Creazione e distribuzione di un pacchetto IOx Docker per l'architettura ARM IR1101

Sommario

Introduzione

In questo documento viene descritto come preparare, creare e distribuire un pacchetto IOx basato su Docker per il gateway Internet of Things (IoT) basato su ARM IR1101.

Prerequisiti

Requisiti

Cisco raccomanda la conoscenza dei seguenti argomenti:

- Linux
- Contenitori
- IOx

Componenti usati

Le informazioni fornite in questo documento si basano sulle seguenti versioni software e

hardware:

- IR1101 raggiungibile su SSH (Secure Shell)
  Indirizzo IP configuratoAccesso al dispositivo con un privilegio di 15 utenti
- Host Linux (per questo articolo viene utilizzata un'installazione minima di Debian 9 (stretch))
- File di installazione del client IOx scaricabili da:
  https://software.cisco.com/download/release.html?mdfid=286306005&softwareid=286306762

Le informazioni discusse in questo documento fanno riferimento a dispositivi usati in uno specifico ambiente di emulazione. Su tutti i dispositivi menzionati nel documento la configurazione è stata ripristinata ai valori predefiniti. Se la rete è operativa, valutare attentamente eventuali conseguenze derivanti dall'uso dei comandi.

Premesse

IR1101 è leggermente diverso rispetto alla maggior parte delle altre piattaforme IOx, in quanto queste sono basate principalmente su x86. IR1101 è basato sull'architettura ARM64v8, pertanto non è possibile distribuire contenitori o pacchetti IOx creati per x86 direttamente sulla piattaforma. Questo documento parte da zero e prepara l'ambiente per la creazione di contenitori Docker basati su ARM64v8 e spiega come costruirli, assemblarli e distribuirli su IR1101 con l'utilizzo di un PC x86.

Ad esempio, viene utilizzato uno script Python di dimensioni molto ridotte, che è un semplice server Web, e viene creato un contenitore Docker per inserirlo in un pacchetto da eseguire su IR1101. L'unica operazione che il server Web eseguirà è l'ascolto su una porta predefinita (9000) e la restituzione di una pagina semplice quando riceve una richiesta **GET**. In questo modo è possibile testare la capacità di eseguire il proprio codice e verificare l'accesso di rete all'applicazione IOx una volta avviata l'esecuzione.

Il pacchetto sarà realizzato dagli strumenti Docker, con l'uso di Alpine Linux. Alpine Linux è un'immagine Linux di piccole dimensioni (circa 5 MB), spesso utilizzata come base per i contenitori Docker.

Poiché la maggior parte dei desktop/notebook/macchine virtuali è basata su x86, è necessario emulare l'architettura ARM64v8 sul computer x86 su cui è costruito il contenitore. Questa operazione può essere eseguita facilmente utilizzando l'emulazione utente QEMU (Quick Emulator). Ciò consente l'esecuzione di file eseguibili in un'architettura non nativa esattamente come verrebbe eseguita sull'architettura nativa.

Configurazione

Parte 1. Generare il pacchetto IOx per IR1101

1. Installare e preparare il client IOx sull'host Linux

Ènecessario **ioxclient** per comprimere il contenitore Docker come un pacchetto IOx una volta generato, quindi prepariamolo prima.

Copiare o scaricare il pacchetto ioxclient. È disponibile all'indirizzo:
https://software.cisco.com/download/release.html?mdfid=286306005&softwareid=286306762.

```
jedepuyd@deb9:~$ scp jedepuyd@192.168.56.101:/home/jedepuyd/ioxclient_1.7.0.0_linux_amd64.tar.gz
.
jedepuyd@192.168.56.101's password:
ioxclient_1.7.0.0_linux_amd64.tar.gz                                  100% 4798KB  75.2MB/s   00:00
```

Estrarre la confezione:

```
jedepuyd@deb9:~$ tar -xvzf ioxclient_1.7.0.0_linux_amd64.tar.gz
ioxclient_1.7.0.0_linux_amd64/ioxclient
ioxclient_1.7.0.0_linux_amd64/README.md
```

Aggiungere il percorso alla variabile **PATH** per renderlo disponibile senza utilizzare la posizione completa. Se si riavvia il computer o si cambia utente, non dimenticare di ripetere questo passaggio:

```
jedepuyd@deb9:~$ export PATH=$PATH:/home/jedepuyd/ioxclient_1.7.0.0_linux_amd64/
```

Avviare **ioxclient** per la prima volta per creare un profilo obbligatorio. Poiché si utilizzerà ioxclient solo per il package del contenitore Docker, i valori possono essere lasciati come predefiniti:

```
jedepuyd@deb9:~$ ioxclient -v
ioxclient version 1.7.0.0
jedepuyd@deb9:~/iox_aarch64_webserver$ ioxclient profiles reset
Active Profile :  default
Your current config details will be lost. Continue (y/N) ? : y
Current config backed up at  /tmp/ioxclient731611124
Config data deleted.
jedepuyd@deb9:~/iox_aarch64_webserver$ ioxclient -v
Config file not found :  /home/jedepuyd/.ioxclientcfg.yaml
Creating one time configuration..
Your / your organization's name :
Your / your organization's URL :
Your IOx platform's IP address[127.0.0.1] :
Your IOx platform's port number[8443] :
Authorized user name[root] :
Password for root :
Local repository path on IOx platform[/software/downloads]:
URL Scheme (http/https) [https]:
API Prefix[/iox/api/v2/hosting/]:
Your IOx platform's SSH Port[2222]:
Your RSA key, for signing packages, in PEM format[]:
Your x.509 certificate in PEM format[]:
Activating Profile  default
Saving current configuration
ioxclient version 1.7.0.0
```

2. Installare e preparare l'ambiente Docker sul computer di compilazione Linux

Questo Docker viene utilizzato per creare un contenitore dall'immagine base Alpine e per includere i file necessari per lo Use Case. I passaggi forniti sono basati sulle guide ufficiali di installazione di Docker Community Edition (CE) per Debian:
https://docs.docker.com/install/linux/docker-ce/debian/

