Creiamo un sensore ZigBee IoT Basato su Greengrass

17 Febbraio 2023 - 11 min. read

AWS Greengrass | Edge Computing | Internet of Things (IoT) | Raspberry | ZigBee

L'Internet of things (IoT) è in continua evoluzione e si parla sempre più spesso di dispositivi connessi per il grande pubblico e nei settori industriale, energetico e Smart City.

Mentre nei prodotti IoT realizzati per gli utenti finali è comune trovare dispositivi direttamente connessi a Internet tramite reti WiFi o 4G, tutta una serie di applicazioni meno visibili ai consumatori possono beneficiare di un'architettura composta da due distinti stack di rete.

In questi casi, un gruppo di dispositivi IoT costituisce una rete locale utilizzando protocolli che possono essere molto diversi da quelli utilizzati su Internet.

Il principale vantaggio è la possibilità di operare reti di dispositivi connessi anche in aree non servite dalla rete 3/4/5G o dove è impossibile connettere ogni dispositivo a Internet. Inoltre, questi dispositivi sono generalmente molto più economici da acquistare e da produrre in serie.

All'interno di queste reti viene spesso individuato un nodo particolare, che funge da coordinatore della rete, e che, se necessario, può fungere da ponte per mettere in comunicazione la rete con il mondo esterno tramite Internet.

Il protocollo ZigBee è uno standard di comunicazione wireless basato sulla specifica IEEE 802.15.4 ed è uno degli stack più popolari per la creazione di una rete locale di dispositivi wireless.

Durante il tempo che ci è concesso dedicare alla ricerca e allo studio di nuovi servizi, abbiamo creato un playground per sperimentare Zigbee e Greengrass. Abbiamo sviluppato un PoC per testare sia la comunicazione Zigbee che vari aspetti di Greengrass.

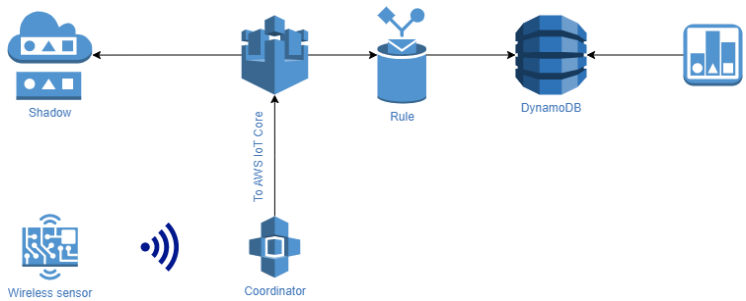
Questo articolo è un diario su come abbiamo creato un dispositivo edge basato su Greengrass e lo abbiamo utilizzato per elaborare i dati raccolti da una rete di sensori in locale, sfruttando le funzionalità principali di AWS IoT per inserire i dati raccolti nella nostra applicazione cloud-native.

Architettura

Per preparare il terreno, abbiamo sviluppato in modo rapido e frugale un sensore wireless utilizzando un modulo XBee, un regolatore di tensione e una fotoresistenza.

XBee utilizza lo standard Zigbee, aggiunge alcune funzionalità e lo racchiude nel suo piccolo pacchetto pulito; inoltre, il modulo è molto più economico e facile da ottenere rapidamente utilizzando i nostri attuali fornitori.

Quindi quello che avevamo in mente era qualcosa di simile al diagramma seguente:



Il sensore rileva la tensione derivata dalla fotoresistenza circa 2 volte al secondo e la trasmette sulla rete ZigBee al nodo coordinatore.

Il nodo coordinatore, che include Greengrass, legge i dati dalla porta seriale, analizza il pacchetto ed estrae il valore trasmesso dal sensore. Poiché il valore è solo un numero intero ed è rumoroso, volevamo bufferizzare ed elaborare i punti dati in locale, calcolare il valore medio per un intervallo di tempo predefinito e inoltrare l'output verso IoT Core.

I dati puliti vengono quindi archiviati sia sullo shadow remoto dell'oggetto, sia in un database per alimentare uno strumento di visualizzazione.

