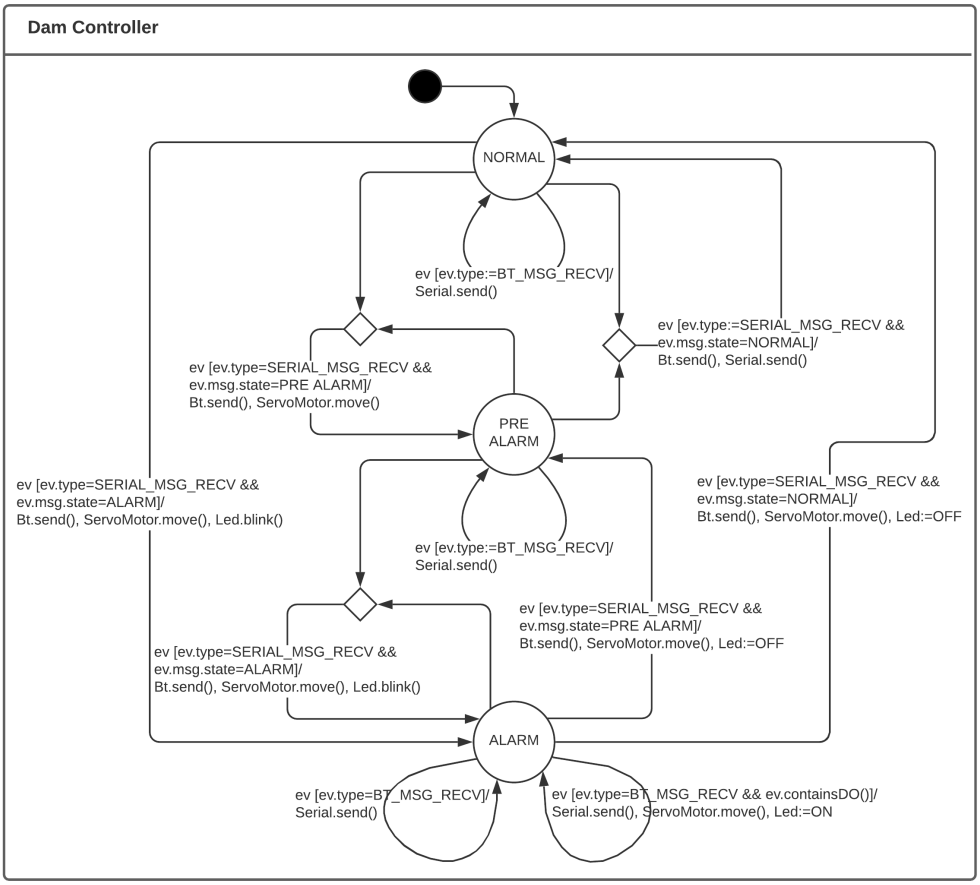

Figura 4.1: DC: diagramma UML a stati

4.2 Schema circuito

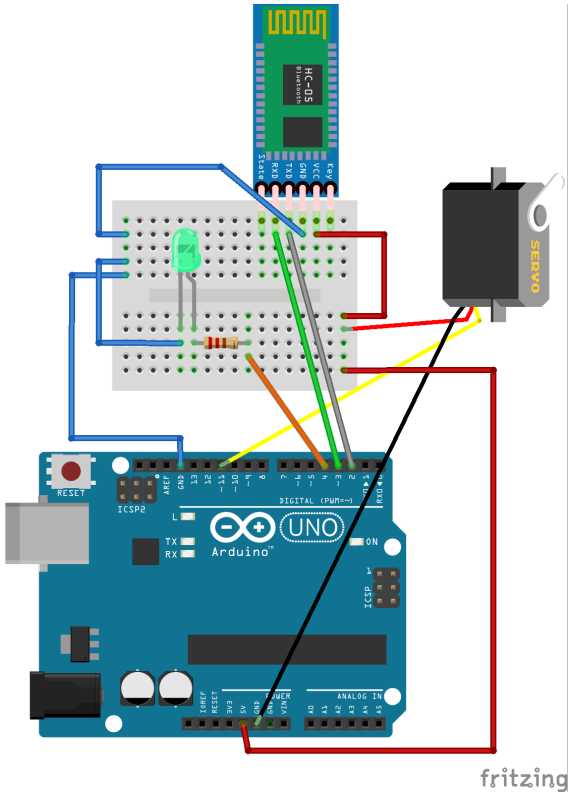

Figura 4.2: Schema del circuito realizzato con Fritizing

4.3 Considerazioni

Durante lo sviluppo di DC ho avuto problemi di gestione della seriale hardware. Inizialmente, optando per un approccio a task basato su macchine a stati finiti sincrone ho notato che i messaggi provenienti da DC non venivano ricevuti interamente. Questo perchè i json che stavo spedendo avevano dimensione maggiore di 64 bytes, la dimensione del buffer della seriale, e Arduino doveva chiamare più volte `serialEvent()` per leggerlo interamente. Siccome `serialEvent()` viene chiamata alla fine del `loop()` e prima veniva eseguita la wait del timer, la parte del messaggio dopo il 64esimo byte andava perso perchè non ricevuto in tempi brevi.