Aggiornare gli elenchi di pacchetti nel computer:

```
jedepuyd@deb9:~$ sudo apt-get update
...
```

```
Reading package lists... Done
```

Installare le dipendenze per utilizzare il repository Docker:

```
jedepuyd@deb9:~$ sudo apt-get install apt-transport-https ca-certificates curl gnupg2 software-
properties-common
Reading package lists... Done
Building dependency tree
...
Processing triggers for dbus (1.10.26-0+deb9u1) ...
```

Aggiungere la chiave GPG (Docker GNU Privacy Guard) come chiave GPG valida:

```
jedepuyd@deb9:~$ curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add -
OK
```

Verificare l'impronta digitale della chiave GPG installata:

```
jedepuyd@deb9:~$ sudo apt-key fingerprint 0EBFCD88
pub   rsa4096 2017-02-22 [SCEA]
      9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88
uid           [ unknown] Docker Release (CE deb) <docker@docker.com>
sub   rsa4096 2017-02-22 [S]
```

Aggiungere il repository stabile Docker:

```
jedepuyd@deb9:~$ sudo add-apt-repository "deb [arch=amd64]
https://download.docker.com/linux/debian $(lsb_release -cs) stable"
```

Aggiornare di nuovo gli elenchi di pacchetti quando si aggiunge il repository Docker:

```
jedepuyd@deb9:~$ sudo apt-get update
...
Reading package lists... Done
```

Docker di installazione:

```
jedepuyd@deb9:~$ sudo apt-get install docker-ce docker-ce-cli containerd.io
Reading package lists... Done
Building dependency tree
...
Processing triggers for systemd (232-25+deb9u9) ...
```

Per poter accedere/eseguire Docker come utente normale, aggiungere questo utente al gruppo Docker e aggiornare l'appartenenza al gruppo:

```
jedepuyd@deb9:~$ sudo usermod -a -G docker jedepuyd
jedepuyd@deb9:~$ newgrp docker
```

3. Installare i pacchetti di emulazione utente QEMU

Dopo aver installato Docker, è necessario installare gli emulatori utente QEMU. Utilizzare l'emulatore QEMU collegato in modo statico dal contenitore Docker in modo da poter eseguire il contenitore per ARM64v8 sul computer Linux basato su x86, anche se il contenitore di destinazione sarà progettato per l'architettura ARM64v8.

Installare i pacchetti:

```
jedepuyd@deb9:~$ sudo apt-get install qemu-user qemu-user-static
Reading package lists... Done
Building dependency tree
...
Processing triggers for man-db (2.7.6.1-2) ...
```

Dopo l'installazione, di seguito sono riportati gli emulatori QEMU collegati in modo statico disponibili in **/usr/bin**:

```
jedepuyd@deb9:~$ ls -al /usr/bin/qemu-*static
-rwxr-xr-x 1 root root 3468784 Nov  8 16:41 /usr/bin/qemu-aarch64-static
-rwxr-xr-x 1 root root 2791408 Nov  8 16:41 /usr/bin/qemu-alpha-static
-rwxr-xr-x 1 root root 3399344 Nov  8 16:41 /usr/bin/qemu-armeb-static
-rwxr-xr-x 1 root root 3391152 Nov  8 16:41 /usr/bin/qemu-arm-static
-rwxr-xr-x 1 root root 2800400 Nov  8 16:41 /usr/bin/qemu-cris-static
...
```

Il primo nell'elenco è quello necessario: arch64 è il nome in codice di ARM64v8 per Linux.

4. Verificare se un contenitore arch64/ARV64v8 è in esecuzione su un computer Linux x86

Dopo aver installato Docker e i necessari file binari QEMU, è possibile verificare se è possibile eseguire un contenitore Docker costruito per ARM64v8 sul computer x86:

```
jedepuyd@deb9:~$ docker run -v /usr/bin/qemu-aarch64-static:/usr/bin/qemu-aarch64-static --rm -
ti arm64v8/alpine:3.7
Unable to find image 'arm64v8/alpine:3.7' locally
3.7: Pulling from arm64v8/alpine
40223db5366f: Pull complete
Digest: sha256:a50c0cd3b41129046184591963a7a76822777736258e5ade8445b07c88bfdcc3
Status: Downloaded newer image for arm64v8/alpine:3.7
/ # uname -a
Linux 1dbba69b60c5 4.9.0-8-amd64 #1 SMP Debian 4.9.144-3.1 (2019-02-19) aarch64 Linux
```

Come potete vedere nell'output, arm64v8 Alpine container è ottenuto e fatto funzionare con l'accesso all'emulatore.

Se si richiede l'architettura del contenitore, è possibile verificare che il codice sia compilato per arch64. Esattamente come l'arco di destinazione per il contenitore deve essere per IR1101.

5. Preparare i file per la creazione del contenitore del server Web Docker

Al termine della preparazione, è possibile creare i file necessari per il contenitore del server Web da eseguire su IR1101.

Il primo file è **webserver.py**, lo script Python che si desidera eseguire nel contenitore. Poiché si tratta solo di un esempio, è ovvio che verrà sostituito con il codice effettivo per l'esecuzione nell'applicazione IOx:

```
jedepuyd@deb9:~$ mkdir iox_aarch64_webserver
jedepuyd@deb9:~$ cd iox_aarch64_webserver

jedepuyd@deb9:~/iox_aarch64_webserver$ vi webserver.py
jedepuyd@deb9:~/iox_aarch64_webserver$ cat webserver.py
```

```
#!/usr/bin/env python
from BaseHTTPServer import BaseHTTPRequestHandler, HTTPServer
import SocketServer
import os

class S(BaseHTTPRequestHandler):
    def _set_headers(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()

    def do_GET(self):
        self._set_headers()
        self.wfile.write("<html><body><h1>IOX python webserver on arm64v8</h1></body></html>")
        logf.write('Got GET\n')
        logf.flush()

def run(server_class=HTTPServer, handler_class=S, port=9000):
    server_address = ('', port)
    httpd = server_class(server_address, handler_class)
    print 'Starting webserver...'
    logf.write('Starting webserver....\n')
    logf.flush()
    httpd.serve_forever()

if __name__ == "__main__":
    log_file_dir = os.getenv("CAF_APP_LOG_DIR", "/tmp")
    log_file_path = os.path.join(log_file_dir, "webserver.log")
    logf = open(log_file_path, 'w')
    run()
    logf.close()
```

Questo codice contiene la logica per la scrittura in un file di log, che sarà disponibile per la consultazione dal responsabile locale.