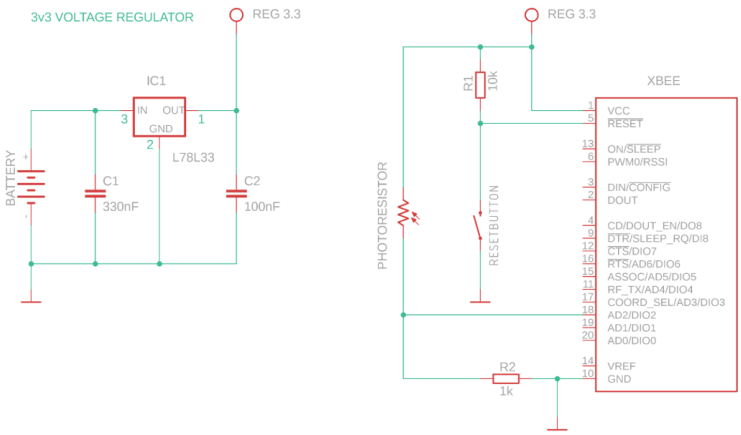
L’hardware

Per il sensore, abbiamo costruito una scheda prototipo con solo uno stadio di alimentazione, realizzata utilizzando un regolatore di tensione e un modulo XBee.

Il modulo ha tutto ciò di cui abbiamo bisogno per soddisfare i nostri requisiti. Ha un ADC integrato e, ovviamente, è in grado di unirsi o formare una rete Zigbee e inviare dati su di essa.

Fortunatamente non è necessario un microcontrollore esterno, il modulo XBee può essere configurato utilizzando XCTU. Questo strumento è un'applicazione gratuita che consente agli sviluppatori di interagire con i moduli Digi RF attraverso un'interfaccia grafica semplice da usare. Ti mostreremo come l'abbiamo utilizzato per configurare i moduli XBee.

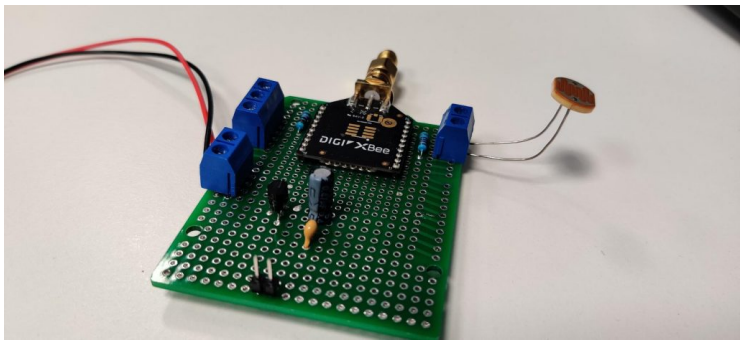
Di seguito è riportato lo schema del sensore, comprendente un'unità di alimentazione (il regolatore di tensione), il partitore di tensione a fotoresistenza e un circuito di ripristino del modulo.



Abbiamo scelto di alimentare la scheda utilizzando una batteria da 9V perché ne abbiamo alcune in ufficio e perché sono compatte, sicure e facili da trovare in qualsiasi negozio.

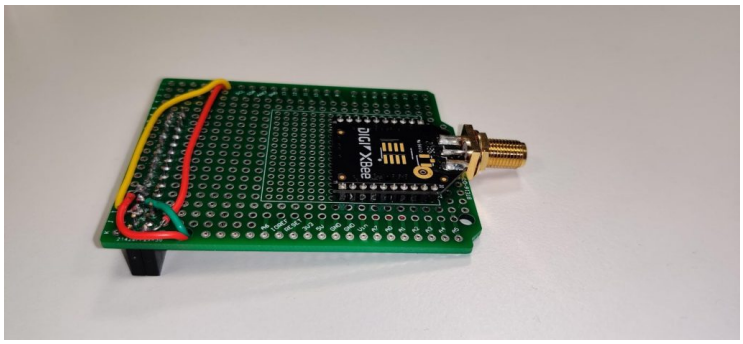
Poiché questo è solo un PoC, non ci andava di costruire qualcosa di alimentato con batterie al litio, e per lo scopo di questo progetto, il nostro regolatore di tensione e una batteria standard da 9V sono stati più che sufficienti.