Potendo solamente diminuire la dimensione del json, ho anche preferito cambiare architettura cosicchè il sistema risultasse più reattivo in modo tale da non dover aspettare prima che i messaggi venissero letti.

Capitolo 5

Dam Dashboard

5.1 Sviluppo

Per quanto riguarda il sotto-sistema Dam Dashboard, ho sviluppato una pagina web, utilizzando:

- Bootstrap e CSS per l'aspetto
- Chart.js per il grafico
- JQuery per implementare delle richieste HTTP GET tramite AJAX e di conseguenza aggiornare il grafico.

Le richieste GET vengono inviate periodicamente e, pensando a Dam Dashboard per essere più indipendente possibile dal resto del sistema, utilizzando un qualsiasi periodo T.

L'idea è che non interessa conoscere la frequenza con la quale vengono inviati i dati da RH a DS, perciò, ogni richiesta GET viene effettuata specificando il timestamp dell'ultimo dato ricevuto. L'API, di conseguenza restituisce tutte le rilevazioni successive. Perciò, se il periodo è minore di quello di invio di RH, potrebbero esserci delle GET senza alcun dato, se è maggiore vengono stampate tutte le rilevazioni ricevute.

In questo secondo caso, il grafico risulterà meno real-time ma nessuna rilevazione andrà persa.

Di seguito lo screenshot della pagina, per visualizzarla:
`<DS_Ip>:8080/index.html`

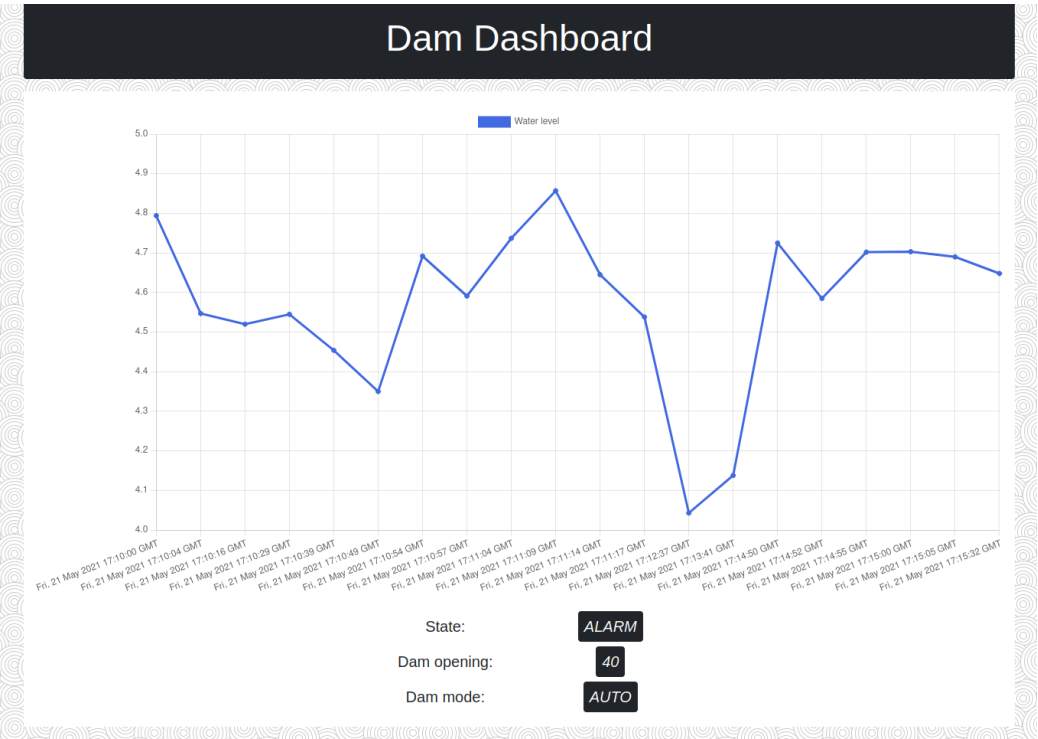

Figura 5.1: Dam Dasboard: index.html

Capitolo 6

Dam Mobile

6.1 Sviluppo

L'applicazione Dam Mobile l'ho realizzata su dispositivo fisico e, come detto precedentemente, comunica solamente via bluetooth con Dam Controller.

