ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI SCIENZE
Corso di Laurea in Ingegneria e Scienze Informatiche

PIATTAFORME PER INTERNET OF THINGS: WINDOWS IOT CORE COME CASO DI STUDIO

Elaborata nel corso di: Programmazione di Sistemi Embedded

Tesi di Laurea di:
MARCO NOBILE

Relatore:
Prof. ALESSANDRO RICCI

Correlatore:
Prof. DARIO MAIO

ANNO ACCADEMICO 2014–2015
SESSIONE II

PAROLE CHIAVE

Internet of Things

Protocolli e piattaforme IoT

Embedded Systems

Windows IoT Core

Smart City

Indice

Elenco delle figure **ix**

1 Internet of Things **3**

1.1 Panoramica . . . 3

1.1.1 Architettura dell'Internet of Things . . . 4

1.1.2 Applicazioni di IoT . . . 5

1.2 Protocolli per Internet of Things . . . 8

1.2.1 UPnP . . . 8

1.2.2 MQTT . . . 9

1.2.3 CoAP . . . 10

1.2.4 XMPP . . . 11

1.2.5 DDS . . . 12

1.2.6 AMQP . . . 15

2 Piattaforme open source per Internet of Things **17**

2.1 Eclipse IoT . . . 17

2.1.1 Standards . . . 17

2.1.2 Frameworks per IoT Gateways . . . 18

2.1.3 Servizi . . . 19

2.2 Kaa . . . 19

2.2.1 Architettura Generale . . . 20

2.2.2 Architettura ad Alto Livello . . . 21

2.3 SiteWhere . . . 23

2.4 Windows IoT Core . . . 25

3 Windows IoT Core **27**

3.1 Introduzione . . . 27

3.2 Quadro Architetturale di Windows 10 27
3.2.1 Kernel Mode 28
3.2.2 User Mode 31
3.3 Universal Windows Platform 33
3.3.1 Famiglie di Devices 33
3.3.2 Interfaccia Utente Universale 33
3.3.3 Linguaggi di programmazione e sviluppo 37
3.4 Windows IoT Core 39
3.4.1 Modalità Headed e Headless 40
3.4.2 Architetture e Progettazione di sistemi IoT con Windows 41
3.4.3 Windows Remote Arduino 41
3.4.4 Firmata 42
3.4.5 Architettura di WRA 43

4 Progetto: Smart Parking 45
4.1 Introduzione 45
4.1.1 Prove preliminari della tecnologia 45
4.1.2 Materiale utilizzato 46
4.2 Requisiti 46
4.3 Analisi 46
4.3.1 Analisi Funzionale 47
4.3.2 Analisi orientata agli Oggetti 48
4.4 Progettazione 50
4.4.1 Progettazione delle classi 50
4.4.2 Progettazione delle associazioni 52
4.4.3 Progettazione degli Stati: SingleParking 54
4.4.4 Rilevazione tramite sensore: messaggi scambiati . . . 55
4.4.5 GET al server per la visualizzazione dello stato: messaggi scambiati 57
4.5 Implementazione 58
4.5.1 Lato Arduino 58
4.5.2 Lato Raspberry Pi 2 59
4.5.3 Reattività agli imprevisti: robustezza 62
4.6 Collaudo 63
4.6.1 Modulo software su Arduino 63
4.6.2 Background Task su Raspberry 64

4.7 Estensioni 65
4.7.1 Sostituzione, integrazione e aggiunta di differenti tipi di dispositivo 65
4.7.2 Cambio di modalità di visualizzazione dei dati 66

5 Conclusioni **69**
5.1 Vantaggi dell'utilizzo di Windows IoT Core 69
5.2 Svantaggi nell'utilizzo di Windows IoT Core 71
5.3 Considerazioni Finali 72

Bibliografia **73**

Elenco delle figure

1.1 Schema di MQTT 10
1.2 Entità di DCPS 13
1.3 AMQP scope 16

2.1 Kaa structure 20
2.2 Kaa High-Level Architecture 22

3.1 Windows NT Architecture 29
3.2 Windows Device Families 33
3.3 Example Use of Visual State Triggers 36
3.4 Default Windows IoT Core app 40

4.1 Use Case Diagram 47
4.2 High Level Structural Diagram 48
4.3 Class Diagram 51
4.4 Association Diagram 53
4.5 Single Parking: StateChart Diagram 54
4.6 Sensor notify: Sequence Diagram 56
4.7 Request to Server: Sequence Diagram 57
4.8 Background Application Template 59
4.9 Parking Status View 66

Introduzione

Questa tesi si pone l'obiettivo di esplorare alcuni aspetti di uno dei settori più in crescita in questi anni (e nei prossimi) in ambito informatico: **Internet of Things**, con un occhio rivolto in particolar modo a quelle che sono le piattaforme di sviluppo disponibili in questo ambito. Con queste premesse, si coglie l'occasione per addentrarsi nella scoperta della piattaforma realizzata e rilasciata da pochi mesi da uno dei colossi del mercato IT: Microsoft. Nel primo capitolo verrà trattato Internet of Things in ambito generale, attraverso una panoramica iniziale seguita da un'analisi approfondita dei principali protocolli sviluppati per questa tecnologia. Nel secondo capitolo verranno elencate una serie di piattaforme open source disponibili ad oggi per lo sviluppo di sistemi IoT. Dal terzo capitolo verrà incentrata l'attenzione sulle tecnologie Microsoft, in particolare prima si tratterà Windows 10 in generale, comprendendo *UWP Applications*. Di seguito, nel medesimo capitolo, sarà focalizzata l'attenzione su Windows IoT Core, esplorandolo dettagliatamente (Windows Remote Arduino, Modalità Headed/Headless, etc.). Il capitolo a seguire concernerà la parte progettuale della tesi, comprendendo lo sviluppo del progetto **Smart Parking** in tutte le sue fasi (dei Requisiti fino ad Implementazione e Testing). Nel quinto (ed ultimo) capitolo, saranno esposte le conclusioni relative a Windows IoT Core e i suoi vantaggi/svantaggi.

Capitolo 1

Internet of Things

In questo capitolo si fornisce una panoramica riguardo Internet of Things e i suoi protocolli più diffusi.

1.1 Panoramica

Internet of Things (IoT) è il termine utilizzato per definire la rete di oggetti fisici, che incorporano unità di calcolo, sensori, etc., provvisti di connettività che permette loro di scambiare dati con altri dispositivi connessi alla rete. L'Internet delle cose permette agli oggetti di essere controllati da remoto, incentiva un'integrazione più diretta tra il mondo fisico e i sistemi di calcolatori.

Questo paradigma è stato introdotto da Kevin Ashton nel 1998, ma solamente negli ultimi anni, con l'assoluta affermazione di Internet, in tutti i suoi aspetti (dal World Wide Web alle sempre più diffuse tecnologie in ambito *mobile*), ha cominciato a vederne i primi sviluppi pratici.

Certamente, il punto di forza principale della visione di IoT [1] è il forte impatto sulla vita di tutti i giorni e sulle abitudini dei potenziali consumatori. Dal punto di vista di un utente, uno dei primi effetti dell'IoT sarebbe visibile in ambito lavorativo e domestico.

Domotica, Smart Office, risparmio energetico, questi sono solo alcuni degli scenari che in un futuro potrebbero cambiare la quotidianità di intere popolazioni.

Insieme a questo, però, sorgono anche numerosi potenziali problematiche che potrebbero dover essere risolte dagli sviluppatori di applicazioni

in questo ambito. Per esempio il raggiungimento di piena interoperabilità tra gli oggetti interconnessi, come fornire a questi ultimi il giusto grado di "smartness", perché siano sufficentemente adattativi e autonomi. Da non sottovalutare la necessità di garantire privacy e sicurezza (da sempre d'obbligo quando si parla di dispositivi che si interfacciano alla rete). In aggiunta, Internet of Things pone numerosi questioni da risolvere correlate al consumo energetico e all'efficente utilizzo delle risorse.

IoT è caratterizzato dalla presenza pervasiva di piccoli device spesso dotati di limitata capacità di calcolo e di memoria. Per queste ragioni, diviene naturale pensare di integrare ad IoT una tecnologia che, per sua natura, possiede virtualmente potenza di calcolo e capacità di *storage* illimitate: il **Cloud**.

Ad oggi numerosi gruppi industriali, di standardizzazione e di ricerca lavorano allo sviluppo di soluzioni che si adattino ai requisiti tecnologici imposti da IoT.

1.1.1 Architettura dell'Internet of Things

L'implementazione di IoT è fino ad ora basata su architetture non standardizzate. In linea di massima ogni architettura studiata ad oggi per l'implementazione di questa tecnologia propone una serie di livelli (o strati). In queste visioni architettoniche, allo strato più basso troviamo sensori, attuatori, e tutti quei dispositivi che permettono agli oggetti (o all'ambiente in generale) di diventare "smart", producendo dei dati che poi saranno consegnati ed elaborati nei livelli superiori. In cima allo schema a livelli troviamo invece le applicazioni, dedite ad interfacciarsi con l'utente finale. Al fine di fornire uno standard globale per l'Internet of Things, vi è al momento un progetto attivo , l'IEEE P2413 [3]. Questo progetto, oltre a porsi l'obiettivo di proporre un'architettura cross-domain per incentivare la crescita del mercato dell'IoT, prevede di occuparsi dei seguenti aspetti:

- Descrizione precisa dei vari domini di applicazione dell'IoT e delle loro astrazioni, rilevazione di differenze/similarità tra domini differenti.

- Produzione di un "Reference Model" che definisca le relazioni tra le varie nicchie di mercato (trasporti, sanità, etc.) e gli elementi architetturali in comune.

- Produzione di un modello per l'astrazione e la sicurezza dei dati(protection, security, privacy, safety).
- Creazione di un'Architettura di riferimento da porre sopra in Reference Model, al fine di coprire la definizione dei blocchi architetturali di base, e la loro abilità di essere inseriti in sistemi multi-strato.

1.1.2 Applicazioni di IoT

Le potenzialità offerte da IoT rendono possibile lo sviluppo di numerose applicazioni basate interamente su di esso. Attualmente solo alcune di queste sono state effettivamente realizzate. In questa sezione, vengono discusse alcuni importanti esempi di applicazioni [2] di Internet of Things.

Industria Aeronautica

IoT può essere d'aiuto per migliorare la sicurezza dei prodotti e dei servizi tramite la scoperta affidabile delle contraffazioni dei componenti meccanici di un veivolo. Per esempio, l'industria Aeronautica è vulnerabile al problema delle *parti non approvate sospette* (SUP: suspected unapproved parts). Una SUP è sostanzialmente una parte di un veivolo per la quale non è garantito il raggiungimento dei requisiti minimi. È possibile risolvere questo problema introducendo una serie di *Pedigrees elettronici* per certe categorie di pezzi di veivolo, che documentino la loro origine e tutti gli eventi critici durante il loro ciclo di vita (come ad esempio una eventuale modifica). Tramite la memorizzazione di questi pedigrees in un database decentralizzato, è possibile effettuare autenticazioni delle suddette parti prima che vengano montate all'interno del veivolo.

Settore Automobilistico

Le applicazioni in questo ambito di IoT includono l'utilizzo di *smart things* per monitorare e riportare una serie di parametri, quali: *pressione delle gomme*, **vicinanza con altri veicoli**, etc. Il sistema dei trasporti, attraverso l'integrazione dell'infrastruttura IoT, vedrà la nascita di comunicazioni V2V (*vehicle-to-vehicle*) e V2I (*vehicle-to-infrastructure*), le quali saranno la chiave per l'avanzamento di Sistemi di trasporto intelligenti, come servizi di sicurezza dei veicoli e gestione del traffico.

Sanità

IoT ha un vasto set di applicazioni possibili in questo settore. Con la possibilità di utilizzare telefoni cellulari con sensori RFID come piattaforma di controllo della consegna di medicinali e di parametri sanitari. Device wireless impiantabili e indirizzabili possono essere posti, per esempio, negli indumenti (*wearable*), al fine di monitorare lo stato di salute del paziente (o di un consumatore generico), così da potergli salvare la vita in situazioni di emergenza (chiamando soccorsi, prevedendo attacchi di cuore, etc.). Chips biodegradabili potranno essere introdotti nel corpo umano per azioni guidate, comandati sempre attraverso la rete, in termini di Internet of Things.

Domotica

I Sistemi domotici sono tipicamente utilizzati per il controllo di luci, riscaldamento, aria condizionanta, sistemi di comunicazione, gestione dei media, sicurezza e altri aspetti propri di un'abitazione, al fine di migliorare il comfort e l'efficenza energetica. E' questo il settore più in crescita ad oggi, che sta accompagnando l'Internet of Things nella sua diffusione.

Industria Farmaceutica

Per i prodotti farmaceutici, la sicurezza è uno degli aspetti più importanti. Nel paradigma IoT comporterebbe un potenziale beneficio l'introduzione di *smart labels* associate ai medicinali, tracciandoli attraverso la catena di distribuzione e monitorandone lo status attraverso sensori specifici. Per esempio: prodotti che necessitano di specifiche condizioni di conservazione possono così venir controllati e, in caso, scartati se le suddette condizioni dovessero venire violate durante il trasporto. Le smart labels potrebbero inoltre produrre beneficio direttamente ai pazienti fornendo un foglietto illustrativo elettronico e informando il consumatore riguardo il dosaggio e la data di scadenza. In collaborazione con un armadietto intelligente, capace di leggere le informazioni trasmesse dalle labels, ai pazienti potrà anche essere notificato di prendere le loro medicine a determinati intervalli, permettendo anche il controllo della regolarità degli stessi.

Industria Manufatturiera

Collegando i prodotti con dispositivi embedded smart, o tramite l'uso di identificatori univoci e *data-carriers* abilitati all'interazione con infrastrutture di rete di supporto e sistemi informatici, il processo di produzione può essere ottimizzato e l'intero ciclo di vita del singolo oggetto (dalla produzione allo smaltimento) monitorato. Attraverso il *tagging* di oggetti e contenitori, un buon livello di trasparenza diviene raggiungibile riguardo la locazione dei lotti, lo stato delle macchine di produzione, etc. Soluzioni di Auto-organizzazione e intelligenza manufatturiera artificiale possono essere progettate attorno ad oggetti identificabili nella rete.

Controllo Ambientale

Le applicazioni IoT per il controllo ambientale utilizzano tipicamente sensori per il monitoraggio della qualità di acqua e aria, condizioni atmosferiche, e, addirittura, movimenti della fauna in uno specifico habitat. E' possibile sviluppare sistemi di controllo terremoti (o tsunami) al fine di fornire servizi di emergenza per aiuti ed allerta tempestiva.

Agricoltura e Allevamento

La tracciabilità degli animali da agricoltura e dei loro movimenti richiede l'utilizzo di tecnologie come Internet of Things, così da rendere possibile la localizzazione *real time* di questi animali (ad esempio in caso di epidemie o malattie contagiose). In aggiunta, con l'applicazione di sistemi di identificazione , gli eventuali malesseri degli animali potranno essere sorvegliati e prevenuti.

Media e intrattenimento

Lo sviluppo delle tecnologie IoT permetterà di filtrare le news in base alla posizione dell'utente. Tag NFC (*Near Field Communication*) potranno essere inseriti nella cartellonistica per connettere il lettore ad un URI specifico, contenente informazioni più dettagliate correlate al cartello/poster.