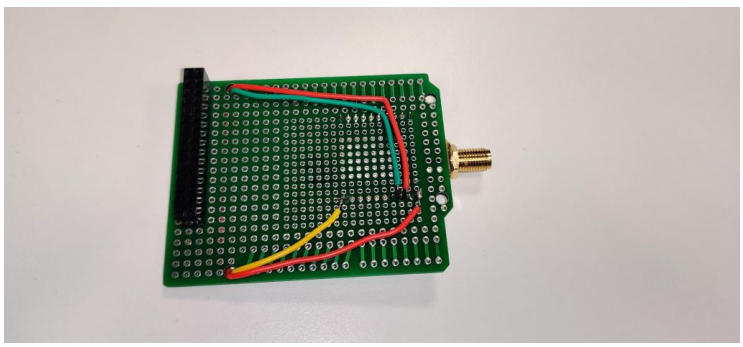
Una volta assemblata, la scheda ha un aspetto simile a questo



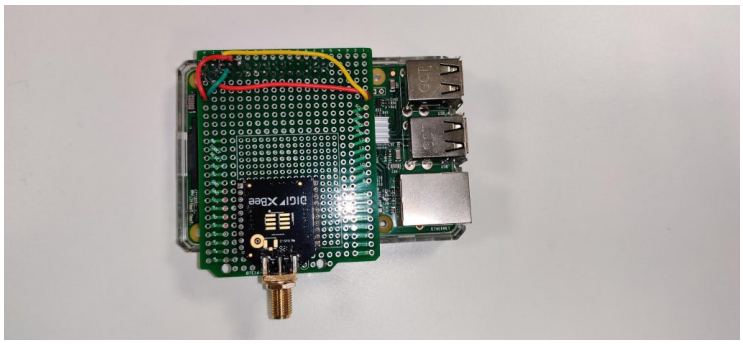
Per il nodo coordinatore, abbiamo utilizzato un Raspberry Pi dotato di sistema operativo Raspbian, Greengrass e un semplice shield che abbiamo realizzato per alimentare e connettere il modulo XBee all'intestazione GPIO. Abbiamo utilizzato la porta seriale hardware integrata.