Il secondo file necessario è Dockerfile. Questo definisce la modalità di creazione del contenitore:

```
jedepuyd@deb9:~/iox_aarch64_webserver$ vi Dockerfile
jedepuyd@deb9:~/iox_aarch64_webserver$ cat Dockerfile
FROM arm64v8/alpine:3.7
COPY qemu-aarch64-static /usr/bin

RUN apk add --no-cache python
COPY webserver.py /webserver.py
```

Il Dockerfile definisce la modalità di generazione del contenitore. Iniziare dall'immagine base Apline per ARM64v8, copiare l'emulatore nel contenitore, eseguire l'apk per aggiungere il pacchetto Python e copiare lo script del server Web nel contenitore.

L'ultima preparazione necessaria prima di poter creare il contenitore consiste nel copiare qemu-arch64-static nella directory da cui verrà generato il contenitore:

```
jedepuyd@deb9:~/iox_aarch64_webserver$ cp /usr/bin/qemu-aarch64-static .
```

6. Creare il contenitore Docker

Una volta completata tutta la preparazione, è possibile creare il contenitore utilizzando il file Dockerfile:

```
jedepuyd@deb9:~/iox_aarch64_webserver$ docker build -t iox_aarch64_webserver .
Sending build context to Docker daemon  3.473MB
Step 1/4 : FROM arm64v8/alpine:3.7
 ---> e013d5426294
Step 2/4 : COPY qemu-aarch64-static /usr/bin
 ---> addf4e1cc965
Step 3/4 : RUN apk add --no-cache python
 ---> Running in ff3768926645
fetch http://dl-cdn.alpinelinux.org/alpine/v3.7/main/aarch64/APKINDEX.tar.gz
fetch http://dl-cdn.alpinelinux.org/alpine/v3.7/community/aarch64/APKINDEX.tar.gz
(1/10) Installing libbz2 (1.0.6-r6)
(2/10) Installing expat (2.2.5-r0)
(3/10) Installing libffi (3.2.1-r4)
(4/10) Installing gdbm (1.13-r1)
(5/10) Installing ncurses-terminfo-base (6.0_p20171125-r1)
(6/10) Installing ncurses-terminfo (6.0_p20171125-r1)
(7/10) Installing ncurses-libs (6.0_p20171125-r1)
(8/10) Installing readline (7.0.003-r0)
(9/10) Installing sqlite-libs (3.25.3-r0)
(10/10) Installing python2 (2.7.15-r2)
Executing busybox-1.27.2-r11.trigger
OK: 51 MiB in 23 packages
Removing intermediate container ff3768926645
 ---> eda469dab9c6
Step 4/4 : COPY webserver.py /webserver.py
 ---> ccf7ee7227c9
Successfully built ccf7ee7227c9
Successfully tagged iox_aarch64_webserver:latest
```

Come prova, eseguire il contenitore appena creato e verificare se lo script funziona:

```
jedepuyd@deb9:~/iox_aarch64_webserver$ docker run -ti iox_aarch64_webserver
/ # uname -a
Linux dae047f1a6b2 4.9.0-8-amd64 #1 SMP Debian 4.9.144-3.1 (2019-02-19) aarch64 Linux
/ # python webserver.py &
/ # Starting webserver...

/ # netstat -tlpn
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:9000            0.0.0.0:*               LISTEN      13/qemu-aarch64-
sta
/ # exit
```

Come si può vedere in questo output, l'architettura del contenitore è l'arch64 di destinazione. E dopo aver avviato lo script, si vede che è in ascolto di richieste sulla porta 9000.

7. Creare il pacchetto IOx

Il contenitore è pronto per essere imballato. Prima di chiedere a ioxclient di farlo, è necessario creare il descrittore del pacchetto: **pacchetto.yaml**.

In questo file viene descritto l'aspetto del pacchetto, il numero di risorse necessarie per l'esecuzione e gli elementi da avviare.

```
jedepuyd@deb9:~/iox_aarch64_webserver$ vi package.yaml
jedepuyd@deb9:~/iox_aarch64_webserver$ cat package.yaml
descriptor-schema-version: "2.7"
```

```
info:
  name: "iox_aarch64_webserver"
  description: "simple docker webserver for arm64v8"
  version: "1.0"
  author-link: "http://www.cisco.com"
  author-name: "Jens Depuydt"

app:
  cpuarch: "aarch64"
  type: "docker"
  resources:
    profile: c1.tiny
    network:
      -
        interface-name: eth0
        ports:
          tcp: ["9000"]

  startup:
    rootfs: rootfs.tar
    target: ["python","/webserver.py"]
```

Come si può vedere, l'architettura della CPU è impostata su arch64. Per accedere alla porta TCP 9000, usare **rootfs.tar** come root e all'avvio, eseguire **python/webserver.py**.

L'ultima operazione da eseguire prima di creare il pacchetto è estrarre il file **rootfs.tar** dal contenitore Docker:

```
jedepuyd@deb9:~/iox_aarch64_webserver$ docker save -o rootfs.tar iox_aarch64_webserver
```