1.2 Protocolli per Internet of Things

In uno scenario di Internet of things, è plausibile pensare che tutti i nostri oggetti *smart* debbano comunicare gli uni con gli altri (device to device, D2D) e/o che raccolgano una serie di dati per poi inviarli ad un server centrale (device to server, D2S). Infine, per permettere la diffusione di questi dati, un server deve poter condividere con altri server i dati ricevuti (server to server, S2S). I protocolli attualmente in voga nell'IoT che agevolano queste forme di interazione sono i seguenti:

- **UPnP**: protocollo che fornisce supporto ai dispositivi in una rete IP ad hoc.
- **MQTT**: protocollo per raccogliere dati da device e trasmettere le informazioni (D2S)
- **CoAP**: protocollo specializzato nel trasferimento via Web dei dati in reti o nodi dalle capacità limitate
- **XMPP**: un caso speciale di D2S; uno tra i miglior protocolli per connettere i device alle persone, fin quando sono connessi a dei servers
- **DDS**: un veloce bus per integrare smart objects (D2D)
- **AMQP**: un sistema di queues per connettere dei servers tra di loro (S2S)

1.2.1 UPnP

Letteralmente *Universal Plug and Play* [1], è un'architettura che utilizza a sua volta multipli protocolli. Aiuta i dispositivi in una rete IP ad hoc a trovarsi, a rilevare i servizi forniti dagli altri, a compiere azioni nella rete e a riportare eventi. Una rete ad hoc è una speciale rete senza una topologia predefinita, gestita dai devices che si trovano l'un l'altro e si adattano all'ambiete circostante. UPnP è ampiamente utilizzato in domotica e ambienti d'ufficio. UPnP è basato su un'applicazione HTTP alla quale sia server che client partecipano. Questo HTTP è stato esteso per essere usato sia su UDP che su TCP, entrambi capaci di unicast (HTTPU) o multicast (HTTPMU). La scoperta di device nella rete viene effettuata utilizzando

SSDP(Simple Service Discovery Protocol), basato anchesso su HTTP(sopra UDP), mentre la sottoscrizione di eventi e di notifiche avviene tramite GENA (*General Event Notification Architecture*). Nel particolare, i dispositivi notificano la loro esistenza e i servizi forniti tramite multicast. Nell'inviare il multicast gli attori possono anche decidere di indirizzarli a certi tipi di dispositivi o servizi. Le azioni sui servizi avvengono tramite chiamate a web servers SOAP. UPnP definisce una gerarchia di oggetti per i dispositivi partecipanti. Ogni device consiste in un *root device*. Ogni root device può pubblicare da zero a n servizi e/o dispositivi embedded. A loro volta, ogni embedded device può pubblicare i suoi servizi e i suoi devices, iterativamente. Ogni servizio pubblica una serie di **azioni** o **variabili di stato**; le azioni prendono in input un set di argomenti composti da un nome, una direzione(Input o Output) e un riferimento a una variabile di stato. Da quest'ultimo si deduce il tipo di dato nell'argomento. Le variabili di stato definiscono lo stato corrente di un servizio, ed ognuna ha un nome, un tipo ed un valore. Inoltre, una variabile di stato può essere normale, ad eventi o ad evento multicast. Quando le variabili ad eventi cambiano il loro valore, esso viene propagato nella rete attraverso un messaggio di evento(variabili ad evento: solo a chi si è sottoscritto; variabili ad eventi multicast: attraverso un messaggio multicast HTTPMU NOTIFY a tutti). Ogni device UPnP-compatibile nella rete viene descritto da un DDD(Device Description Document) in formato XML, memorizzato nel dispositivo stesso. Quando questo decide di connettersi alla rete, include la locazione del documento nel caso in cui terze parti fossero interessate a scaricarlo. Nel DDD, oltre alle generalità del dispositivo, vi si trovano anche i riferimenti ai dispositivi embedded e ai servizi forniti. Così come i dispositivi, ogni servizio pubblicato in rete viene descritto da un SCPD(Service Control Protocol Description).

1.2.2 MQTT

Letteralmente *Message Queue Telemetry Transport* [4], il suo obiettivo principale è il controllo remoto di numerosi devices di piccole dimensioni che hanno bisogno di essere monitorati e controllati. Permette di raccogliere dati da questi ultimi e trasportarli nel sistema informatico di destinazione. L'architettura più naturale per questo tipo di protocollo è di tipo Hub-and-spoke: tutti i device si connettono ad un server centralizzato. Poiché non si vogliono avere perdite di dati, MQTT è costruito su TCP.

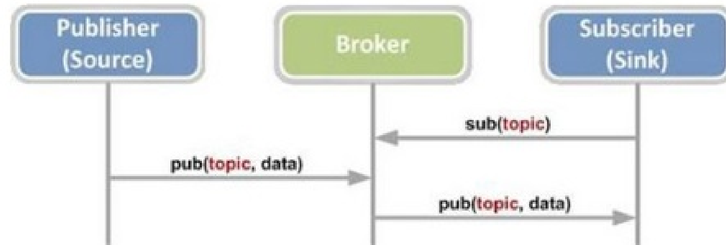


Figura 1.1: Schema di MQTT

Come si vede in figura 1.1, il protocollo MQTT è basato sul pattern **publish/subscribe** (opposto al request/respond di HTTP, ad esempio). Questo pattern presenta tre tipi di attori:

- **Publisher**: si connette al Message Broker e pubblica contenuti
- **Subscriber**: si connette ad un Message Broker e si sottoscrive ai contenuti a cui è interessati
- **Message Broker**: si assicura che i contenuti pubblicati vengano consegnati ai relativi Subscriber

Un Content è indentificato dall'argomento. Quando un publisher pubblica un contenuto, può scegliere se quest'ultimo debba essere memorizzato dal server oppure no. Se questo viene memorizzato, i Subscribers ricevono la versione più recente del contenuto alla prima sottoscrizione. Gli argomenti vengono ordinati in una struttura ad albero, come in un filesystem (padre/-figlio/...). Quando un subscriver si sottoscrive, può farlo relativamente ad un singolo argomento oppure ad un interno ramo di argomenti.

1.2.3 CoAP

Letteralmente *Constrained Application Protocol* [5], simile a HTTPU (HTTP Unicast), solo più semplice da codificare e decodificare, per i seguenti motivi:

- Rimpiazza gli header di testo di HTTPU con header binari più compatti.

- Riduce il numero di opzioni settabili nell'header.
- Riduce il set di metodi utilizzabili, abilitando solamente: GET, POST, PUT, DELETE.
- Riduce il numero di possibili codici di risposta.

Il protocollo CoAP prevede chiamate ai metodi tramite servizi di messaggi *confirmable* o *nonconfirmable*. Quando un ricevitore riceve un messaggio confirmable, spedisce sempre al mittente un acknowledgement, in questo modo il sender può reinviare i messaggi per i quali non ha ricevuto nessun acknowledgement (una volta scaduto un timeout). CoAP si taglia fuori dallo schema *Internet Media Type*, utilizzato in HTTP, e lo sostituisce con un set ridotto di *Content-Formats*, dove ogni formato è identificato da un numero intero(invece che dal suo corrispondente Internet Media Type). Oltre a ridimensionarsi per coprire l'esigenza di un contesto limitato, CoAP introduce anche alcune nuove features:

- Supporta multicasting per ricercare device in rete e comunicare attraverso i firewall.
- Comprende un'estensione che permette di bloccare gli algoritmi di trasporto per trasferire una mole di dati più consistente del normale.
- Un'altra estensione permette di avere unarchitettura di notifica e sottoscrizione a eventi.
- Supporta la crittografia nel caso unicast tramite DTLS (Datagram Transport Layer Security).

1.2.4 XMPP

Originariamente chiamato *Jabber*, XMPP(Extensible Messaging and Presence Protocol) [6] era stato pensato per l'Instant Messaging. Questo protocollo utilizza XML per realizzare comunicazioni person-to-person in linguaggio naturale. E costruito sopra TCP o alcune volte addirittura sopra HTTP. Il suo punto di forza è l'indirizzamento *nome@dominio.com*, che aiuta a connettere i nodi. Nell'IoT, XMPP offre un modo semplice di indirizzare un device. Non è progettato per essere veloce, poiché molte implementazioni utilizzano polling o controllo di update on demand. É una

buona soluzione per applicazioni IoT costumer-oriented. Il *core* di questo protocollo presenta le seguenti tecnologie:

- Uno strato base di streaming XML.
- Crittografia del canale tramite TLS (*Transport Layer Security*).
- String Authentication tramite SASL (*Simple Authentication and Security Layer*).
- Utilizzo di UTF-8 per un supporto unicode completo.
- Informazione sulla presenza (*network availability*) incorporata.
- Abilitata la presenza di liste di contatti.

1.2.5 DDS

In contrasto con MQTT e XMPP, il *Data Distribution Service* (DDS) [7] mira ai dispositivi che utilizzano direttamente i dati provenienti da altri device. DDS può consegnare efficentemente milioni di messaggi al secondo simultaneamente a più dispositivi. É stato concepito come una soluzione infrastrutturale alla programmazione di applicazioni incentrate sui dati. Il suo scopo è di nascondere interamente le problematiche della gestione della comunicazione nella programmazione di applicazioni data-centric. I dispositivi hanno bisogno di comunicare con tanti altri in modi diversi, quindi TCP è troppo restrittivo in questo caso. Questo protocollo offre un dettagliato controllo della *Quality of Service*, multicast, affidabilità configurabile, e ridondanza pervasiva. DDS offre la possibilità di filtrare e selezionare quali dati debbano andare dove (con multiple destinazioni ove richiesto). Per questo preciso uso, l'architettura utilizzata è un *bus diretto D2D* con un modello di dati relazionale. Così come un database controlla gli accessi ai dati, il bus controlla l'accesso e le modifiche ai dati da parte di molteplici utenti simultanei. Sistemi ad alte prestazioni di device integrati utilizzano DDS. Lo standard è rappresentato da una serie di API suddivise in due livelli:

- **DCPS** (*Data Centric Publish/Subscribe*) è il livello inferiore di DDS che definisce le entità, i ruoli, le interfacce e le policy di QoS per

la piattaforma publish/subscribe, nonché le tecniche di discovery dei partecipanti alla comunicazione. DCPS rappresenta, in sostanza, la parte dello standard relativa alla comunicazione di rete.

- **DLRL** (*Data Local Reconstruction Layer*) è il livello superiore di DDS che definisce il modello di interazione tra il mondo ad oggetti dell'applicazione ed i dati provenienti da DCPS. Tramite DLRL è possibile mappare i dati scambiati all'interno di un topic con un oggetto del livello applicativo, in modo tale da propagare, in modo automatico e trasparente, gli aggiornamenti dell'oggetto dall'applicazione verso la rete e viceversa. DLRL è definito opzionale all'interno di DDS.

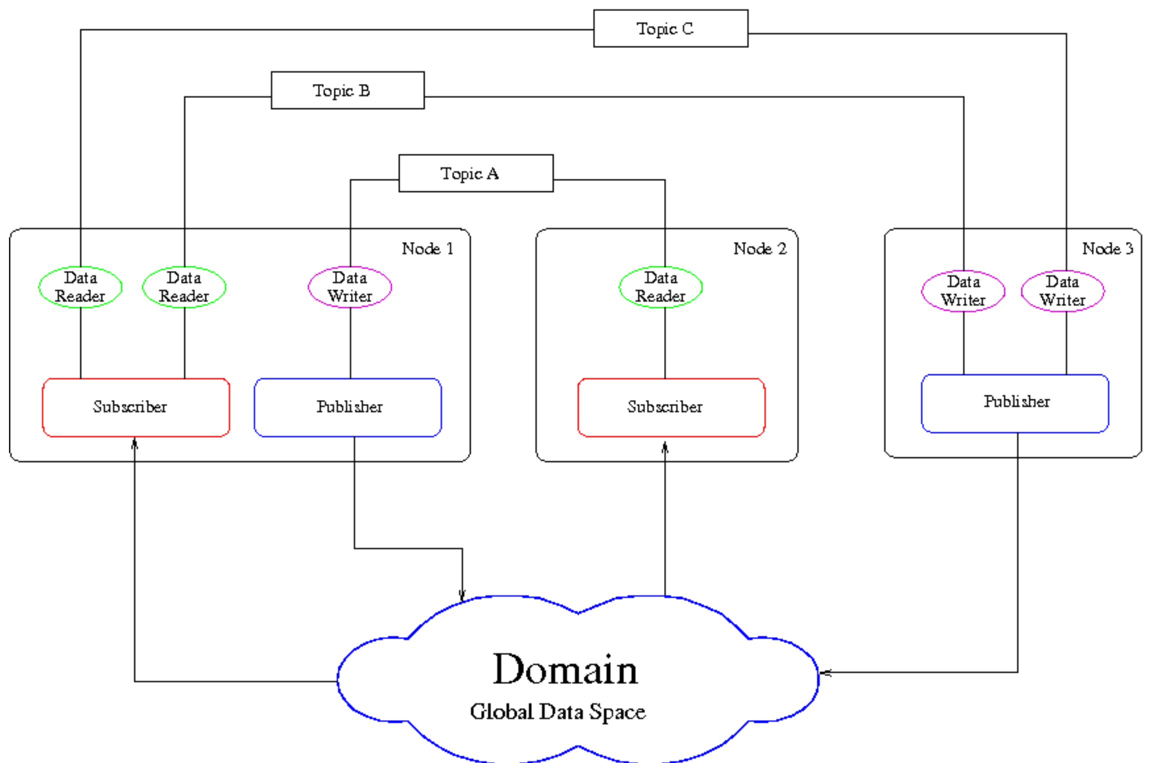


Figura 1.2: Entità di DCPS

Per meglio comprendere il funzionamento del livello inferiore (DCPS), vengono di seguito elencate le entità che ne fanno parte (raffigurate in figura 1.2):

- **DomainParticipantFactory**: un Singleton che funge da Factory per l'accesso allo spazio di comunicazione in DDS, chiamato *Global Data Space* (GDS). Il GDS è diviso in domini, ognuno dei quali è identificato da un nome. All'interno di un dominio si possono avere diversi topic. Due topic con lo stesso nome possono coesistere in DDS solo all'interno di diversi domini.

- **DomainParticipant**: punto di accesso per la comunicazione in uno specifico dominio. Un DomainParticipant può essere creato solo attraverso il DomainParticipantFactory. Il DomainParticipant a sua volta funge da Factory per la creazione di Topic, Publisher e Subsbriber.

- **Topic**: partizione tipizzata del dominio di comunicazione, all'interno del quale i Publisher ed i Subscriber pubblicano e ricevono i dati. Ogni Topic è caratterizzato da un nome e da un tipo. Il nome identifica il topic all'interno del dominio ed il tipo caratterizza i dati scambiati all'interno del topic stesso. Per distinguere i diversi oggetti di un topic si utilizza il concetto di *chiave* (preso in prestito dal modello Entità-Relazione tipico dei database relazionali). Una chiave è composta da un membro o da una combinazione di membri del tipo.

- **Publisher**: entità responsabile della diffusione dei dati provenienti dai diversi DataWriter ad esso associati. Un Publisher funge da Factory per la creazione dei DataWriter.