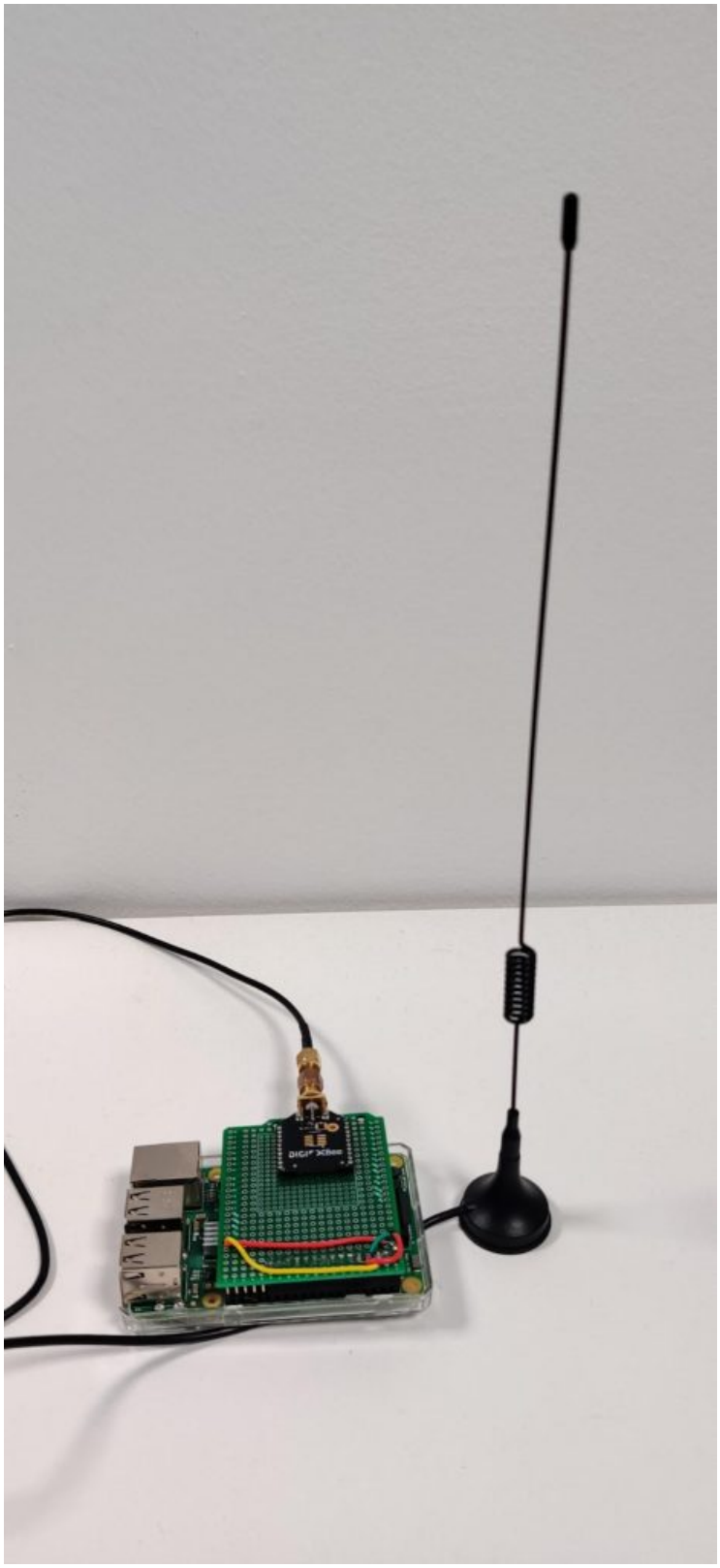
Lo shield





Il coordinatore completamente assemblato





Il software

Dopo aver installato Greengrass sul Raspberry pi (puoi seguire la documentazione ufficiale o il nostro articolo che dovrebbe aiutarti a farlo in pochi passaggi), possiamo sviluppare il nostro codice sorgente.

Il compito del nostro Raspberry pi è raccogliere i messaggi ricevuti tramite ZigBee, analizzarli e infine inviare il valore letto dal sensore tramite MQTT su un named shadow IoT Core. Fortunatamente, come accennato in precedenza, possiamo leggere facilmente questi dati collegandoci al dispositivo seriale Linux e leggendo i byte in ingresso. Il codice è uno script Python piuttosto semplice:

```
import json
import time
from serial import Serial, PARITY_NONE
import awsiot.greengrasscoreipc
import awsiot.greengrasscoreipc.client as client
from awsiot.greengrasscoreipc.model import UpdateThingShadowRequest

LIGHT_TOPIC = "$aws/things/<your_iot_device_name>/shadow/name/<named_
shadow>"
QOS = QOS.AT_LEAST_ONCE
ipc_client = awsiot.greengrasscoreipc.connect()


def send_light_value(payload):
        update_thing_shadow_request = UpdateThingShadowRequest()
        update_thing_shadow_request.thing_name = "p2bc-core-device"
        update_thing_shadow_request.shadow_name = "light"
        update_thing_shadow_request.payload = json.dumps(
        {"state": {"reported": payload}})
        op = ipc_client.new_update_thing_shadow()
        op.activate(update_thing_shadow_request)
        fut = op.get_response()

        result = fut.result(10)
        return result.payload
```

```
with Serial('/dev/ttyS0', 9600, timeout=None, parity=PARITY_NONE, rts
cts=1) as ser:
        while True:
        s = ser.read(1)
        buff = list(s)
        if len(buff) == 0 or buff[0] != 126:
                print("continuing")
                continue
        buff = buff + list(ser.read(2))
        frame_length = buff[1] * 255 + buff[2]
        buff = buff + list(ser.read(frame_length + 1))
        print(f"Buffer: {buff}")
        print("Sending to IoT core")
        light = get_light_intensity(buffer) # The implementation depe
nds on the board you chose
        send_light_value({"light": light})
```

Possiamo impacchettare questo script e distribuirlo grazie alla funzionalità più utile di Greengrass: le distribuzioni automatizzate delle funzione Lambda.

Per fare ciò, dobbiamo solo creare una nuova funzione Lambda nel nostro account AWS e abilitare il versioning.

Dopo averlo fatto, per inviare il pacchetto al nostro core device Greengrass, dobbiamo creare un custom component e collegarlo all'ultima versione della funzione Lambda che abbiamo appena creato. Ricorda che i componenti Greengrass Lambda possono essere di due tipi:

- On-demand, che funzionano proprio come le normali funzioni Lambda; vengono invocati quando si verifica un evento, eseguono il loro codice ed terminano l'esecuzione quando hanno finito.
- Pinned (o long-lived), sono script che possono essere eseguiti senza un input e, se si inserisce il codice al di fuori dell'handler della funzione, può essere eseguito per un periodo di tempo indefinito.

La nostra scelta è stata ovviamente quest'ultima, poiché vogliamo raccogliere e inoltrare continuamente le informazioni lette dal dispositivo seriale.

Infine (questi passaggi sono parecchi, ma dopo averli eseguiti un paio di volte ti abitui al processo e diventa quasi meccanico) devi creare, o aggiornare se hai già eseguito il processo una volta, una distribuzione e selezionare quali componenti distribuire con quella distribuzione. Nel nostro caso, vogliamo impacchettare la nostra funzione Lambda e il componente pubblico aws.greengrass.ShadowManager, che consente ai nostri dispositivi principali di pubblicare messaggi e sottoscriversi allo shadow topic di IoT Core.