A questo punto, è possibile utilizzare **ioxclient** per generare il pacchetto IOx per IR1101:

```
jedepuyd@deb9:~/iox_aarch64_webserver$ ioxclient package .
Currently active profile :  default
Command Name:  package
No rsa key and/or certificate files provided to sign the package
Checking if package descriptor file is present..
Validating descriptor file /home/jedepuyd/iox_aarch64_webserver/package.yaml with package schema
definitions
Parsing descriptor file..
Found schema version  2.7
Loading schema file for version  2.7
Validating package descriptor file..
File /home/jedepuyd/iox_aarch64_webserver/package.yaml is valid under schema version 2.7
Created Staging directory at :  /tmp/017226485
Copying contents to staging directory
Creating an inner envelope for application artifacts
Generated  /tmp/017226485/artifacts.tar.gz
Calculating SHA1 checksum for package contents..
Updated package metadata file :  /tmp/017226485/.package.metadata
Root Directory :  /tmp/017226485
Output file:  /tmp/475248592
Path:  .package.metadata
SHA1 : 95abe28fc05395fc5f71f7c28f59eceb1495bf9b
Path:  artifacts.tar.gz
SHA1 : bdf5596a0747eae51bb0a1d2870fd09a5a16a098
Path:  package.yaml
SHA1 : e65a6fcbe96725dd5a09b60036448106acc0c138
Generated package manifest at  package.mf
Generating IOx Package..
Package generated at /home/jedepuyd/iox_aarch64_webserver/package.tar
```

Al momento, è disponibile un pacchetto da installare su IR1101 pronto come package.tar. Nella parte successiva viene illustrato come preparare il dispositivo per la distribuzione.

Parte 2. Configurare IR1101 per IOx

1. Abilitare Webinterface, IOx e Local Manager

Local Manager è una GUI che consente di distribuire, attivare, avviare, gestire e risolvere i problemi delle applicazioni IOx. Per IR1101, è integrato nella normale interfaccia Web di gestione. Quindi, prima è necessario attivarlo.

Eseguire questi passaggi su IR1101 per abilitare IOx e l'interfaccia Web.

```
BRU_IR1101_20#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
BRU_IR1101_20(config)#iox
BRU_IR1101_20(config)#ip http server
BRU_IR1101_20(config)#ip http secure-server
BRU_IR1101_20(config)#ip http authentication local
BRU_IR1101_20(config)#username admin privilege 15 password 0 cisco
```

Nell'ultima riga viene aggiunto un utente con privilegi 15. Questo utente avrà accesso all'interfaccia Web e al gestore locale IOx.

2. Configurare le reti IOx

Prima di accedere all'interfaccia Web, aggiungere la configurazione richiesta per la rete IOx. Per informazioni di base, vedere la documentazione di IR1101 per IOx: https://www.cisco.com/c/en/us/td/docs/routers/access/1101/software/configuration/guide/b_IR1101config/b_IR1101config_chapter_010001.html

In breve, le applicazioni IOx possono comunicare con il mondo esterno utilizzando l'interfaccia VirtualPortGroup0 (paragonabile a Gi2 su IR809 e Gi5 su interfacce IR829).

```
BRU_IR1101_20(config)#interface VirtualPortGroup0
BRU_IR1101_20(config-if)# ip address 192.168.1.1 255.255.255.0
BRU_IR1101_20(config-if)# ip nat inside
BRU_IR1101_20(config-if)# ip virtual-reassembly
BRU_IR1101_20(config-if)#exit
```

Quando si configura l'interfaccia VirtualPortGroup0 come NAT (Network Address Translation) all'interno, è necessario aggiungere l'istruzione ip nat external sull'interfaccia Gi 0/0/0 per consentire la comunicazione da e verso le applicazioni IOx con l'utilizzo di NAT:

```
BRU_IR1101_20(config)#interface gigabitEthernet 0/0/0
BRU_IR1101_20(config-if)#ip nat outside
BRU_IR1101_20(config-if)#ip virtual-reassembly
```

Per consentire l'accesso alla porta 9000 per il contenitore, che è possibile fornire 192.168.1.15, è necessario aggiungere una porta in avanti:

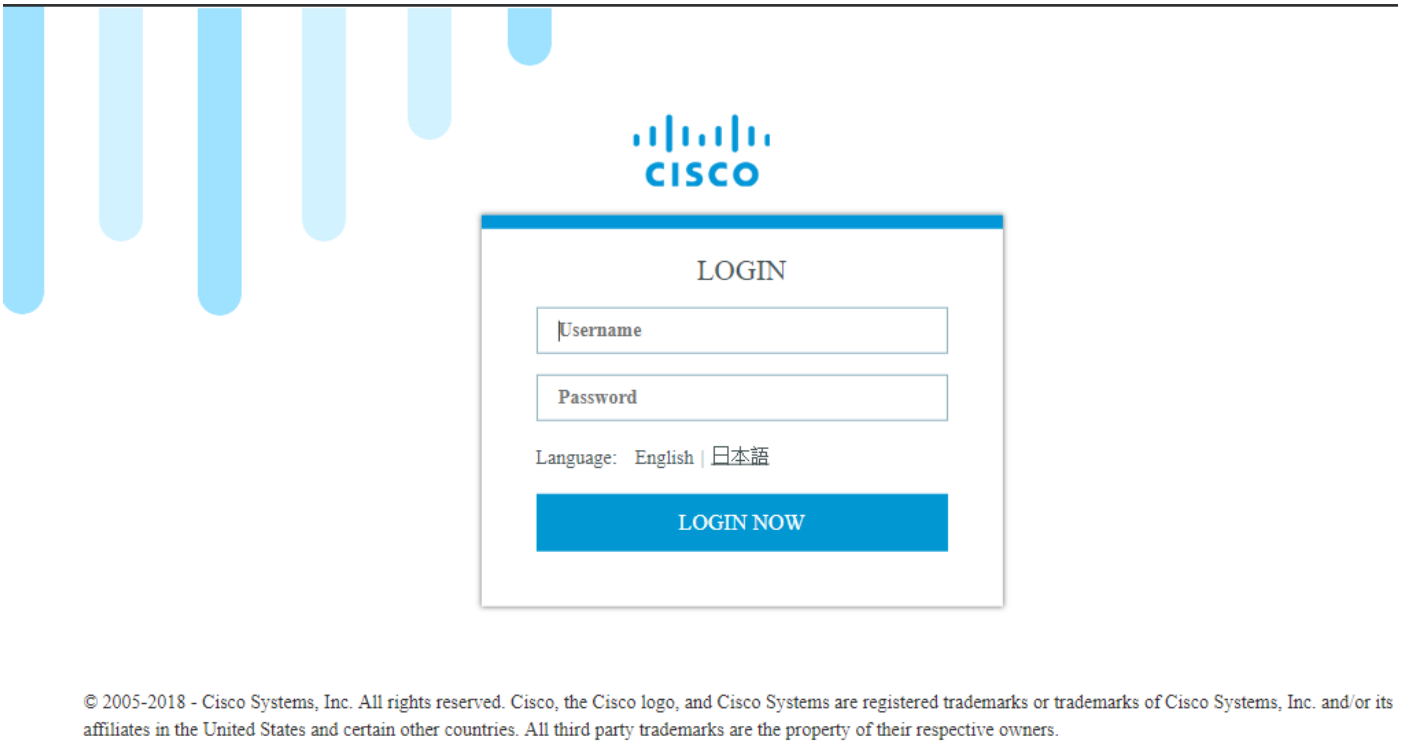
```
BRU_IR1101_20(config)#$ip nat inside source static tcp 192.168.1.15 9000 interface
```

```
GigabitEthernet0/0/0 9000
```