- **DataWriter**: punto di accesso per la pubblicazione di dati all'interno di un topic. Il DataWriter viene creato a partire da un Publisher, il quale a sua volta è associato univocamente ad un Topic. Per questo motivo il DataWriter è un'entità astratta che viene resa concreta tipizzando la sua interfaccia con il tipo di dato corrispondente al topic a cui il DataWriter si riferisce.

- **Subscriber**: entità responsabile della ricezione dei dati pubblicati sul topic a cui esso è associato. Funge da Factory per la creazione dei DataReader e, a run-time, si occupa di smistare ai diversi DataReader ad esso associati i dati ricevuti.

- **DataReader**: punto di accesso per la ricezione di dati all'interno di un topic. Come il DataWriter, anche il DataReader è un'entità astratta che viene resa concreta tipizzando la sua interfaccia con il tipo di dato corrispondente al topic a cui il DataReader si riferisce.

1.2.6 AMQP

AMQP(*Advanced Message Queuing Protocol*) [8] è basato sulle code. Invia messaggi transazionali tra server. Questo portocollo è focalizzato nel non perdere messaggi, perciò le comunicazioni avvengono tramite TCP. In più, i nodi finali devono notificare ogni ricezione di messaggio. É per lo più utilizzato per scambio di messaggi nel business. AMQP permette alle applicazioni di specificare quali messaggi dovranno essere ricevuti e da chi, come dovranno essere effettuati gli scambi e altri parametri legati a sicurezza, performance e affidabilità. Nell'IoT, AMQP è appropriato per analisi lato server.

Architettura di AMQP

L'architettura è suddivisa in 3 parti principali che definiscono:

- Protocolli di rete.
- Rappresentazione dei dati dei messaggi impacchettati.
- Semantica dei servizi *Brokers*.

Da quanto osservabile in figura 1.3, gli strati sono così composti:

- **Transport**: trasporto affidabile basato su TCP/IP (suddiviso in *canali*)
- **Messaging Protocol & Type System**: trasporto di messaggi attraverso Peers di Trasporto (tramite anche trasferimento di responsabilità sui dati).
- **Broker**: rappresenta l'intermediaro nell'architettura. Si occupa di memorizzare per un lasso di tempo i messaggi e presenta features di distribuzione (quali *Shared Queues, Topis*, etc.). Il tutto viene gestito tramite la presenza di indirizzi locali nel broker.

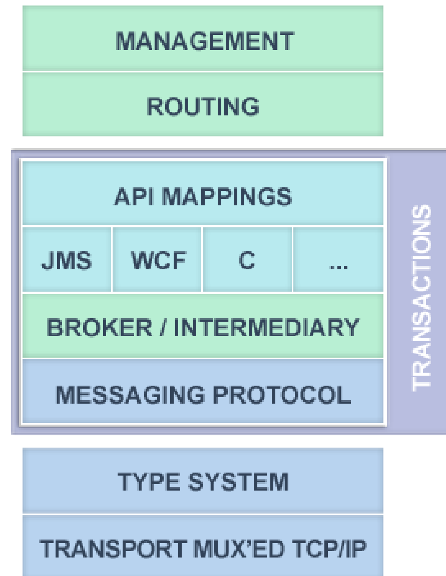


Figura 1.3: AMQP scope

- **API mappings**: strato che permette ad AMQP di essere cross-language e possiede una rappresentazione canonica dei concetti presenti tra le diverse API
- **Routing (tra Brokers)**: strato di invio di messaggi tra i brokers. Vengono anche inoltrati gli indirizzi delle parti comunicanti e propagate le sottoscrizioni a determinati messaggi/topics.
- **Managenent**: gestione standardizzata di un qualsiasi broker AMQP. Vengono inviati messaggi di gestione ai broker per effettuare i cambiamenti. É in questo strato che si fa leva sulle capacità di sicurezza e routing dei livelli inferiori.

Capitolo 2

Piattaforme open source per Internet of Things

Attualmente non vi sono molte piattaforme costruite appositamente per IoT open source. In questo capitolo ne verranno esplicate alcune delle più famose, più o meno complesse, e per domini più o meno specifici.

2.1 Eclipse IoT

Eclipse ha rilasciato una piattaforma di sviluppo Open Source per lo sviluppo di applicazioni IoT [9] per agevolare gli sviluppatori in molti degli aspetti critici della creazione di sistemi per IoT (modificare le impostazioni delle applicazioni a run-time, deploy sicuro di nuove versioni dellapplicazione, controllo che il gateway abbia un accesso appropriato all rete). Eclipse ha costruito questa tecnologia col preciso scopo di fornite implementazioni open source di standard, servizi e frameworks per uno sviluppo per Internet of Things aperto a tutti.

2.1.1 Standards

CoAP

(Constrained Application Protocol) è un protocollo RESTful indirizzato a devices embedded e reti wireless. CoAOP estende in alcuni ambiti HTTP, fornendo *notifiche push* native e gruppi di comunicazione.

ETSI SmartM2M

Fornisce specifiche per servizi e applicazioni M2M, focalizzandosi particolarmente su aspetti di IoT e Smart Cities

MQTT

Per l'implementazione di questo standard è stato costituito il progetto **Paho**.

OMA LightweightM2M

LWM2M è uno standard industriale per la gestione di devices M2M e IoT. Si basa su CoAP ed è ottimizzato per comunicazioni su reti cellulari o di sensori. E' costituito da un modello estensibile che permette lo scambio di dati tra applicazioni, oltre alla gestione del core dei devices (upgrade di firmware, monitoraggio della connesione, . . .).

2.1.2 Frameworks per IoT Gateways

IoT Gateways aiuta a gestire l'interazione tra sensori e attuatori, servizi cloud, etc. Eclipse IoT presenta due Frameworks per costruire gateways:

Kura

Basata su framework Java/OSGi(*Open Service Gateway iniziative*, implementa un modello a componenti completo e dinamico). Le sue API offrono accesso all'hardware sottostante (porte seriali, GPS, etc.), gestione delle configurazioni di connessione, comunicazione con piattaforme integrate M2M/IoT, oltre che alla gestione dei gateways.

Mihini

Scritto in linguaggio di scripting Lua, fornisce una gestione di connettività a basso livello per assicurare che una connessione di rete affidabile sia disponibile. Si comporta come uno strato di astrazione per l'hardware sottostante e permette lo scambio intelligente di trasmissioni dati tra server e devices.

2.1.3 Servizi

Realizzati da Eclipse per la costruzione di applicazioni IoT per specifici domini industriali.

SmartHome

Framework per la realizzazione di soluzioni in ambito domotico con una focalizzazione particolare sugli ambienti eterogenei (come ad esempio soluzioni che si trovano a dover gestire con l'integrazione di diversi protocolli e standards). L'obiettivo di SmartHome è quello di fornire un accesso uniforme ai dispositivi e alle informazioni, facilitandone le possibili interazioni.

Eclipse SCADA

Modo per connettere dispositivi industriali differenti ad un sistema di comunicazione comune, cosi da poter processare a posteriori e far visualizzare i dati agli operatori. Fornisce un sistema SCADA(*Supervisory Control And Data Acquisition*) che include un servizio di comunicazione, un sistema di monitoraggio, archiviazione dei dati e visualizzazione.

2.2 Kaa

Kaa [10] è un middleware per Internet of Things open source. Offre un certo numero di servizi, quali:

- Gestione e raggruppamento di devices
- Personalizzazione delle configurazioni e dei comportamenti dei dispositivi
- Comunicazione cross-device
- Monitoraggio real-time, collezionamento e analisi di dati telematici
- Invio di notifiche e coinvolgimento degli utenti

Questa piattaforma permette la gestione dei dati prodotti dagli *smart objects* connessi, mette a disposizione un server(con relativa infrastruttura

back-end) per alleggerire il carico di lavoro degli sviluppatori ed infine fornisce degli appositi SDK per gli end-point della rete. Questi SDK hanno il compito di abilitare lo scambio bidirezionale real-time di dati con il server.

2.2.1 Architettura Generale

La struttura del framework KAA consiste di un **server Kaa** e **endpoint SDKs**. Nello specifico, il server Kaa espone interfacce per lintegrazione e offre capacità amministrative. Un **endpoint SDK** è una libreria che si occupa della comunicazione, data marshalling (il processo che trasforma i dati secondo uno standard prima che venga trasmesso in rete), persistenza dei dati, etc. Questo SDK può essere utilizzato per creare *Client KAA*, cioè parti di software che utilizzano le funzionalità di Kaa e sono installati su uno smart device connesso in rete. E proprio il Client Kaa che processa i dati e li spedisce al server Kaa. Una rappresentazione visuale dell'architettura Kaa è visibile in figura 2.1

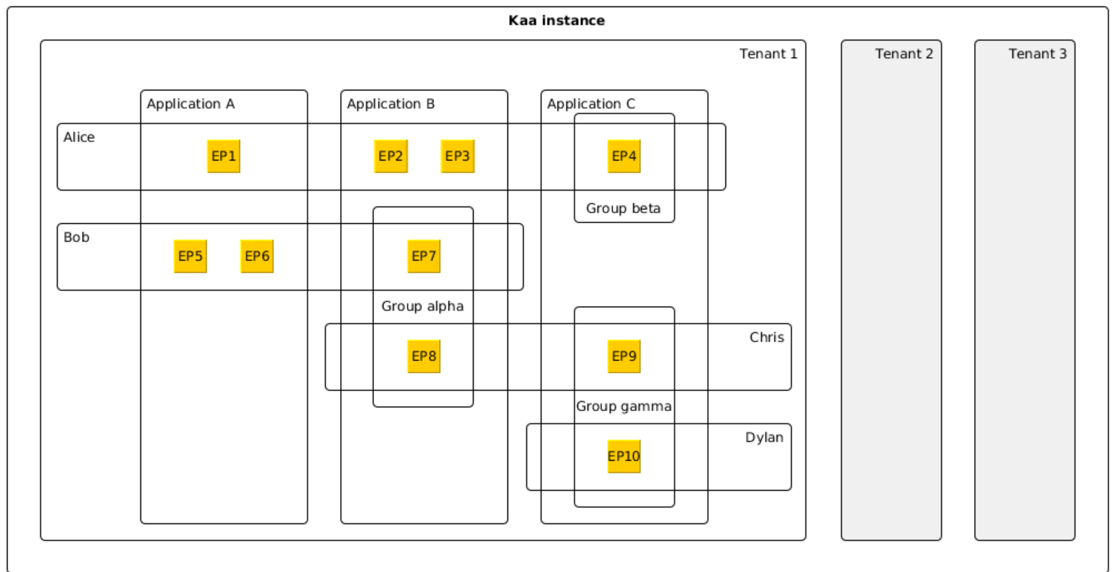


Figura 2.1: Kaa structure

Istanza Kaa

Le istanze di Kaa non sono altro che uno o più Servers Kaa (Kaa cluster) interconnessi con degli endpoints (Kaa Clients).

Applicazione Kaa

Una *application* in Kaa rappresenta una famiglia di implementazioni di uno specifico software utilizzato da degli endpoints (per esempio due versioni di misuratori di frequenza del suono che differiscono nella loro implementazione per, rispettivamente STM32 e Arduino, sarebbero considerate la stessa applicaziona Kaa).

Tenant

Una tenant in Kaa è un'entità commerciale separata che contiene i suoi endpoints, applicazioni e utenti.

2.2.2 Architettura ad Alto Livello

Un server Kaa è la composizione di *Control*, *Operations* e *Bootstrap servers* (figura 2.2). Esso utilizza anche databases per la persistenza dei dati e Apache ZooKeeper per la coordinazione dei server. In più, Kaa fornisce una web UI che si integra con il Control server e permette all'utente di creare applicazioni, registrare e configurare endpoints, creare gruppi di endpoints, etc.

Control server

É responsabile della gestione di tutto il sistema di dati, processa le chiamate API dall'interfaccia Web o dai sistemi integrati esterni e inoltra le notifiche agli Operation servers. Gestisce inoltre i dati memorizzati nel database (indipendentemente per ogni tenant) e notifica gli updates tramite il protocollo Thrift-based. Un Control server mantiene sempre una lista aggiornata degli Operation servers disponibili, ottenendola con ZooKeeper

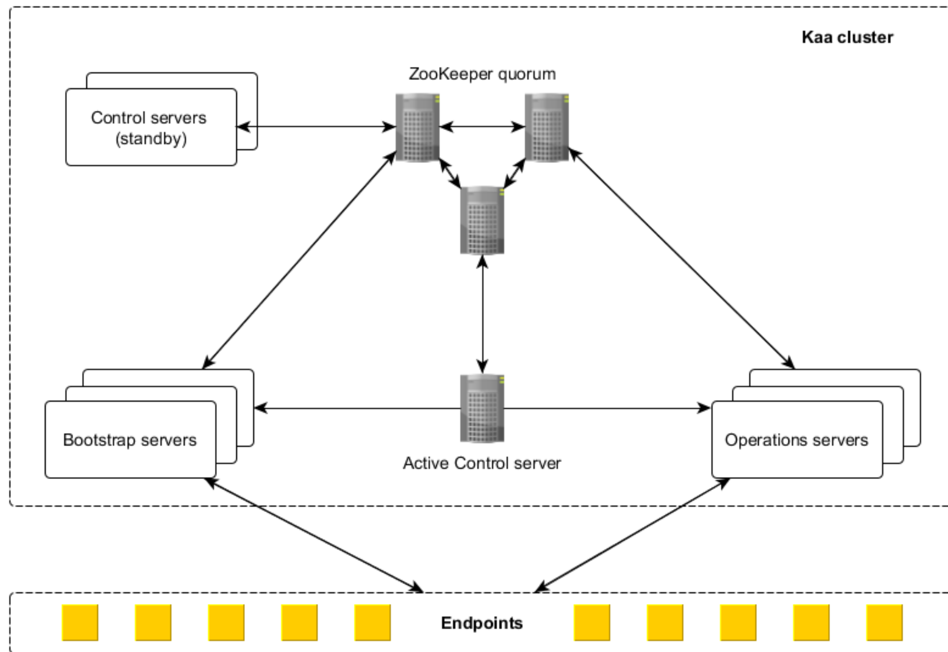


Figura 2.2: Kaa High-Level Architecture

Operations server

É il server *lavoratore* responsabile della gestione di richieste multiple da più Clients contemporaneamente. I compiti più comuni di queste entità sono:

- Registrazione di endpoints
- Aggiornamento dei profili degli endpoints
- Configurazione della distribuzione degli updates
- Inoltro di notifiche

Multipli Operations server possono essere uniti in un cluster per la gestione dello scaling. É presente, in caso di cluster, una procedura di *salvataggio* per gli endpoints nel caso in cui uno degli operation server subisca un malfunzionamento, redirottandoli verso un server disponibile più vicino.

Bootstrap server

É il responsabile del dirottamento degli endpoints verso gli operation servers. Ogni endpoint possiede al suo interno una lista di Bootstrap servers da poter contattare, e, facendolo , ricevono in risposta dal Bootstrap un set di Operation servers disponibili, insieme alle credenziali di sicurezza.