6.1.1 Connessione bluetooth

Appena l'app viene connessa al dispositivo bluetooth, viene immediatamente richiesto l'ultimo dato memorizzato dal DS. Questo perchè se lo stato del fiume è normale, si potrebbe anche non ricevere alcun messaggio e il display continuerebbe a indicare "CONNECTED" senza però ottenere alcuna informazione relativa al fiume.

6.1.2 Display, ricezione messaggi

Nel display viene mostrato:

- Stato del fiume
- Apertura della diga
- Modalità della diga
- Eventuale livello dell'acqua

Inoltre, quando il fiume è in stato di allarme, può essere attivata la modalità manuale. Appena selezionata, viene resa interagibile la SeekBar, già impostata al 50%. Questo perchè insieme all'indicazione del cambiamento di modalità in manuale, viene inviata anche l'apertura standard, il 50% e non 0% o 100% per non causare eccessive chiusure o aperture.

Quando invece viene re-impostata la modalità automatica viene solamente spedita tale informazione.

Quando il fiume non è più in stato di allarme, lo Switch e la SeekBar vengono automaticamente disabilitati.

Di seguito sono mostrati gli screenshot di come si presenta l'app.

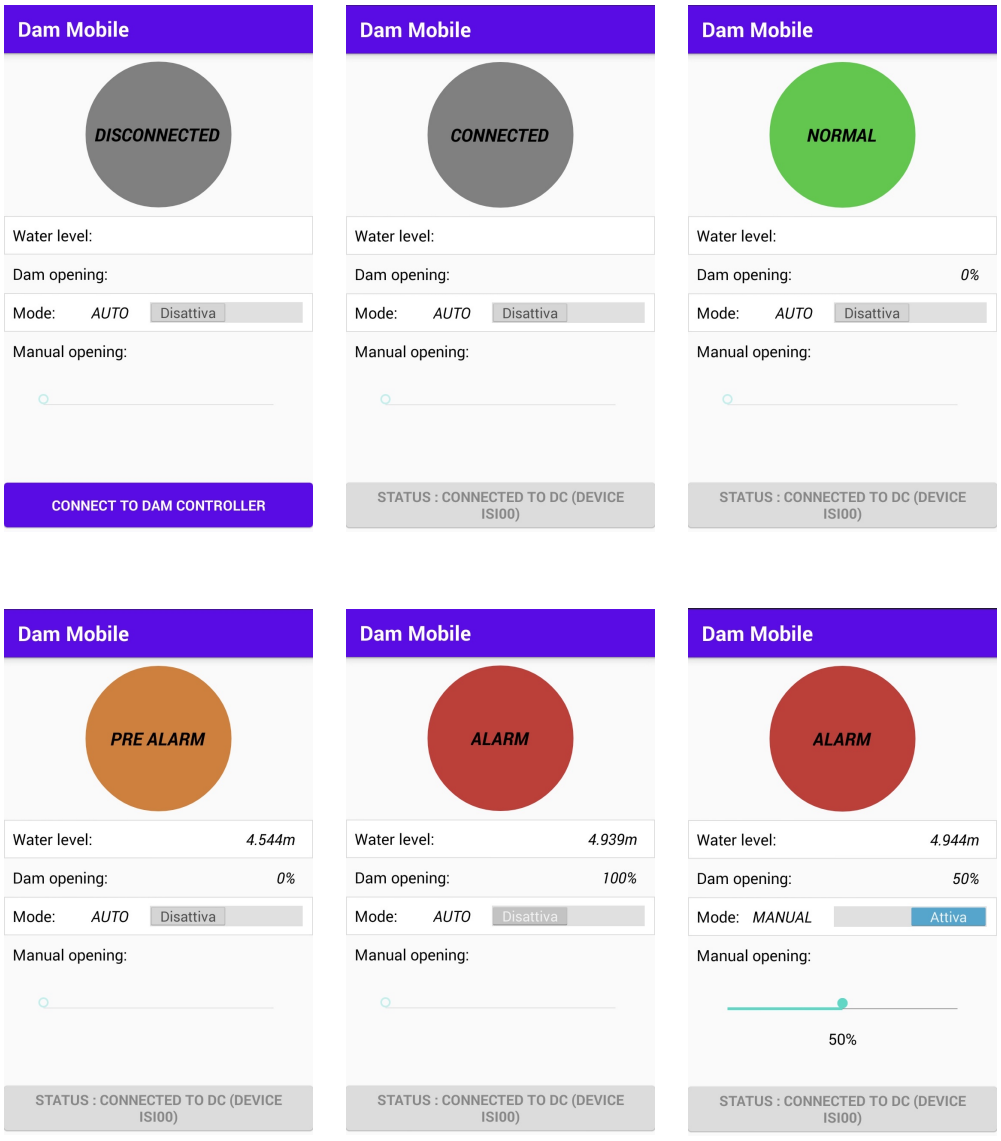

Figura 6.1: Screenshots di Dam Mobile