Il core device dovrebbe quindi ricevere un nuovo job e installare o aggiornare i componenti selezionati con quella distribuzione. Questo dovrebbe richiedere un paio di minuti e, una volta terminata la fase di installazione, la funzione Lambda dovrebbe avviarsi automaticamente e trasmettere i dati al named shadow selezionato.

E qui sono iniziati i problemi...

GreenGrass e le porte seriali: un rapporto problematico

Dopo aver installato e distribuito la nostra funzione Lambda di prova, qualcosa ha preso una piega inaspettata: nessun dato è stato inviato ad IoT Core.

Dopo una rapida indagine, abbiamo trovato un messaggio di errore abbastanza autoesplicativo nei log del dispositivo (puoi trovarli in /greengrass/v2/logs/)

```
could not open port /dev/ttyS0: [Errno 1] Operation not permitted: '/dev/ttyS0'.
```

Nessun problema, abbiamo pensato: abbiamo commesso un piccolo errore nell'autorizzazione del dispositivo nella configurazione della distribuzione. Abbiamo pensato che il componente Lambda necessitasse di autorizzazioni per accedere al dispositivo seriale e questo può essere impostato nella "Configurazione del processo Linux" nella sezione "Dispositivo", ma impostarlo non ci ha aiutato a risolvere il problema.

Dopo aver ricontrollato il tutto, l'errore continuava a persistere. Anche se la distribuzione specificava di consentire all'utente di accedere ai dispositivi, per

sicurezza abbiamo aggiunto il nostro utente linux greengrass (ggc_user) al gruppo dialout.

Nulla cambiò.

"Operation not permitted" è un errore diverso da "permission denied". La prima volta che ho visto questo tipo di errore è stato quando stavo sperimentando i container Docker, provando a modificare /etc/hosts/: anche se sei root otterrai questo errore.

Dopo aver esplorato la nostra installazione, abbiamo scoperto che greengrass.service è in esecuzione nella propria slice cgroup linux e non ha ottenuto l'autorizzazione per accedere al dispositivo della porta seriale.

Ok, che cos'è una slice?

Molte cose sono cambiate sotto il cofano nei sistemi Linux dopo l'adozione di systemd, anche se, il più delle volte, sono completamente invisibili agli utenti finali e agli amministratori di sistema. I cgroup Linux dovrebbero avere il proprio articolo per spiegarli ma, in breve, sono la tecnologia chiave che consente l'esecuzione di Docker e di altre soluzioni di containerizzazione.

Con i cgroups puoi eseguire processi e controllare l'utilizzo delle risorse (come CPU, memoria e accesso ai dispositivi), anche se un programma viene eseguito come root. Se sei un vecchio amministratore di sistema come me, puoi considerarli come nuovi, più utilizzabili e moderni chroot jail.

Con questo in mente, abbiamo prima cercato di convalidare la nostra ipotesi. Quando un demone si avvia in una slice, puoi trovare la sua configurazione nella directory `/sys/fs/cgroup/system.slice/daemon-name/` (`/sys/fs/cgroup/systemd/system.slice/greengrass.service`)

Un file di sola scrittura chiamato devices.allow, contiene i dispositivi e le autorizzazioni a cui il nostro programma può accedere.

Ogni processo (anche nei container) che viene avviato ottiene la sua directory con il suo file per le autorizzazioni. Ad esempio, la nostra funzione Lambda aveva la sua configurazione in `/sys/fs/cgroup/systemd/system.slice/greengrass.service/D79cAW6fOnnzHB9J4flf5rRNTphoM0KFRxiZY0-89ck` dalla documentazione del kernel Linux abbiamo

visto che scrivendo la stringa "magica" c *:* rwm nella directory del processo all'interno del file devices.allow tutto funzionava correttamente.

"we need a more permanent solution to our problems" - Caiaphas - Jesus Christ superstar

Ok, configuriamo la nostra unità systemd per permettere alle nostre funzioni Lambda di accedere alla porta seriale!

Abbiamo scoperto dalla documentazione di systemd che, aggiungendo la riga `DeviceAllow=/dev/ttyS0 rw` nella nostra unità (/etc/systemd/system/greengrass.service) dovrebbe essere sufficiente.