2.3 SiteWhere

Questa *Open Platform* [11] fornisce agli sviluppatori una serie di funzionalità, raggruppate in 5 macrogruppi

Framework di Core e Piattaforme

- Fornisce un server che agisce da controller per la computazione dei dati provenienti dai dispositivi.
- Fornisce un *Core Object model* e delle API per la persistenza delle informazioni in un formato normalizzato, così che dati provenienti da differenti tipi di devices possano essere integrati agli altri.
- Utilizza un approccio *pluggable* che permette a terze parti di estendere e personalizzare il sistema, al fine di integrarlo con altre nuove tecnologie.
- Fornisce un'applicazione amministrativa HTML5 che permette a tutti i dati del sistema di poter essere visualizzati e manipolati.
- Fornisce servizi REST che permettono ad applicazioni esterne di interagire con il sistema.
- Offre Client Java che può interagire con i servizi REST forniti, per rendere, a sistemi esterni che usano Java, un facile interfacciamento a SiteWhere

Persistenza di Big Data

- Offre due implementazioni per la persistenza dei dati, **MongoDB** e **Apache HBase**. Entrambe progettate per immagazzinare ingenti quantità di dati con throughput elevato e bassa latenza.

- L'implementazione di MongoDB ottimizzata permette di effettuare *push* di 10000 eventi al secondo in un'istanza di Cloud di media potenza.

- L'implementazione di HBase, basata su uno schema progettato ad hoc, scala linearmente attraverso un cluster, il quale eventualmente può anche essere espanso.

- Entrambe le implementazioni citate sopra sono conformate alle API dei dispositivi SiteWhere, che forniscono una visualizzazione coerente dei dati indipentemente da dove questi sono allocati.

Framework di Comunicazione tra Devices

- Permette a tutti i tipi di device di interagire con SiteWhere. Gli sviluppatori possono creare plugins conformi con le interfacce del framework per aggiungere dei devices a SiteWhere

- Le interfacce per le sorgenti di eventi permettono a SiteWhere di ricevere input di ogni tipo. Molti protocolli sono inclusi pronti all'uso(quali **MQTT**, **Stomp**, **AMQP**, **JMS**, **WebSockets**, **connessioni dirette a sockets**, etc.)

- Le interfacce per la destinazione dei comandi forniscono un sistema pluggable per inviare comandi a devices attraverso vari protocolli e codifiche. Nativamente SiteWhere ne include già varie implementazioni.

- Le interfacce di gestione per altre funzionalità di sistema, come registrazione di dispositivi, sono anche queste pluggable.

Gestione delle risorse

- Associazione di device con risorse esterne, quali persone o oggetti fisici.

- Le informazioni di queste risorse sono offerte tramite un **Asset Management Framework** (anch'esso pluggable) che permette a sistemi esterni recuperarle.

- SiteWhere traccia gli assegnamenti tra risorse e devices nel tempo, fornendo una *assignment history.* In questo modo è possibile risalire a quale device fosse collegata una risorsa, per esempio al verificarsi di un dato evento.

Integrazione

SiteWhere fornisce nativamente un supporto per l'integrazione di altri frameworks e servizi.

2.4 Windows IoT Core

Con l'introduzione del Sistema Operativo Windows 10, Microsoft ha introdotto sul mercato anche un nuovo framework appositamente pensato per l'Internet of Things, Windows IoT Core. E' oggi possibile scaricare e installare un sistema dalle dimensioni ridotte, compatibile con alcune Architettura hardware (quali Raspberry Pi 2, Intel Galileo e MinnowBoard MAX). Questo Core entra a far parte in tutto e per tutto delle famiglie di device Windows 10, supportando tutta una serie di API universali della piattaforma, più un gruppo più ristretto di API esclusive. Per questo progetto, Microsoft ha stretto un partnership anche con il team di Arduino, realizzando un framework speciale anche per la serie Arduino UNO (*Windows Remote Arduino*). Questi argomenti verranno trattati nei seguenti capitoli.

Capitolo 3

Windows IoT Core

3.1 Introduzione

Con Windows 8 Microsoft introdusse Windows Runtime(WinRT) come architettura condivisa per rendere quasi universalmente compatibili le applicazioni dai dispositivi dotati di tale Sistema Operativo (desktop, mobile, tablet, etc.). Windows 10 introduce **Universal Windows Platform** (UWP), la quale evolve il modello WinRT e si inserisce nel core unificato Windows 10. Essendo parte di questo core, UWP ora implementa una piattaforma per applicazioni unificata su ogni device. Con questa evoluzione, le app UWP possono invocare non solo le API di WinRT, comuni a tutti i devices, ma anche API specifiche della famiglia di dispositivi sul quale l'app sta eseguendo (inclusi Win32, .NET). A questo proposito è stata unificata anche l'interfaccia grafica, tramite *layout* e *controlli adattativi*, per garantire copertura di un vasto range di risoluzioni. Il framework di sviluppo .NET (corredato del motore *Common Language Runtime*) è disponibile per Windows 10 ed è possibile utilizzarlo per sviluppare applicazioni UWP.

3.2 Quadro Architetturale di Windows 10

Windows 10 fa parte della famiglia **Windows NT** (Windows NT 10.0) [17], la quale comprende sistemi operativi **preemptive** (con la capacità di interrompere l'esecuzione di un task per cominciarne un altra, attraverso un *context switch*) e **reentrant** (l'esecuzione può essere interrotta in qualsiasi

momento e poi richiamata successivamente *in sicurezza*). I Sistemi Windows NT sono composti da un'architettura a **strati**, suddivisi tra due componenti principali: **Kernel mode** e **User Mode** (figura 3.1).

3.2.1 Kernel Mode

Possiede libero e totale accesso alle risorse hardware e di sistema, ed esegue codice in un'**area di memoria protetta**. Qui avviene il controllo di:

- **Accessi allo scheduling**
- **Assegnamento di priorità ai threads**
- **Gestione della memoria**
- **Interazioni con l'Hardware**

La priorità dei processi in questa modalità è di norma sempre superiore a quelli in User Mode. I processi in User Mode devono richiedere al kernel l'esecuzione di alcune operazioni per loro conto. L'apparato Kernel Mode è costituito da **Executive Services** (composti da numerosi moduli relegati a compiti specifici), **Kernel Drivers**, **Kernel** e **Hardware Abstraction Layer**.

Executive Services

Le informazioni di questo strato sono contenute nel file **NTOSKRNL.EXE**. I compiti con cui esso ha a che fare sono: **I/O**, **gestione di oggetti**, **sicurezza** e **processi**. Anche alcuni servizi di sistema (come *system calls*) vengono implementati a questo livello. Per svolgere al meglio i suoi compiti, Executive Services si compone dei seguenti componenti:

- **Object Manager**: utilizzato per ridurre le duplicazioni di risorse negli altri sottosistemi *executive*. Per l'Object Manager, ogni risorsa è vista come un oggetto, anche essa in realtà è fisica. La creazione di un oggetto è un processo a due fasi: **creazione** e **inserimento**. La creazione consiste nell'allocazione di un oggetto vuoto e nella prenotazione di tutte le risorse necessarie da parte del manager. L'inserimento dell'oggetto lo rende accessibile attraverso il suo nome, o

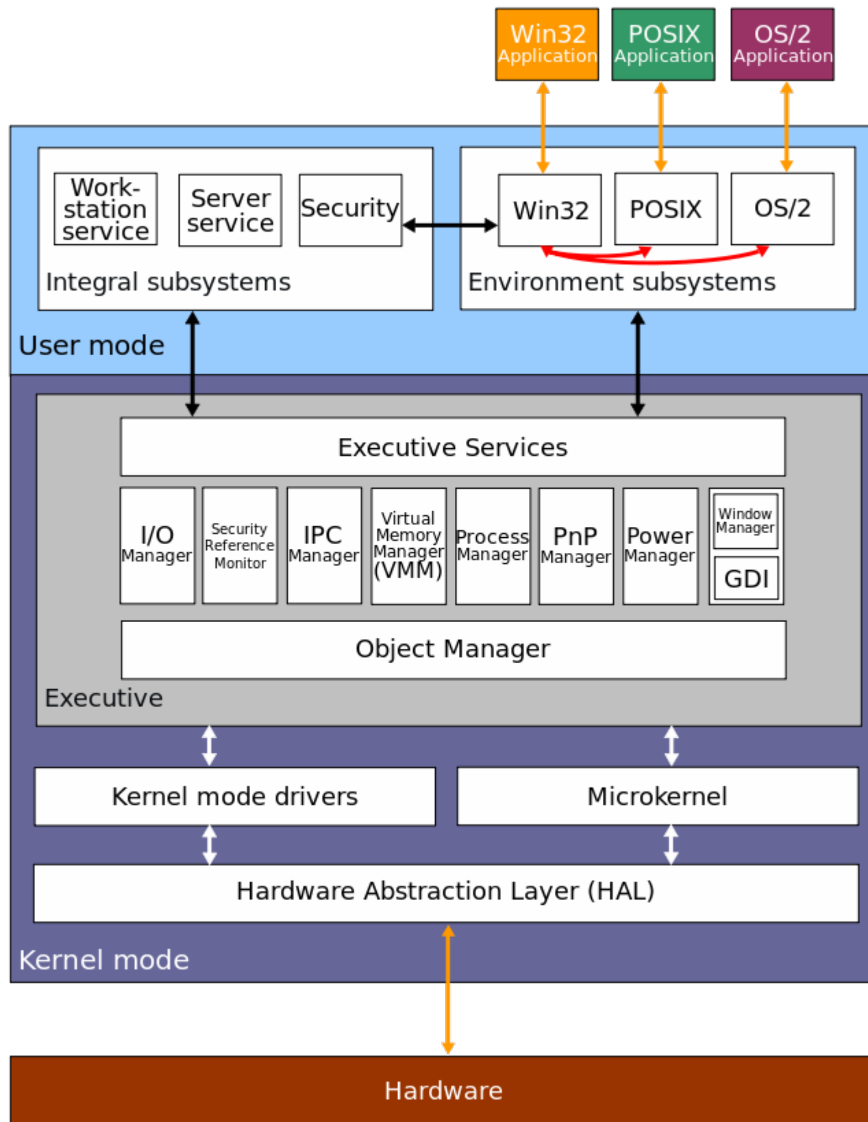


Figura 3.1: Windows NT Architecture

un *cookie* chiamato **handle**. L'interno ciclo di vita dell'oggetto viene gestito sempre da questo componente, fin quando non viene incaricato di eliminarlo.

- **Cache Controller**: Strettamente connesso con **Memory Manager**, **I/O Manager** e **I/O drivers** per fornire una cache comune per i file di I/O. Lavora su blocchi di file, per operazioni tra file locali e remoti, e assicura un certo grado di coerenza con i file mappati a memoria (ossia file adibiti a memoria virtuale), poiché i blocchi di cache sono un caso speciale di file mappati a memoria e le *cache misses* sono un caso speciale di *page fault*

- **Configuration Manager**: implementa **Windows Registry** (database immagazzina i valori di configurazione e le opzioni).

- **I/O Manager**: Permette ai devices di communicare con i sottosistemi in User Mode. Traduce i comandi di *read* e *write* in user mode in comandi analoghi che vengono poi passati ai *device drivers*. Può accettare richieste di I/O da file system e le traduce in chiamate specifiche al device.

- **Local Procedure Call (LPC)**: fornisce porte di comunicazione interprocesso con semantiche di connessione.

- **Memory Manager**: gestisce la memoria virtuale, controlla la paginazione , la protezione delle aree di memoria, e implementa un **allocatore** di memoria.

- **Process Structure**: Gestisce la creazione e la distruzione di Threads e Processi, e implementa il concetto di **Job**, ovvero un gruppo di processi che possono essere terminati come se venisse considerato uno.

- **PnP Manager**: Gestisce il **Plug and Play**, e supporta il riconoscimento di dispositivi e la loro installazione in fase di *boot*. Ha anche la responsabilità di avviare o stoppare dispositivi su richiesta.

- **Graphical Drive interface**: è responsabile di Tasks quali disegnare linee o curve, rendering di font, etc.

Kernel

Si trova nel livello tra i processi Executive e l'HAL. Fornisce sincronizzazione multi-processo, scheduling di thread e interruzioni, gestione e dispatching di eccezioni. E' responsabile anche di inizializzare i drivers in fase di boot iniziale e di tutte quelle operazioni tradizionali di un microkernel. Il kernel spesso si interfaccia con il Process Manager e il livello di astrazione è tale, che è solo quest'ultimo a chiamare il primo.

Kernel-mode Drivers

Windows NT utilizza questo strato per interagire con i dispositivi Hardware. Ogni device è visto dal codice in user-mode come un file oggetto dell'I/O Manager. I Kernel-mode Drivers si compongono di tre livelli: **Highest**, **Intermediate** e **Lower** drivers.

Hardware Abstraction Layer

É lo strato che si pone tra l'hardware vero e proprio e il resto del Sistema Operativo. E' stato progettato per *nascondere* le differenze tra i vari hardware, e fornisce una piattaforma solida sulla quale gira il kernel. Lo strato HAL include codice per hardware specifici che controlla le interfacce di I/O, gli interrupt e multipli processori. A discapito della sua posizione nella raffigurazione dell'architettura, questo strato non sta totalmente al di sotto dell'area Kernel-mode, dato che tutte le implementazioni ad oggi conosciute di HAL dipendono in qualche modo dallo strato Kernel, o addirittura dall'Executive. E' importante sottolineare che l'astrazione dell'hardware *non* include l'astrazione dell'**instruction set**.

3.2.2 User Mode

I sistemi e sotto-sistemi presenti in User Mode hanno **accesso limitato** alle risorse di sistema. Questi hanno la capacità di inoltrare richieste di I/O ai driver appropriati presenti in kernel mode attraverso un componente chiamato **I/O Manager**. In questo strato sono presenti due tipi di sottosistemi, esposti di seguito.

Environment Subsystems

Sottosistemi delegati al lancio di applicazioni scritte per diversi tipi di Sistema Operativo. Nessuno degli Environment Subsystems può accedere *direttamente* all'hardware, e deve richiedere l'accesso alla memoria al componente chiamato **Virtual Memory Manager** (presente in kernel mode). I due principali sottosistemi di questo gruppo sono:

- **Win32**: fornisce supporto alle applicazioni Windows a 32 bit e alle **Virtual DOS Machines** (consentono di lanciare applicazioni MS-DOS e Windows a 16 bit). Contiene anche la Console e gestisce *shutdown* e *hard-error* per tutti gli altri sottosistemi Environment. Il sottosistema Win32, denominato **csrss.exe**, include la funzionalità di Gestione Attività (*Window Manager*). Gestisce gli eventi di Input (provenienti per esempio da tastiera o mouse), per poi passarli alle applicazioni che necessitano di questi inputs.

- **Interix**: contiene numerose utilities Unix (compilatore GCC, GNU debugger, librerie pthreads, sockets, etc.)

Integral Subsystem

Ha il compito di controllare le funzioni specifiche di sistema per conto degli Environment Subsystems. E' costituito da:

- **Security Subsystem**: gestisce le autenticazioni, i permessi e altri aspetti di sicurezza al fine di garantire o negare l'accesso alle risorse. Gestisce richieste di login e determina quali risorse di sistema necessitano un controllo da parte di Windows NT.

- **Workstation Service**: è un'API che si occupa di fornire l'accesso alla rete.

- **Server Service**: è un'API che permette al computer di creare servizi di rete

3.3 Universal Windows Platform

3.3.1 Famiglie di Devices

Le applicazioni Windows 8.1 e Windows Phone 8.1 nascono per essere eseguite su uno specifico Sistema Operativo (o Windows, o Windows Phone). Con Windows 10, invece, le app possono operare su una o più famiglie di dispositivi. Ogni famiglia di dispositivi possiede un suo insieme di API *specifiche* e comportamenti.