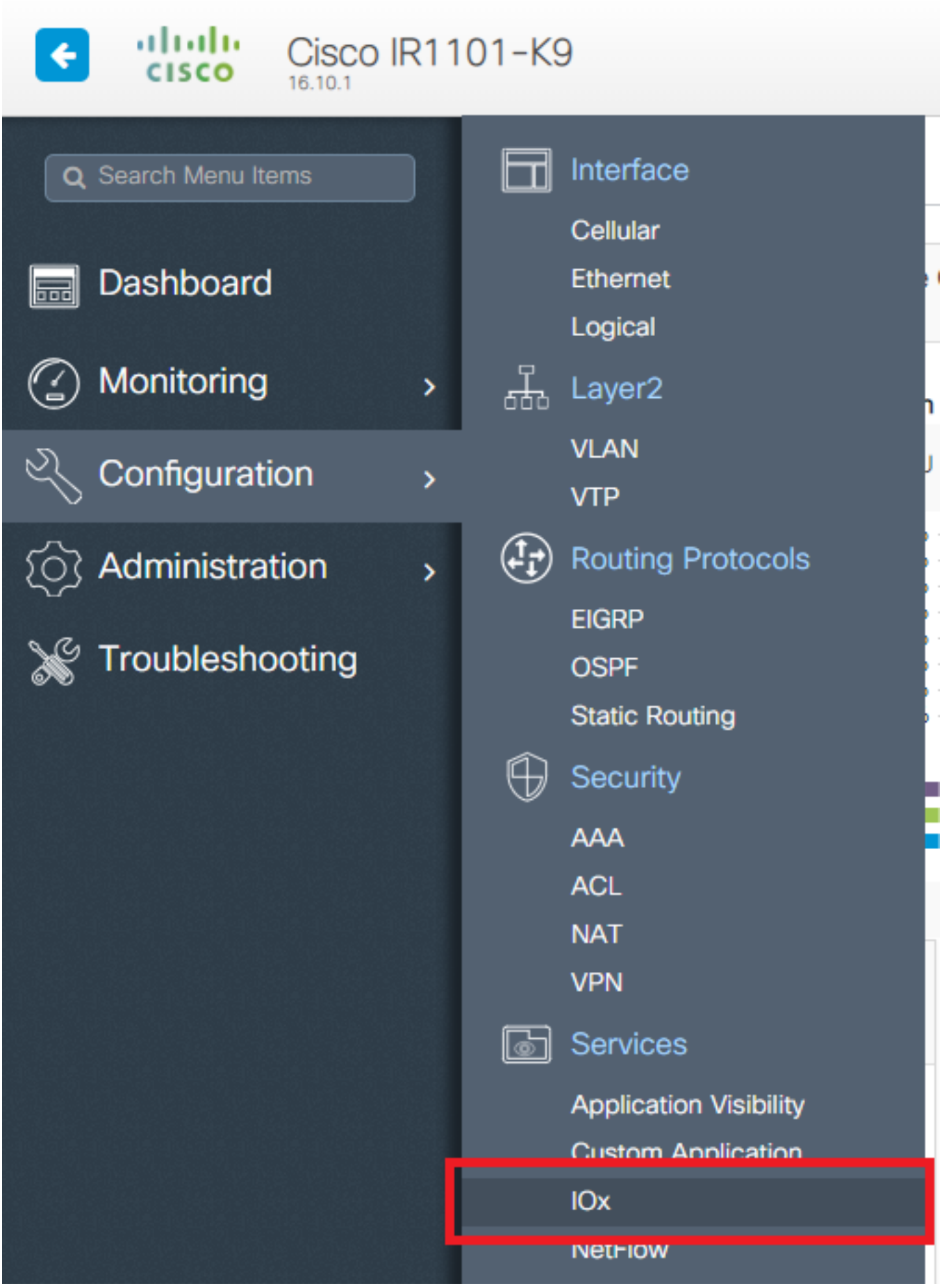
Per questa guida, utilizzare IP configurati staticamente per ciascuna applicazione IOx. Se si desidera assegnare dinamicamente indirizzi IP alle applicazioni, è necessario aggiungere la configurazione di un server DHCP nella subnet di VirtualPortGroup0.

Parte 3. Accedere a Local Manager e distribuire l'applicazione IOx

Dopo aver aggiunto queste linee alla configurazione, è possibile accedere a IR1101 utilizzando l'interfaccia Web. Passare all'indirizzo IP Gi 0/0/0 utilizzando il browser, come mostrato nell'immagine.