No, non funziona, abbiamo un altro problema:

```
unable to create start process: failed to run container sandbox: container_linux.go:380: starting container process caused: process_linux.go:545: container init caused: process_linux.go:508: setting cgroup config for procHooks process caused: failed to write "b *:* m": write /sys/fs/cgroup/devices/system.slice/greengrass.service/D79cAW6fOnnzHB9J4flf5rRNTphoM0KFRxiZY0-89ck/devices.allow: operation not permitted.
```

Questo errore sembra ancora più strano: significa che il processo di Greengrass sta provando a dare accesso per i block device alla nostra Lambda (mentre la nostra porta seriale è un character device).

Per farla breve: dopo aver giocherellato con le autorizzazioni, non abbiamo trovato nulla che potesse consentire l'esecuzione del nostro processo, quindi abbiamo dovuto implementare una soluzione hacky. Fateci sapere nei commenti se avete suggerimenti.

Sistemare tutto al volo

Quindi, sapevamo che l'impostazione manuale delle autorizzazioni di processo avrebbe risolto il nostro problema. Abbiamo solo bisogno di automatizzare il processo di concessione delle autorizzazioni alla nostra Lambda una volta che è stata avviata.

Un piccolo script con inotifywait fa il trucco:

```
#!/bin/bash
inotifywait --monitor /sys/fs/cgroup/devices/system.slice/greengrass.
service --event create |
while read dir action file; do
        echo "**** $dir $action $file ***"
        if [ $action == "CREATE,ISDIR" ]; then
                sleep 1
                echo "c *:* rwm" >  $dir/$file/devices.allow
        fi
done
```

Mettendo questo file in /greegrass/v2/filemonitor.sh e aggiungendo una unit systemd (ad esempio /etc/systemd/system/filemonitor.service) con questo contenuto

```
[Unit]
Description=Filemonitor
After=greengrass.service
[Service]
Type=simple
PIDFile=/greengrass/v2/alts/filemonitor.pid
RemainAfterExit=no
Restart=on-failure
RestartSec=10
[Install]
WantedBy=multi-user.target
```

ha risolto il problema! Non dimenticatevi di attivare il servizio con systemctl daemon-reload; `systemctl enable filemonitor`
Questo script monitora la directory cgroup Greengrass. Quando un nuovo processo si avvia, scriverà nel suo file device.allow i permessi, permettendo così al nostro codice di aprirsi e leggere finalmente dal dispositivo seriale.

Dopo aver risolto questo problema (che ci ha richiesto più tempo del previsto), siamo finalmente riusciti a leggere i dati dalla porta seriale e quindi a inoltrarli a IoT Core tramite il nostro componente Lambda. Non appena i nuovi dati arrivano nel cloud

AWS, una IoT Rule salva la nuova versione dello shadow nella nostra tabella DynamoDB.

Cosa ci portiamo a casa?

Ci piace Greengrass, ci piace davvero, perché semplifica molto il processo di implementazione del nostro codice sui nostri dispositivi edge.

Vediamo tutti i vantaggi di avere un orchestratore che è stato sviluppato e testato a fondo, ma a volte può intralciarti quando vuoi eseguire attività molto semplici (come leggere da una porta seriale, cosa che accade spesso nel mondo IoT!), e finisci per dedicare molto alla ricerca della causa e delle possibili soluzioni (o se sei fortunato e hai un Damiano come noi in beSharp, puoi accorciare molto il tempo di ricerca su Google con il suo aiuto).

Le nostre ricerche ci hanno permesso di apprendere molte cose su Linux e Greengrass, ad esempio abbiamo appreso che nella prima versione di Greengrass (v1) c'era effettivamente un componente che ti consente di connetterti a dispositivi seriali in modo più sicuro con poche configurazioni sul lato IoT, senza alcun minaccioso script bash eseguito sul dispositivo, ma al giorno in cui siamo scrivendo questo articolo, questa funzionalità non è stata portata alla versione più recente (v2).

Ci piacerebbe trovare una soluzione più elegante, quindi se sei più fortunato o conosci meglio di noi questo argomento, lascia un commento nella sezione sottostante per aiutare noi e tutti gli altri sysops a risolvere questo problema!

Mattia Costamagna

Ingegnere DevOps e sviluppatore cloud-native @ beSharp. Adoro passare il mio tempo libero a leggere romanzi e ascoltare musica rock e blues degli anni '70. Sempre alla ricerca di nuove tecnologie e framework da testare e utilizzare. La birra artigianale è il mio carburante!

Copyright © 2011-2023 by beSharp spa - P.IVA IT02415160189