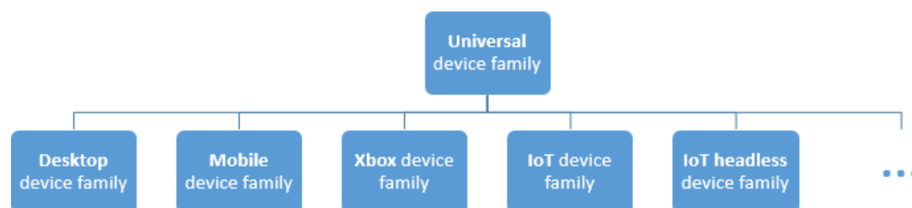


Figura 3.2: Windows Device Families

Le API di ogni famiglia sono catalogate da un nome e un numero di versione. La **Universal Device family** invece, non è direttamente la base di un Sistema Operativo. Al contrario, infatti, ogni famiglia di Sistema eredita (come fosse una sottoclasse) tutte le API di questa categoria. Questo garantisce ad ogni famiglia (e quindi ad ogni dispositivo) di possedere tale set base di API universale. Un beneficio dato dalle famiglie è che si può implementare un app veramente universale per tutta la piattaforma windows 10, per poi potersi specializzare (utilizzando *codice adattativo*) e utilizzare features presenti solo in alcuni tipi di device(ovvero utilizzare API proprie delle famiglie).

3.3.2 Interfaccia Utente Universale

Come specificato in precedenza, un'applicazione UWP [12] può essere eseguita su diversi tipi di device, i quali presentano differenti forme di input, risoluzioni, densità DPI, etc. Per questo Windows 10 mette a disposizione nuovi controlli universali, layout e tools per aiutare lo sviluppatore ad adattare la UI il più possibile al device sul quale l'applicazione viene lanciata.

Per esempio, si potrebbe voler implementare un UI che si adatti alla risoluzione dello schermo quando l'applicazione sta girando su un device mobile piuttosto che su un pc desktop. Per le UWP alcuni aspetti legati all'interfaccia grafica si adattano automaticamente in base al device (come controlli, bottoni e sliders). Potrebbe essere altresì necessario adattare il design della user-experience in base al dispositivo. Per queste necessità, Windows 10 viene incontro agli sviluppatori con alcune features, quali:

- **Controlli Universali** e **pannelli di layout** che aiutano ad ottimizzare l'interfaccia in base alla risoluzione dello schermo.
- **Gestori di input comuni** che permettono di intercettare gli input di tocco, penna, mouse, tastiera o controller (ad esempio un controller dellXbox).
- Ausilio di **Tools** che aiutano a progettare UI adattative in base a risoluzioni differenti.
- **Scaling adattativo** che aggiusta la risoluzione e la differenza di DPI tra i dispositivi.

Controlli universali e pannelli di layout

Windows 10 include nuovi controlli (come calendari e split view). I controlli sono stati aggiornati per lavorare al meglio sugli schermi più ampi, adattarsi in base al numero di pixel liberi sul dispositivo e gestire multipli tipi di input.

UI adattativa con pannelli adattativi

I pannelli di layout definiscono la *grandezza* e la *posizione* dei loro figli in base allo spazio libero. Per esempio uno StackPanel ordina i suoi figli in modo sequenziale (orizzontalmente o verticalmente), Grid si comporta come una grid CSS che li posiziona in celle, etc. Il nuovo RelativePanel implementa uno stile che viene definito in base alle relazioni tra i suoi figli. Esso è stato pensato per creare layouts adattabili in base al cambiamento di risoluzione, e facilita il procedimento di riarrangiamento degli elementi, permettendo la creazione di interfacce grafiche più dinamiche senza l'uso di layout innestati. In un Relative Layout sono stati introdotti i concetti di *rightof*, *leftof*, *below*, etc.

Utilizzo di Visual State Triggers

I **Visual State Triggers** si utilizzano nell'implementazione delle interfacce grafiche, per permettere a queste ultime di adattarsi in base al numero di pixels rimasti disponibili sullo schermo. Definiscono una *soglia* oltre la quale uno **stato di visualizzazione** viene attivato. Quest'ultimo, poi, setta le proprietà del layout appropriate alla dimensione della finestra che ha generato il cambio di stato. Per poter implementare questa feature basta inizializzare allinterno del layout un elemento figlio chiamato *VisualStateGroup*, per poi innestargli una serie di *VisualState*, ognuno col corrispettivo *StateTriggers*, contenenti ognuno il layout prestabilito all'interno del tag *VisualState.Setters*. Un esempio dell'utilizzo di questa feature viene rappresentato in figura 3.3.

```
<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup>
            <VisualState x:Name="wideView">
                <VisualState.StateTriggers>
                    <AdaptiveTrigger MinWindowWidth="720" />
                </VisualState.StateTriggers>
                <VisualState.Setters>
                    <Setter Target="best.(RelativePanel.RightOf)" Value="free"/>
                    <Setter Target="best.(RelativePanel.AlignTopWidth)" Value="free"/>
                </VisualState.Setters>
            </VisualState>
            <VisualState x:Name="narrowView">
                <VisualState.Setters>
                    <Setter Target="best.(RelativePanel.Below)" Value="paid"/>
                    <Setter Target="best.(RelativePanel.AlignLeftWithPanel)" Value="true"/>
                </VisualState.Setters>
                <VisualState.StateTriggers>
                    <AdaptiveTrigger MinWindowWidth="0" />
                </VisualState.StateTriggers>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
    ...
</Grid>
```

Figura 3.3: Example Use of Visual State Triggers

Tools per anteprime

Una volta disegnata l'interfaccia di una applicazione UWP, lo sviluppatore può controllare come questa viene visualizzata nelle diverse risoluzioni tramite la **device preview toolbar** di Visual Studio. É così possibile anche testare il corretto funzionamento di eventuali utilizzi di VisualState.

Scaling Adattativo

Con scaling grafico si intende la capacità di un UI di modificare le dimensioni dei propri elementi basandosi sulle caratteristiche dello schermo. Windows 10 introduce un'evoluzione del modello di scaling preesistente. In aggiunta

al contenuto vettoriale scalare, vi è un set unificato di fattori scalari che fornisce una dimensione consistente per elementi grafici attraverso le diverse dimensioni di schermo e risoluzioni. Lo Store sceglie e scarica l'asset che meglio vi si adatta basandosi sui DPI del dispositivo.

Gestione comune degli input

Per accedere agli input sono state prodotte una serie di API apposite:

- **CoreIndipendentInputSource** permette di elaborare un input grezzo nel main Thread o in un task separato

- **PointerPoint** unifica un tocco, un click e un dato della penna in un singolo set di eventi e interfacce che possono essere elaborati in un qualsiasi Thread utilizzando CoreInput

- **PointerDevice** è una API che supporta la richiesta delle capacità del device, così da poter determinare quale modalità di input sono possibili per un singolo device.

3.3.3 Linguaggi di programmazione e sviluppo

Per realizzare un'applicazione UWP, Windows permette di scegliere tra diversi linguaggi di programmazione, quali: **C#**, **C++**, **Visual Basic**, etc. Per il design di GUI è possibile usare lo standard **XAML**, già utilizzato per le applicazioni Windows/Windows Phone 8/8.1 (Con C++ è permessa anche la scelta di DirectX). Per quanto riguarda **JavaScript**, è possibile utilizzarlo associandogli, per la visualizzazione grafica, il linguaggio **HTML**. Ognuno dei linguaggi di programmazione/markup citato, già di per se oppure grazie alla piattaforma UWP, è da intendersi universale tra i dispositivi Windows 10 ed è quindi trattato allo stesso modo su ogni famiglia di device. Se si desidera produrre codice per una o solo alcune famiglie di devices, si può decidere di utilizzare codice e UI adattative (come specificato precedentemente). Di seguito vengono descritti alcuni casi particolari possibili quando ci si confronta con questo tipo di framework.

Chiamare un'API implementata dalla famiglia target

Ogni volta che si vuole invocare un'API, è necessario sapere se quest'ultima sia o meno implementata dalla famiglia target dell'applicazione che abbiamo creato. Se l'API che decidiamo di invocare è della stessa famiglia del nostro device, è possibile richiamarla tranquillamente senza eseguire alcun controllo, poiché vi è la certezza che, se imponiamo ad un sistema di poter essere installato solo su una certa famiglia di device e invochiamo API di quest'ultima, il codice andrà sempre bene ad ogni installazione.

Chiamare un API che NON è implementata dalla famiglia target

In questo caso si può scegliere di utilizzare *codice adattativo* per scrivere la nostra invocazione. Per scrivere codice adattativo vi sono *due passi* da seguire:

1. **Rendere utilizzabili le API** a cui si vuole accedere nel proprio progetto. Per farlo, bisogna aggiungere un riferimento all'extension SDK che rappresenta la famiglia di devices proprietaria di quella API.
2. Bisogna **utilizzare la classe Windows.Foundation.Metadata.ApiInformation** in una condizione (*if*) per scoprire la presenza o meno dell'API che si vuole chiamare nel dispositivo corrente. Questa condizione viene scoperta al momento della *run* dell'applicazione su un dispositivo. Tramite questa classe possiamo scoprire se:
   - Un tipo è presente nelle API del device

     ```
     ApiInformation.IsTypePresent(nometipo)
     ```

   - Un evento è presente nelle API del device

     ```
     ApiInformation.IsEventPresent(nomeTipo, nomeEvento)
     ```

   - Un metodo è presente nelle API del device

```
ApiInformation.IsMethodPresent(nomeTipo,
   nomeMetodo)
```

- Una proprietà è presente nelle API del device

```
ApiInformation.IsPropertyPresent(nomeTipo,
   nomeProprietà)
```

- Un intero contratto di API è presente nel device

```
ApiInformation.IsApiContractpresent(nomecontract,
   versionemaggiore, versioneminore)
```

API Win32 in UWP

Un'applicazione UWP o un componente di WinRT scritto in C++/CX ha accesso alle API Win32 (che sono parte di UWP). Queste API sono implementate da tutte le famiglie di device Windows 10. Per utilizzarle basta collegare l'applicazione con Windowsapp.lib, la quale copre il ruolo di "ombrello" che provvede a esportare Win32 per UWP.

3.4 Windows IoT Core

In questa sezione verranno introdotte alcune caratteristiche del Core ideato per Internet of Things di casa Microsoft, con uno speciale occhio di riguardo per la parte concernente l'utilizzo di tale piattaforma con il MCU Arduino UNO.

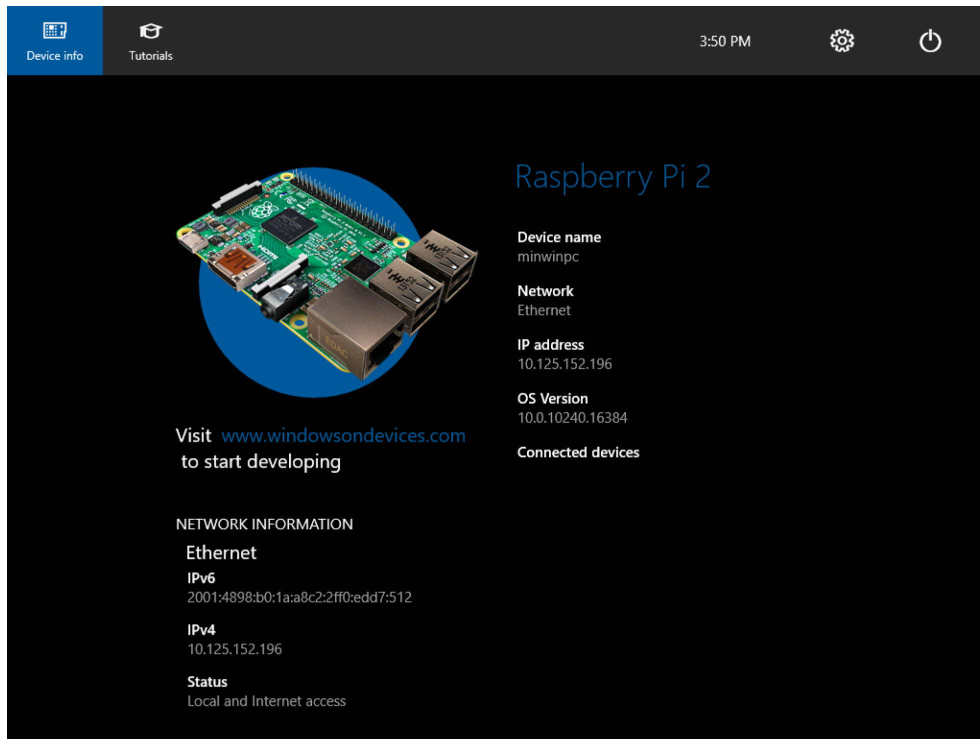


Figura 3.4: Default Windows IoT Core app

3.4.1 Modalità Headed e Headless

Windows IoT Core può essere configurato in modalita **Headed** o in modalità **Headless** [13]. La differenza tra queste due modalità consiste nella presenza/assenza dell'interfaccia grafica. Di default, Windows 10 IoT Core si trova in modalità Headed e, all'avvio, lancia una applicazione predefinita (figura 3.4) che mostra a video le informazioni riguardanti il sistema e il dispositivo (ad esempio lindirizzo IP). Inoltre, in modalità Headed, l'intero *stack UWP UI* è disponibile per app interattive. Quando il dispositivo viene impostato in modalità headless, lo stack si disabilità e le app perdono ogni tipo di interattività. Le risorse utilizzate dal sistema, in questo stato, risultano diminuite sensibilmente. Quando un dispositivo si trova in headed mode, all'avvio viene lanciata una singola app UI e non vi è alcun meccanismo per effettuare switch verso altre applicazioni (eccezion fatta per il debugger di Visual Studio che, quando deve compiere la run di

un'applicazione, la pone in primo piano automaticamente).

3.4.2 Architetture e Progettazione di sistemi IoT con Windows

Con l'avvento UWP, è ora possibile pensare un'applicazione IoT (per Windows 10 IoT Core) come un tipo di applicazione non troppo differente da una qualsiasi altra app Windows (per desktop/tablet). In questo senso Microsoft non ha cercato di indirizzare gli sviluppatori verso una direzione particolare, anzi, si è limitata a costruire un insieme di API per il corretto utilizzo della parte di GPIO (esclusiva di dispositivi quali Raspberry e similari). Anche per la parte Arduino (spiegata nel dettaglio successivamente) sono state costruite un'insieme di primitive per la gestione degli I/O da e verso il device (classiche read/write di pin, callback di eventi specifici...). Si presume, quindi, che debba essere il progettista a pensare un sistema IoT costruito su questa piattaforma in termini più astratti, sfruttando il paradigma a oggetti per un design di più alto livello.