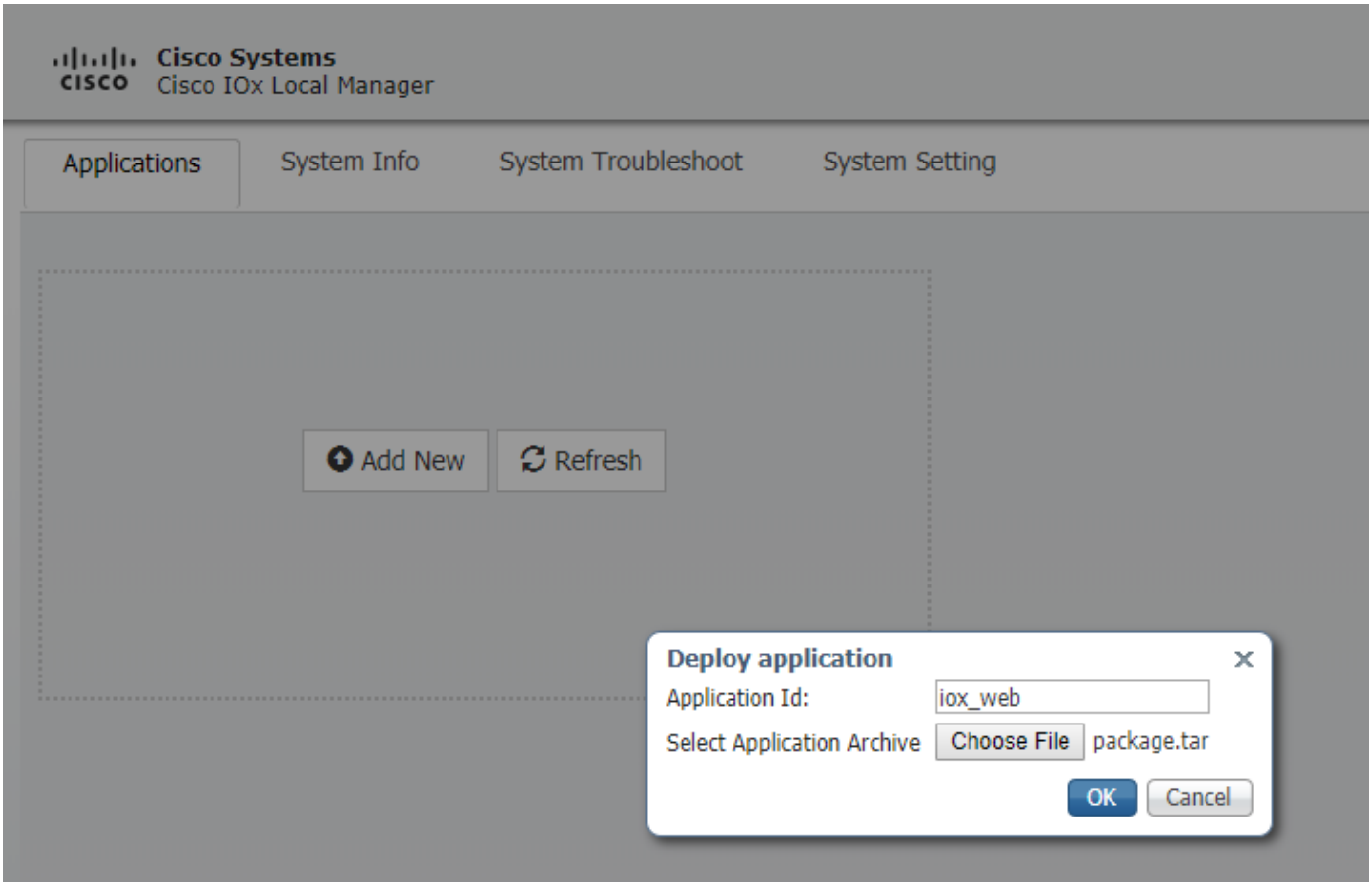


Utilizzare l'account con privilegi 15 creato nel passaggio 1. per accedere all'interfaccia Web e passare a **Configuration** - IOx come mostrato nell'immagine.