3.4.3 Windows Remote Arduino

Windows Remote Arduino è un componente della libreria Runtime di Windows (open source), che permette il controllo del microcontrollore tramite Bluetooth, USB, connessione Wifi o Ethernet. L'aggiunta di questo componente ad un progetto IoT, comprendente Arduino, darà accesso automatico alle sue funzionalità attraverso uno dei linguaggi WinRT(C++, CX, C# e JavaScript). La libreria è stata costruita basandosi sul protocollo **Firmata**, il quale è implementato in un pre-package incluso nel software nativo di sviluppo Arduino (*Arduino IDE*). Windows Remote Arduino va a inserirsi nel gap tra il mondo fisico e il software, consentendo diverse funzionalità, quali:

- **GPIO - I/O analogici e digitali**
  - *Digital Write*
  - *Digital Read*
  - *Analog Write*
  - *Analog Read*

  - *Settaggio pinMode*
  - *Ricezione di eventi* quando i valori cambiano/sono riportati

- **I^{2}C**
  - *Invio/Ricezione dati tra devices via I^{2}C*
  - *Sottoscrizione a device, queries ripetute e reporting automatico*

3.4.4 Firmata

Firmata [15] è un protocollo per la **comunicazione** tra software (su un host computer) e microcontrollori. Può essere implementato nel firmware di qualsiasi architettura di microcontrollori, così come in un package interno ad un software (vi sono numerose implementazioni per diversi linguaggi di programmazione in rete). Questo protocollo utilizza il formato di messaggi MIDI, e i messaggi scambiati sono lunghi 2 byte (ad esempio il comando 0xD0, *Channel Pressure message*). I messaggi più frequentemente utilizzati da Firmata sono i **Midi System Exclusive messages** (Sysex), ognuno dei quali è identificato da una codifica univoca.

Utilizzo di Firmata in Arduino

Ci sono due modelli principali per utilizzare Firmata attraverso Arduino [16]:

1. Lo sviluppatore dello sketch Arduino utilizza i vari metodi forniti dalla libreria di Firmata per inviare e ricevere dati tra l'Arduino e il software *running* su un host. Per esempio, un utente può inviare dati analogici all'host usando :

   ```
   Firmata.sendAnalog(analogPin, analogRead(analogPin));
   ```

   oppure inviare i pacchetti in formato Stringa:

   ```
   Firmata.sendString(stringToSend)
   ```

2. Caricare uno sketch chiamato StandardFirmata sull'Arduino, per poi utilizzare l'host computer esclusivamente per interagire con il microcontrollore. (Si può trovare StandardFirmata nellIDE di Arduino tra gli *Examples*). Questa modalità in genere è la più utilizzata.

3.4.5 Architettura di WRA

L'architettura di Windows Remote Arduino [14] si compone di tre livelli: **interfaccia**, **protocollo** e **trasporto**. Ogni livello è un *cliente* dei livelli inferiori e dipende da questi ultimi. Viceversa, nessun livello dipende dai livelli soprastanti. Nello specifico, questi tre livelli sono:

- **RemoteWiring** (interfaccia)
- **Firmata** (protocollo)
- **Seriale** (trasporto)

Per esempio, il livello Firmata traduce le richieste del RemoteWiring tramite il protocollo, per poi passarle allo strato seriale, il quale ha il compito di trasportarlo da/a Arduino. Lo strato Firmata non ha alcuna conoscenza dell'implementazione dello strato seriale, ma dipende strettamente dalla sua interfaccia per poter funzionare correttamente.

RemoteWiring

La classe principale di questo strato è **RemoteDevice**, che rappresenta il punto d'accesso all'Arduino. Essa deve essere utilizzata nella maggior parte dei casi ed offre un'interfaccia con metodi quanto più simili possibile a quelli che si utilizzerebbero nelle Arduino Wiring API. Tutte le chiamate a questo livello vengono redirette allo strato Firmata, perciò se si desidera implementare comportamenti avanzati è necessario estendere o bypassare RemoteWiring (andando a contemplare direttamente lo strato UwpFirmata).

UwpFirmata

L'implementazione di questo strato è *wrapped* da un componente della libreria Windows Runtime chiamato **UwpFirmata**. Questa classe non modifica

o aggiunge funzionalità, ma si limita ad aggiustare i parametri (esempio: *char** viene trasformato in *String*) e a tradurre i paradigmi (esempio: *callbacks* diventano *events*). Il vantaggio che UwpFirmata fornisce è l'abilità di leggere gli input da un Thread in background, invece di essere costretto dall'esecuzione di un singolo main Thread (grazie alle funzionalità di Windows questo diventa infatti possibile).

Seriale

É lo strato di trasporto che provvede alla comunicazione fisica tra le applicazioni e l'Arduino. IStream è linterfaccia che definisce i requisiti di uno stream per questa comunicazione. Attualmente questa classe è implementata dalla classe di default BluetoothSerial, USBSerial e NetworkSerial (per le comunicazioni seriali su Windows 10, in base alla tecnologia utilizzata). IStream presenta 5 funzioni che devono essere implementate da ogni classe che la estende. É importante sottolineare che queste funzioni devono assolutamente essere operazioni *sincrone*

- *void begin(int SerialConfig)*: inizializza lo Stream
- *void end()*: termina lo Stream
- *int available(void)*: ritorna il numero di byte nello Stream
- *short read(void)*: legge un singolo carattere dallo Stream (1 byte)
- *void write(char)*: scrive un singolo carattere nello Stream (1 byte)

Capitolo 4

Progetto: Smart Parking

4.1 Introduzione

Per lo svolgimento della parte progettuale, è stato necessario prendere confidenza con la tecnologia impiegata per lo sviluppo: Windows 10 IoT Core e Windows Remote Arduino.

4.1.1 Prove preliminari della tecnologia

Sono stati riprodotti alcuni degli esempi forniti dagli sviluppatori sul sito ufficiale Microsoft:

- **Blinky App versione USB**, con interfaccia grafica, per comandare i pin di arduino, attraverso una UWP operativa su pc Desktop Windows 10. In questo caso è stata utilizzata la classe *USBSerial* per la comunicazione Computer-Arduino.

- **Blinky App versione Wifi**, applicazione analoga alla precedente. In questo caso è stato testato il funzionamento dello shield wifi e della comunicazione attraverso la classe *NetworkSerial*

.

Al termine dello studio preliminare delle basi per l'applicazione pratica della tecnologia, si è pensato di sviluppare un prototipo di soluzione IoT in ambito Smart City: un **Parcheggio Intelligente**.

4.1.2 Materiale utilizzato

Per la realizzazione dello **Smart Parking**, è stato utilizzato il seguente elenco di materiale:

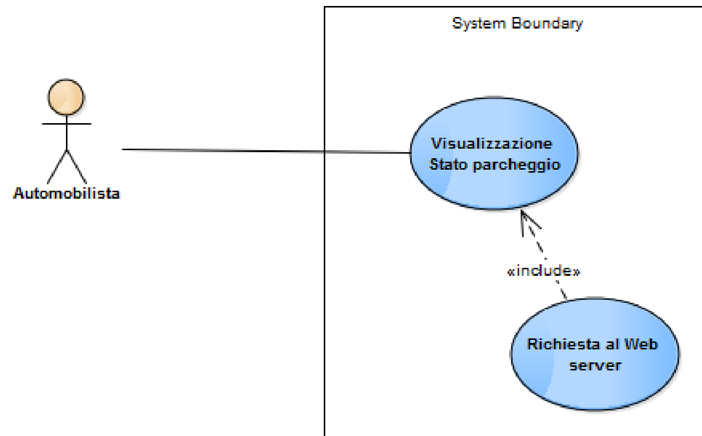
- 1x **Raspberry Pi 2**
- 2x **Arduino UNO**
- 2x **Shield Wifi per Arduino**
- 1x **Breadboard**
- 2x **Sensori di prossimità HC-SR04**
- **Jumpers**
- 1x **Cavo Ethernet**

4.2 Requisiti

"Si intende realizzare un sistema IoT per la visualizzazione del numero di posti liberi in un parcheggio intelligente. Ad ogni singolo posto auto è assegnato un Arduino, dotato di un apposito sensore di prossimità per la rilevazione della presenza/assenza di un'automobile. Sarà presente un Raspberry Pi 2 centrale, incaricato della memorizzazione e gestione delle connessioni ad ogni singolo posto auto (in questo progetto inseriti a mano dallo sviluppatore). Ogni singolo parcheggio può risultare libero o occupato. Raspberry fornisce un Web Server per la visualizzazione dello stato del parcheggio (alla porta designata)."

4.3 Analisi

Partendo dalle specifiche riportate, si procede in questa fase con una prima analisi orientata alle **funzioni**, agli **oggetti** e agli **stati**.



Use Cases.png

Figura 4.1: Use Case Diagram

4.3.1 Analisi Funzionale

Dal diagramma dei Casi D'Uso in figura 4.1 emergono le funzionalità principali del progetto in relazione agli utenti che lo utilizzano.

Smart Parking è pensato come un sistema autonomo che fornisce ad un generico **Automobilista** il *servizio* di poter visualizzare lo stato del parcheggio gestito. L'unica operazione che l'automobilista deve compiere è quella di richiederne lo stato (nel caso del progetto, via HTTP, quindi si presuppone attraverso un browser Web).

4.3.2 Analisi orientata agli Oggetti

Analizzando gli elementi enunciati nei requisiti, è stato prodotto uno schema strutturale di massima (figura 4.2).

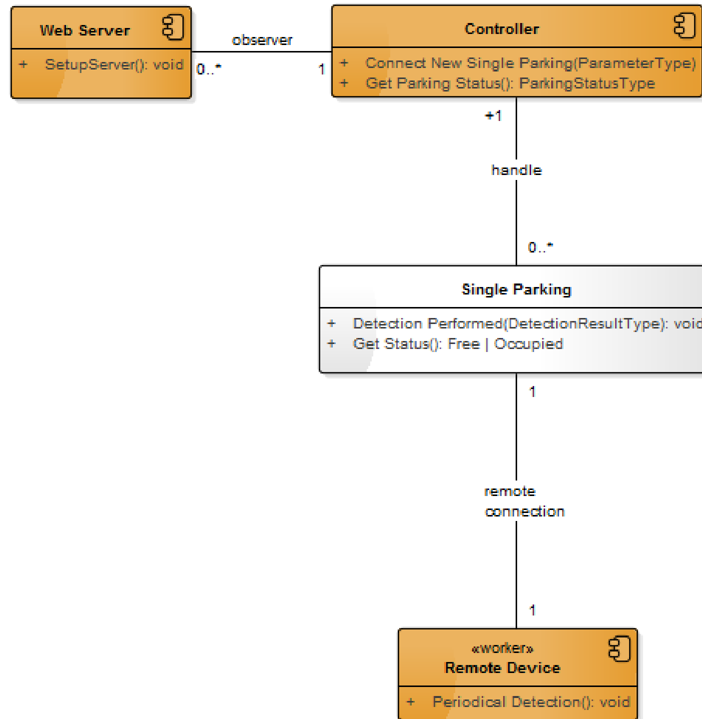


Figura 4.2: High Level Structural Diagram

In questo diagramma si rilevano 3 componenti fondamentali del sistema:

- **WebServer**: incaricato della visualizzazione grafica dello stato del parcheggio. Ha il compito di rimanere in attesa di richieste(ascoltando ad una porta predefinita) e di servirle restituendo indietro una rappresentazione dello status del parcheggio (via HTML)

- **Controller**: interpretato, in questo sistema, da Raspberry. Ha il compito di memorizzare i riferimenti a tutti i dispositivi fisici e di poterli interrogare per scoprire lo status del posto auto a loro asso-

ciato (se *libero* o *occupato*). La modalità di effettuazione di questa interrogazione, verrà trattata in seguito.

- **Remote Device**: Dispositivo fisico assegnato ad uno specifico posto auto del parcheggio. É provvisto di un sensore di prossimità. Deve poter effettuare misurazioni ed inviarne il risultato al Raspberry tramite la rete.

Oltre ai componenti descritti, ad unire il Device fisico e il Controller, troviamo l'entità **Single Parking**, che rappresenta un'*astrazione* di un singolo posto auto, incapsulandone la logica e gli stati. É corretto pensare alla possibilità che , in un'eventale altra applicazione dello Smart Parking, si possa voler cambiare l'hardware incorporato in ogni posto auto (per esempio inserendo Raspberry al posto dell'Arduino, o comunque altre schede supportate da Microsoft). Per ovviare a questa evenienza, in fase di progettazione si introdurrano concetti ad hoc.

4.4 Progettazione

In questa fase vengono raffinati tutti i concetti e le dinamiche presentate in fase d'analisi, al fine di muovere un passo in avanti verso l'implementazione vera e propria del progetto.

4.4.1 Progettazione delle classi

Basandosi dal diagramma dei componenti e considerando alcuni accorgimenti in ambito di astrazione/estendibilità menzionati, è stato prodotto il diagramma delle classi (figura 4.3)

WebServer

Questa classe, in questo specifico caso, svolge il ruolo di *View*. Espone un metodo per l'inizializzazione e deve poter rimanere in attesa di richieste http su una porta predefinita, per poi fornire una rappresentazione dello stato del parcheggio in linguaggio HTML. Dalle sue relazioni è chiaro che dovrà interfacciarsi al controller per ottenere le informazioni che le servono.

ParkingController

Poiché ci si aspetta di avere un controller unico, si può pensare di applicare il pattern **Singleton** a questa classe. Espone i metodi necessari per ottenere lo stato del parcheggio e anche un altro metodo utile ad aggiungere connessioni di dispositivi al sistema (e, di conseguenza, di altri posti auto).

Single Parking

Come enunciato in precedenza, questa classe non è che un'astrazione di un posto auto fisico. E' stata introdotta per semplificare le operazioni al controller e per fornire un distacco tra l'implementazione delle connessioni con i devices a basso livello e la logica ad alto livello del parcheggio. Una volta impostati indirizzo e tipo di device al quale connettere il posto auto, il controller può preoccuparsi solo di osservare il SingleParking , considerandolo come un'entità con due possibili stati (Libero o Occupato), il quale, in caso di cambio di stato, si incaricherà di notificarlo tramite uno specifico evento.

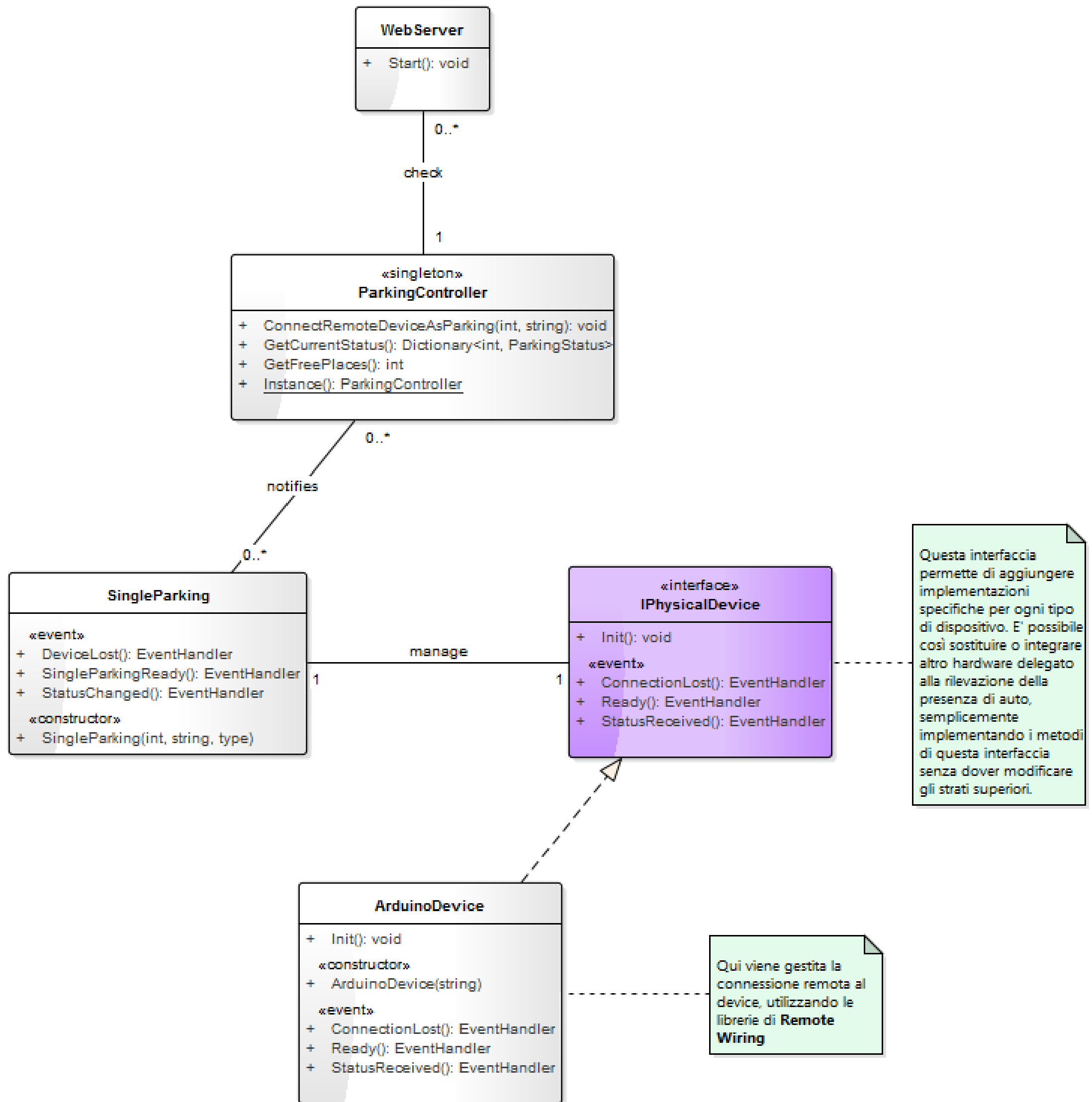


Figura 4.3: Class Diagram

IPhysicalDevice

Interfaccia non contemplata in precedenza tra i componenti, viene introdotta nello schema al fine di permettere future estendibilità del programma. Grazie ad essa viene nascosta ai livelli superiori l'implementazione della connessione all'hardware specifico. Poiché SingleParking opera attraverso di essa, l'integrazione di altri dispositivi, e le relative implementazioni, non vanno a modificare la sua logica, blindando letteralmente i livelli superiori a IPhysicalDevice.

ArduinoDevice

Implementazione specifica per la gestione della connessione verso un arduino remoto dotato di sensore di prossimità. In questa classe viene incapsulato l'utilizzo della libreria **Remote Wiring** (ampiamente trattata nei capitoli precedenti).

4.4.2 Progettazione delle associazioni

Partendo dal diagramma delle classi mostrato in fase di Analisi, nello schema in figura 4.4 vengono progettate le associazioni tra le varie classi.

Da notare come, poiché si è scelto di applicare il pattern *Singleton* alla classe ParkingController, non è stato necessario aggiungere riferimenti alla classe WebServer.

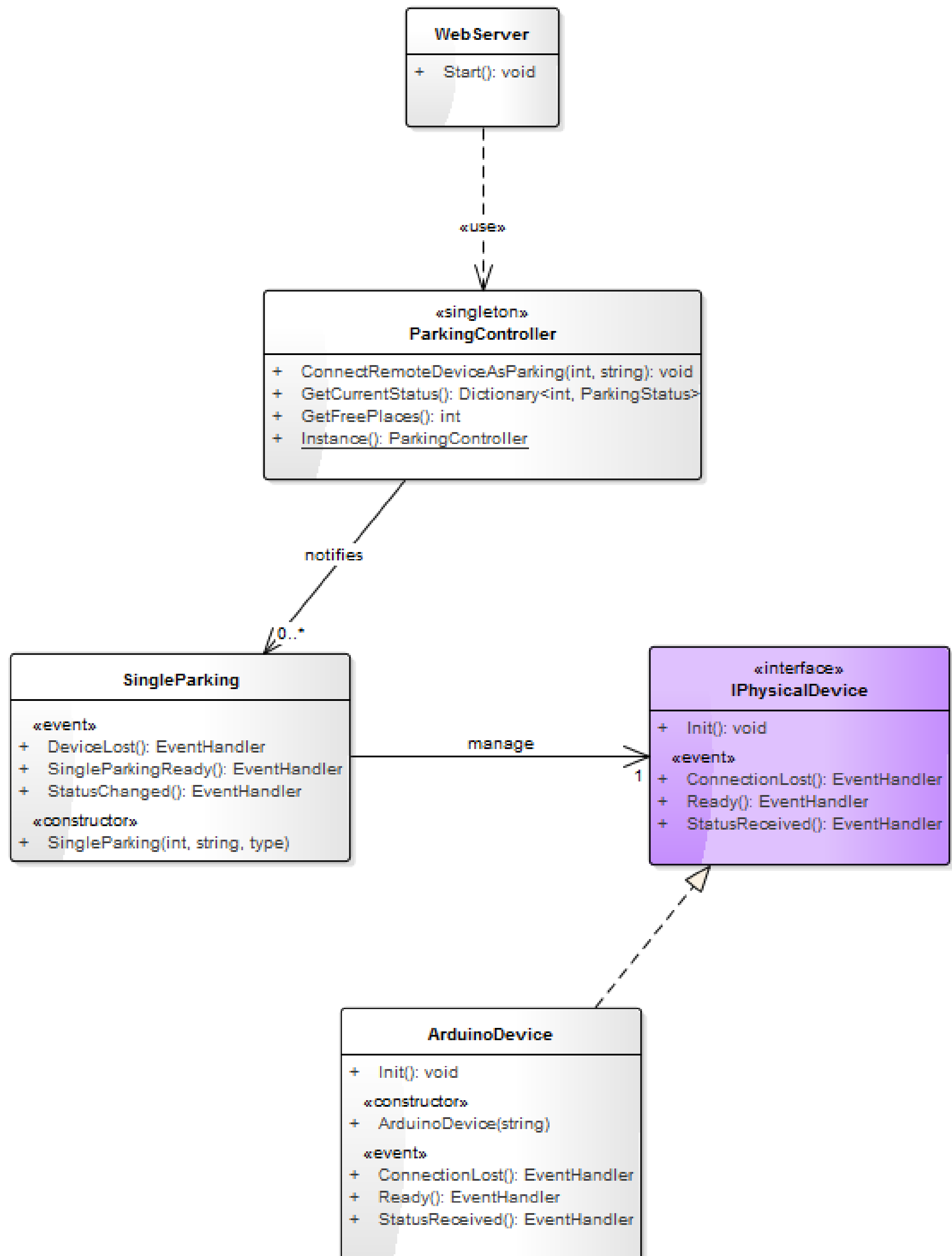


Figura 4.4: Association Diagram

4.4.3 Progettazione degli Stati: SingleParking

E' necessario, come da specifiche, modellare lo status di un singolo posto auto, il quale può risultare **Free** o **Occupied**. Nel diagramma degli stati in figura 4.5, vengono esposti anche i *trigger events* per il cambio di stato, relativi alla classe che incapsula la logica del posto auto: **Single Parking**. Questi eventi possibili sono due:

- **Ricezione di "f"**: Arduino invia un byte rappresentante tale lettera quando il sensore rileva una distanza sufficientemente grande da poter considerare un parcheggio *libero*
- **Ricezione di "o"**: Arduino invia un byte rappresentante tale lettera quando il sensore rileva una distanza sufficientemente piccola da poter considerare un parcheggio *occupato*

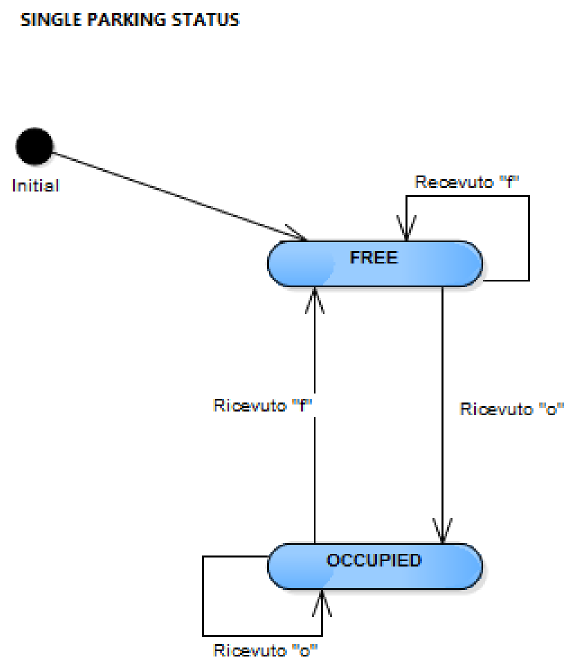


Figura 4.5: Single Parking: StateChart Diagram

4.4.4 Rilevazione tramite sensore: messaggi scambiati

Come anticipato, l'intero programma si basa su una logica ad **Eventi**. Vi è una fase di inizializzazione di tutti i componenti (e delle connessioni), per poi passare ad una fase di "Ascolto e Aggiornamento". Infatti, una volta pronti tutti i collegamenti ai singoli posti auto, saranno i device dotati di sensore (Arduino, nel nostro caso specifico) ad effettuare controlli **periodici** seguiti poi, dall'invio di una notifica verso il sistema di controllo centrale. In questa sezione si analizza il *viaggio* di una singola notifica attraverso gli strati della nostra architettura.

Come si nota dall'Analisi orientata agli Oggetti, tutte le classi che si diramano al di sotto del controller forniscono degli *event* triggerabili. É attraverso questi che viene trasportato un singolo messaggio. Partendo dal device fisico, la notifica ("o" oppure "f", in base allo stato del posto auto) viene ricevuta dalla classe che implementa IPhysicalDevice (nel nostro caso ArduinoDevice). La classe **RemoteDevice**(fornita dalle librerie di remote Wiring) fornisce un metodo per intercettare l'invio di stringhe da parte dell'Arduino e, implementando IPhysicalDevice, *trasferisce* il messaggio, così com'è, alla classe SingleParking. SingleParking, una volta ricevuta la notifica, controlla internamente il suo stato precedente (libero o occupato) e, solo nel caso in cui lo stato ricevuto sia diverso dall'ultimo, invia una callback al controller attraverso l'evento *StatusChanged*. Il controller provvede, infine, ad aggiornare lo status globale del parcheggio. Tutti i passaggi di questo scambio sono rappresentati nello schema di sequenza in figura 4.6.

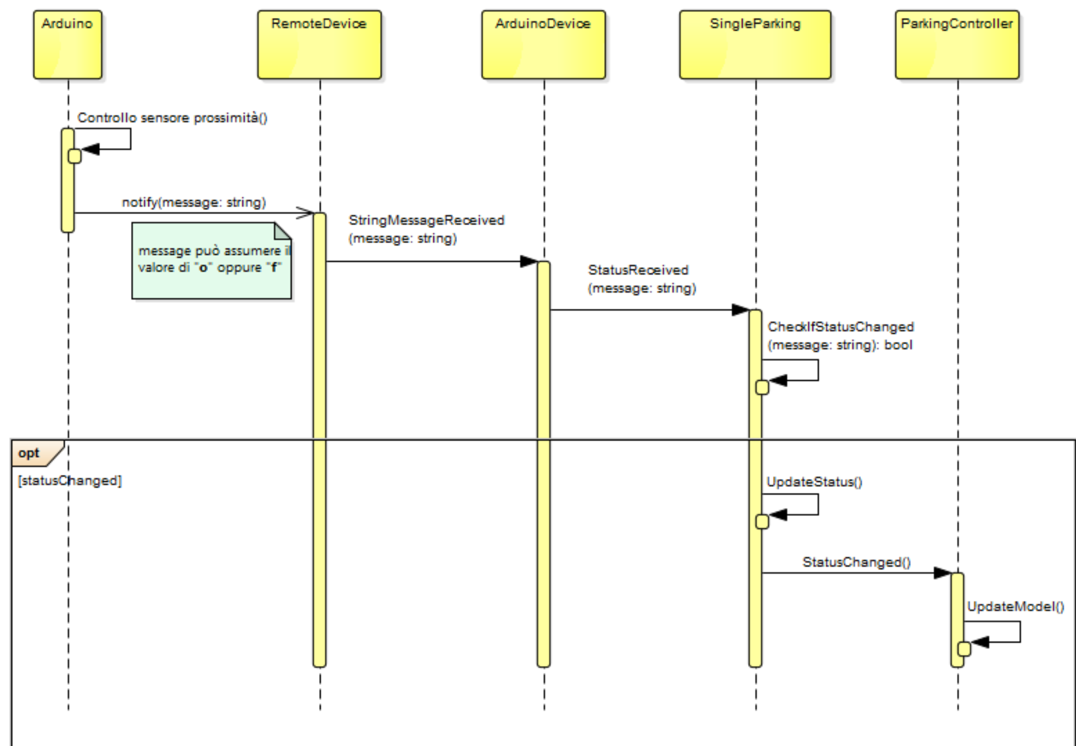


Figura 4.6: Sensor notify: Sequence Diagram

4.4.5 GET al server per la visualizzazione dello stato: messaggi scambiati

La modalità fornita dal programma per la visualizzazione dei posti occupati (e non) del parcheggio, consiste nel richiedere, via HTTP, al server in ascolto sul Raspberry, un documento HTML contenente tutte le informazioni. In questo processo, quindi, sono presenti anche interazioni tra la classe **WebServer** e il singleton **ParkingController**. Nello specifico, non appena il server viene invocato a seguito di una GET, richiede al controller tutti i dati correnti riguardo al parcheggio. Una volta processati questi dati e prodotto una stringa contenente HTML, provvede a rispedire al mittente un messaggio HTTP: *200 OK*. Anche in questo caso, per meglio comprendere come esattamente avvengono tutti i passaggi, è stato prodotto un diagramma di sequenza esplicativo (figura 4.7).

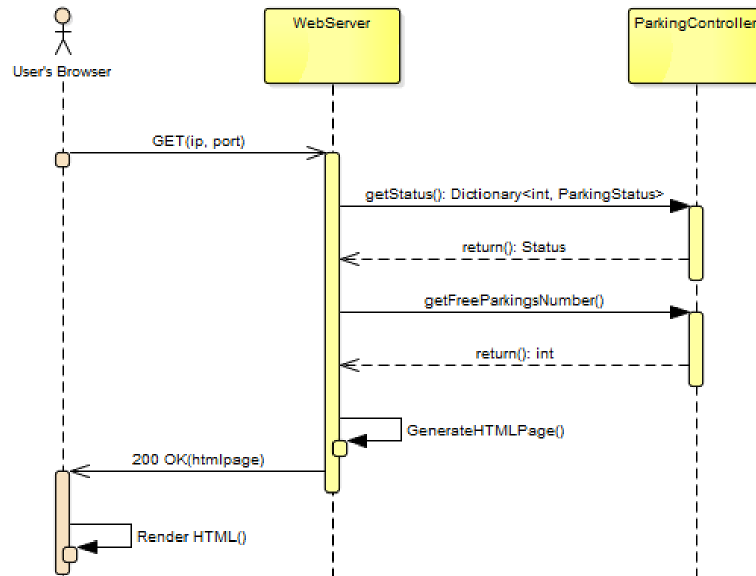


Figura 4.7: Request to Server: Sequence Diagram

4.5 Implementazione

In questa sezione verranno mostrate le implementazioni di alcune parti del progetto, anche in relazione alla progettazione effettuata a monte.