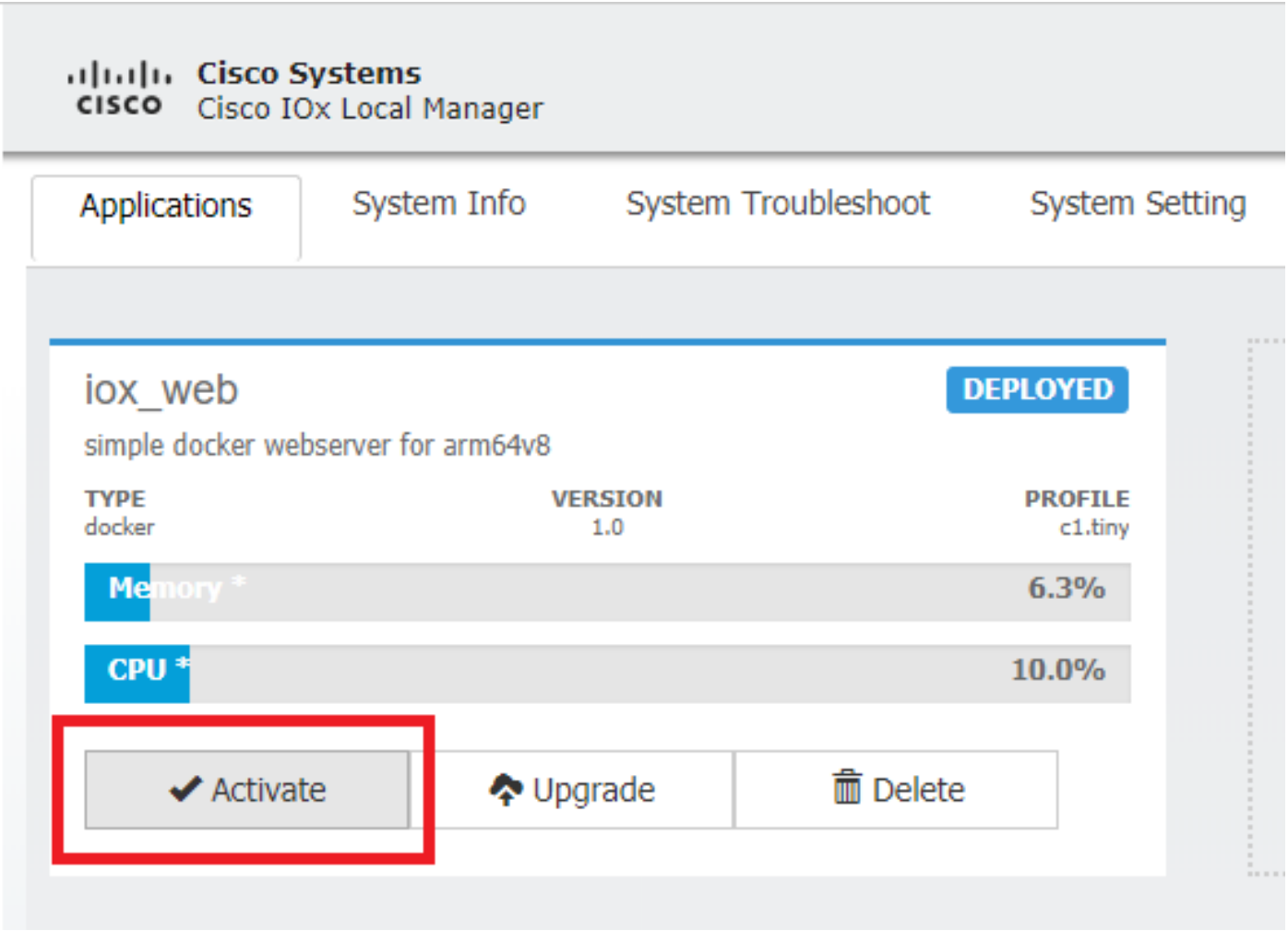


Nell'accesso a IOx Local Manager, utilizzare lo stesso account per continuare come mostrato nell'immagine.

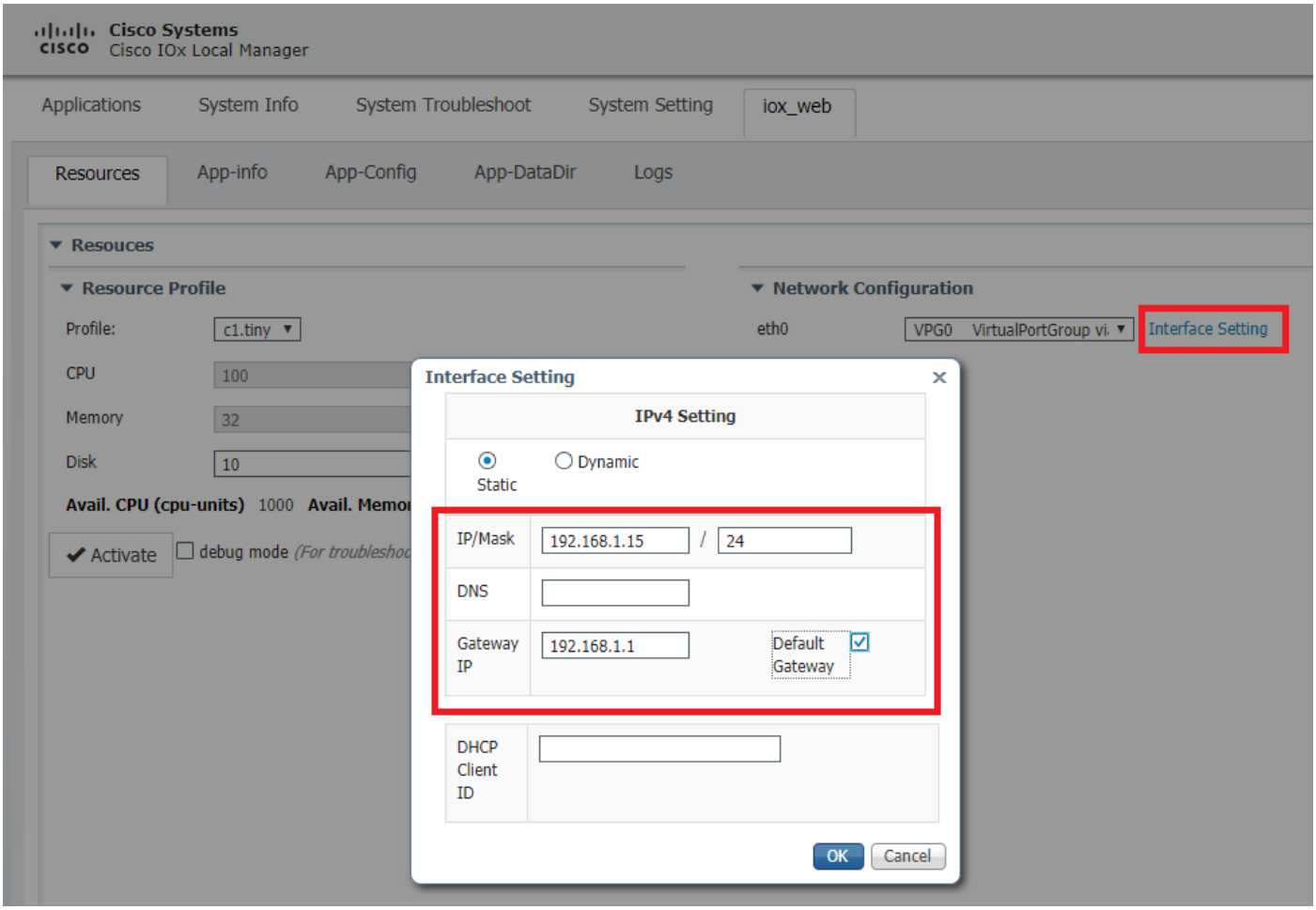
Fare clic su **Add New** (Aggiungi nuovo), selezionare un nome per l'applicazione IOx e scegliere il file package.tar creato nella parte 1, come mostrato nell'immagine.



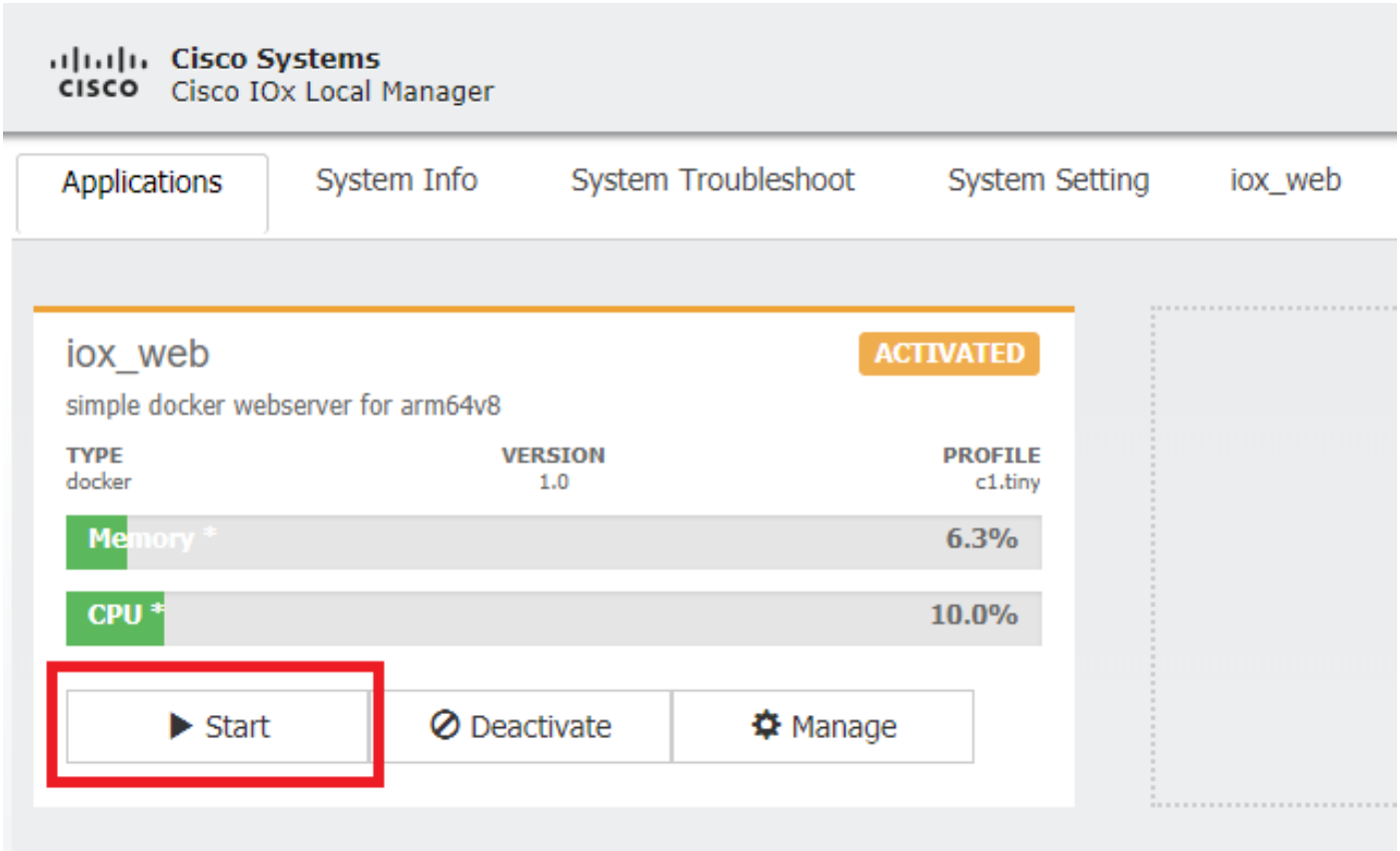
Una volta caricato il pacchetto, è possibile attivarlo come mostrato nell'immagine.



Nella scheda **Risorse**, aprire l'impostazione dell'interfaccia per specificare l'indirizzo IP fisso che si desidera assegnare all'app, come mostrato nell'immagine.



Fare clic su **OK**, quindi su **Attiva**. Una volta completata l'azione, tornare alla pagina principale di Gestione locale (pulsante **Applicazioni** nel menu superiore), quindi avviare l'applicazione come mostrato nell'immagine.



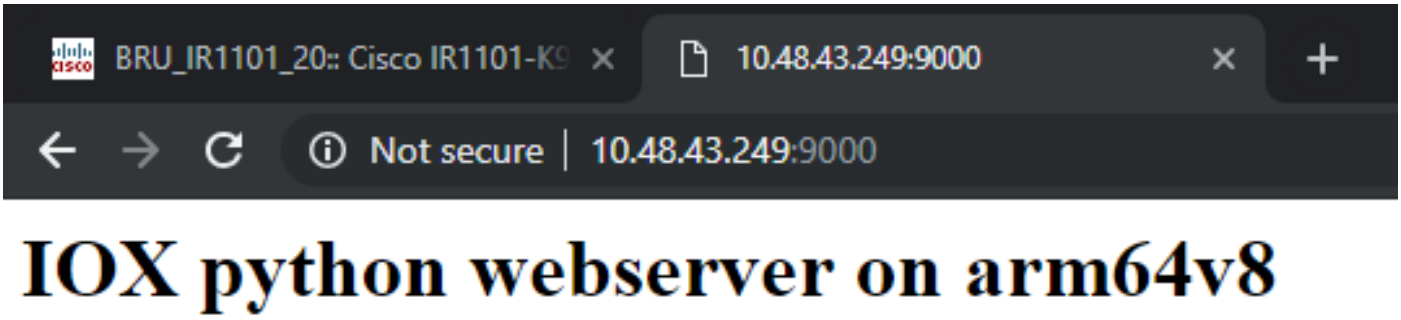
Dopo aver eseguito questi passaggi, l'applicazione dovrebbe essere in esecuzione e disponibile attraverso la porta 9000 con l'utilizzo dell'interfaccia Gi 0/0/0 di IR1101.

Verifica

Fare riferimento a questa sezione per verificare che la configurazione funzioni correttamente.

Per procedere alla verifica, è possibile accedere all'indirizzo IP dell'interfaccia Gi 0/0/0 su IR1101 utilizzando la porta 9000.

Se tutto va bene, dovrebbe vedere questo come segue, come è stato creato nello script Python.



Risoluzione dei problemi

Le informazioni contenute in questa sezione permettono di risolvere i problemi relativi alla configurazione.

Per risolvere il problema, è possibile controllare il file di log creato nello script Python con l'ausilio di un gestore locale.

Passare ad **Applicazioni**, fare clic su **Gestisci** nell'applicazione **iox_web**, quindi selezionare la scheda **Log** come mostrato nell'immagine.