4.5.1 Lato Arduino

Come trattato nel capitolo precedente, per utilizzare Windows IoT Core e Arduino, è necessario contemplare il protocollo Firmata per attivare una comunicazione. Windows Remote Arduino prevede il caricamento all'interno dell'MCU di uno sketch ad hoc: **Stardard Firmata**. Dato che in questo progetto verranno utilizzati degli shield Wifi per la connessione remota degli Arduino, StandardFirmata non ci permette di lavorare col device nel modo corretto. É stato invece caricato negli Arduino **Standard Firmata Wifi** una variante apposta per reti wireless. Semplicemente modificando *ssid name* e *password* all'interno dello sketch, il dispositivo riesce ad accedere alla rete e rimanere in ascolto di connessioni verso se stesso.

Dopo aver collegato all'Arduino il sensore di prossimità **HC-SR04**, è stato necessario implementare un metodo per effettuare un controllo periodico sul sensore, al fine di rilevare o meno la presenza di una macchina nel parcheggio. Per farlo è stata inserita una funzione all'interno del **loop** di controllo e il periodo tra un controllo ed il successivo è stato scelto in modo arbitrario dallo sviluppatore. Di seguito il codice della funzione aggiunta:

```
if(millis - lastDetection > DETECT_PERIOD){
    long duration, distance;
    digitalWrite(trigPin, LOW);
    delayMicroseconds(2);

    digitalWrite(trigPin, HIGH);
    delayMicroseconds(10);

    digitalWrite(trigPin, LOW);
    duration = pulseIn(echoPin, HIGH);

    //Calculate the distance (in cm) based on the speed of sound.
    distance = duration/58.2;
```

```
    if(distance >= PARKING_LENGHT){
      Firmata.sendString("f");
    } else {
      Firmata.sendString("o");
    }

    lastDetection = millis;
  }
```

Non è stato alterato lo sketch originale in nessun altro modo e, alla luce dei test, tutto ha funzionato a dovere.

4.5.2 Lato Raspberry Pi 2

Si è deciso di porre il Raspberry in modalità **Headless**, ovvero privandolo di interfaccia grafica, in quanto, per il caso di studio corrente, la visualizzazione delle informazioni avviene soltando attraverso richiesta al server web da parte di altri dispositivi. Anche per questo, come *template* di progetto, è stato scelto uno tra quelli nuovi proposti ad hoc da Visual Studio per Internet of Things: **Background Application** (figura 4.8).

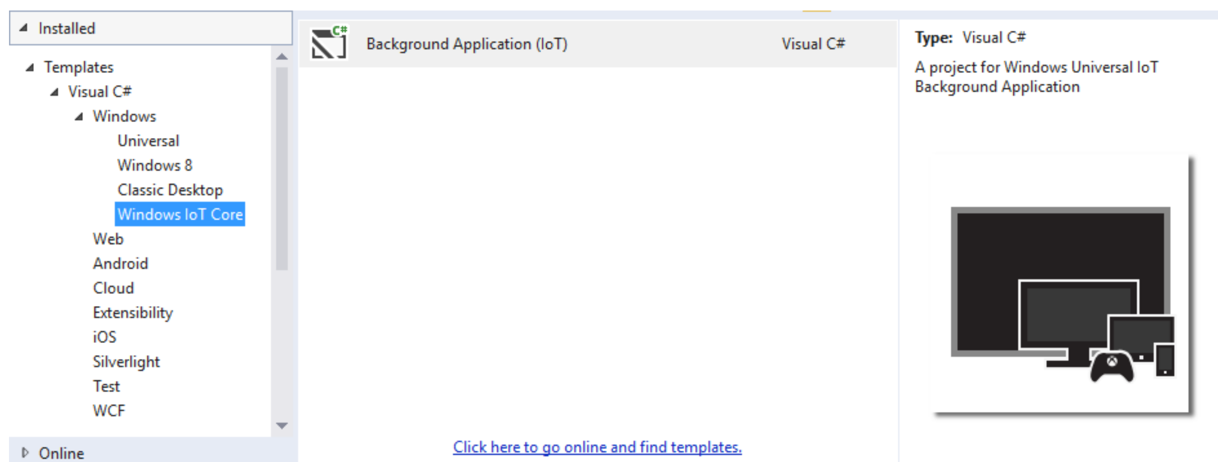


Figura 4.8: Background Application Template

Una Background Application è caratterizzata da un main Task che si avvia non appena viene lanciata. Viene chiamato **Startup Task**, estende **IBackgroundTask** ed è proprio in questo che viene eseguita l'inizializzazione dei componenti dell'applicazione.

Inizializzazione WebServer

Come mostra il diagramma delle classi in figura 4.3, la classe *WebServer* espone un metodo 'Start()" per attivare l'ascolto delle richieste HTTP entranti ad una data porta. Per l'implementazione via codice, è bastato associare uno **StreamSocketListener** direttamente alla porta desiderata tramite binding e gestire le richieste entranti asincrone, triggerando l'*event* corrispondente.

```
public async void Start()
       {
           StreamSocketListener listener = new
              StreamSocketListener();

           await listener.BindServiceNameAsync(ListeningPort);

           listener.ConnectionReceived += async (sender, args) =>
           {

                      //Handle here the request received

                      //Then send a response

                }

           };
```

Inizializzazione e connessione ad un Arduino assegnato ad un posto auto

La classe ParkingController espone un metodo atto a connettere un nuovo device (il quale si trova già connesso nella rete wireless) al sistema, così da aggiungere un nuovo posto auto al monitoraggio. La signature di questo metodo è la seguente:

```
void ConnectRemoteDeviceAsParking(int parkingNumber, string
    ipAddress)
```

L'implementazione di questo metodo è molto semplice: il controller si incarica di aggiungere al suo modello una nuova istanza di SingleParking, *senza* però inserirla nello **status globale**. Il costruttore della classe SingleParking, come unica azione, tenta di connettersi ad un device remoto, e lo fa interfacciandosi (come già spiegato in precedenza) alla classe IPhysicalDevice. Prima imposta i parametri (quali indirizzo IP), dunque effettua la chiamata a *Init*.

Poiché vi possono essere più di una tipologia di device connessi per la rilevazione tramite sensore, al costruttore di SingleParking viene anche passato una variabile di tipo **PhysicalDeviceType**, ed è questo parametro, gestito da uno *switch*, a decidere quale implementazione di IPhysicalDevice utilizzare per quel preciso SingleParking. Nel caso di una connessione ad un Arduino Remoto, si utilizzano le librerie apposite di Windows Remote Arduino(opportunamente incluse nel progetto). Di seguito viene mostrato il metodo *Init()* implementato dalla classe **ArduinoDevice**.

```
public void Init()
        {
            _connection = new NetworkSerial(new
               HostName(_ipAddress), DefaultArduinoConnectionPort);
            _firmata = new UwpFirmata();
            _arduino = new RemoteDevice(_firmata);
            _firmata.begin(_connection);
            _connection.begin(9800, SerialConfig.SERIAL_8N1);
```

```
        _connection.ConnectionEstablished +=
            ConnectionEstablished;
        _connection.ConnectionLost += OnConnectionLost;
        _arduino.StringMessageReceived +=
            ArduinoOnStringMessageReceived;
    }
```

Come mostrato da codice, vengono inizializzati i 3 strati di Remote Wiring: **RemoteDevice**, **UwpFirmata** e **NetworkSerial**. Una volta invocata la *begin()*, vengono associati i metodi di callback per i possibili messaggi ricevuti da parte di Arduino.

Non appena l'IPhysicalDevice richiama l'evento asincrono *Ready*, viene propagata la notifica fino al controller, che provvede ad inserire la precisa istanza di SingleParking nello status globale.

4.5.3 Reattività agli imprevisti: robustezza

É possibile pensare che, durante l'operatività dell'intero sistema, possa accadere che uno dei device assegnati ai singoli parcheggi abbia un malfunzionamento dovuto, per esempio, ad un crash o ad una mancanza di corrente. In questo caso è necessario fornire due funzionalità:

- **Informare l'utente** del malfunzionamento relativo ad un posto auto (indicando, ad esempio, il numero del posto auto che ha riscontrato il problema.)

- **Attivare** un meccanismo di **auto-correzione** della situazione critica dopo ripristinato il dispositivo non funzionante

Per rispettare questi due punti, nel progetto è stato inserito all'interno del main task un **controllo periodico** della ricezione dei report riguardanti i sensori da parte dei device. Se, per esempio, viene rilevato un intervallo di tempo troppo grande dall'ultima rilevazione, si suppone che il dispositivo abbia subito un problema. L'intero ciclo di operazioni è il seguente:

1. Il controllo del periodo avviene all'interno della classe SingleParking, responsabile del proprio dispositivo.

2. In caso di rilevamento di errore, SingleParking aggiorna i suoi parametri interni, dunque invia un callback al controller tramite l'evento *DeviceLost*.

3. Il controller aggiorna il suo status interno in modo tale da **non mostrare più il parcheggio mal funzionante**.

4. Il SingleParking ritenta una connessione verso il suo dispositivo.

5. Se la connessione ha successo, tutto il sistema, tramite i callbacks di connessione, viene riaggiornato come spiegato nelle sezioni precedenti.

Le operazioni elencate dalla 1 alla 4 avvengono in modo **sincrono**. Questa scelta è voluta in quanto, in eventuali scenari provvisti di centinaia di parcheggi, si preferisce ritardare la rilevazione di errori di qualche secondo piuttosto che sovraccaricare il sistema con troppi thread asincroni separati, ognuno assegnato al controllo di un parcheggio.

4.6 Collaudo

4.6.1 Modulo software su Arduino

Su Arduino è stato montato uno speciale sketch analogo a **StandardFirmata**, ma ad hoc per la connessione Wifi tramite Shield: **Standard Firmata Wifi**.

E' stato sottoposto a test tramite **Applicazioni di prova** prima di essere inserito nel progetto.

Collaudo delle funzioni di Firmata

Dal testing delle varie funzionalità che questo sketch ci offre, è stato rilevato un malfunzionamento relativo ai messaggi **entranti**. Tramite il corrispondente strato UwpFirmata sull'applicazione Windows, si è tentato di inviare dei messaggi all'Arduino denominati **Sysex**, ovvero messaggi speciali identificati da un byte univoco per ogni tipo di messaggio (vi sono alcuni tipi di messaggi Sysex preimpostati, e altri customizzabili).

Non è stato possibile in nessun modo ricevere da parte dell'Arduino alcun messaggio sysex personalizzato inviatogli (seppur seguendo il protocollo di

utilizzo avanzato dello strato Firmata). Non è stata compresa la natura del problema(a causa dell'obbligo di utilizzo dello sketch Standard Firmata Wifi, sviluppato da altre persone).

Tuttavia, i messaggi **uscenti** verso il Raspberry non hanno subito alcun tipo di errori, e tutti sono stati inviati e ricevuti correttamente.

Neanche l'inserimento nel loop di controllo dello sketch Firmata di un controllo periodico sul sensore interferisce con l'attività di comunicazione.

Prestazioni

A sistema avviato, la **connessione** da parte del Raspberry con l'Arduino avviene in modo abbastanza celere: all'incirca in **1-2s**.

La frequenza di invio messaggi con lo status è impostata, nel caso del progetto, a 5 secondi. Il sensore di prossimità per effettuare una rilevazione impiega all'incirca 20-25 microsecondi.

4.6.2 Background Task su Raspberry

Il flusso di controllo di questa Applicazione consiste in un **MainThread** che periodicamente svolge l'attività di **controllo di integrità delle connessioni** in atto verso i device assegnati ai posti auto. Oltre a questo, vengono generati dalle librerie di Remote Wiring processi separati per la gestione dei callback dei messaggi ricevuti, che risalgono lo stack di chiamate agli **event** del sistema fino al controller.

Prestazioni

Il sistema opera su un Raspberry Pi 2 *Model B* con le seguenti caratteristiche principali:

- Processore quad core Cortex A7: 900MHz
- RAM: 1Gb

Il sistema, una volta a regime, si attesta ad un consumo di CPU del Raspberry al 30-50%.

Si consideri che ciò avviene avendo 2 posti auto. Per una futura estensione del numero dei parcheggi, sarebbe consigliabile aumentare il periodo di rilevazione di ogni device fino anche a 30s (comunque verosimile in una

situazione di parcheggio reale), al fine di non sovraccaricare il sistema con troppi messaggi entranti troppo velocemente.

Il consumo di RAM dell'applicazione invece è sui 150 MB massimi in fase di *run*.

4.7 Estensioni

Il sistema, realizzato come progetto, ha lo scopo di essere un **prototipo** di parcheggio intelligente con tutti i parametri adattati al caso specifico in miniatura, ma eventualmente configurabili. Queste premesse sono uno dei motivi per i quali la progettazione dello *Smart Parking* è stata effettuata con attenzione anche all'**estendibilità** del software. Di seguito vengono analizzate alcune possibili estensioni, applicabili in riferimento alle scelte fatte in fase di Analisi e Progettazione.

4.7.1 Sostituzione, integrazione e aggiunta di differenti tipi di dispositivo

Windows IoT Core ad oggi supporta i seguenti dispositivi:

- **Raspberry Pi 2**
- **MinnowBoard MAX**
- **Intel Galileo**
- **Arduino**

In Smart Parking vengono utilizzati soltanto due di questi dispositivi. Sarebbe quindi possibile, eventualmente, estendere l'applicazione assegnando ai posti auto altri dispositivi *comunque* capaci di gestire un sensore di prossimità. **IPhysicalDevice** è l'interfaccia ad hoc per questo tipo di evenienza. Per implementare la gestione globale di un Arduino, è stata implementata la classe **ArduinoDevice**. Un'altra eventuale implementazione di IPhysicalDevice (ad esempio per Galileo) non dovrebbe avere altro che la connessione al dispositivo remoto (all'interno del metodo *Init()*) e la gestione dei callback di ricezione status. Così facendo non viene alterata in alcun modo la logica di funzionamento del programma e nessuna modifica deve essere apportata agli altri componenti.

4.7.2 Cambio di modalità di visualizzazione dei dati

Smart Parking fornisce la visualizzazione dello stato del parcheggio via HTML (figura 4.9), previa richiesta GET all'indirizzo del Raspberry e ad una porta predefinita. In ottica Internet of Things, chiaramente, l'attivazione di un Web Server interno è soltanto uno dei numerosi modi per fornire un'interfaccia grafica delle informazioni processate dal dispositivo *centrale*.

Smart Parking

Free Parkings: 2

Parking Number	Status
1	Free
2	Free

Figura 4.9: Parking Status View

Per questo motivo si osserva che, per inserire nell'applicazione un qualsiasi altro componente dedito a provvedere una *View*, l'implementazione **separata** del controller dal Web Server diventa cruciale. ParkingController, una volta inizializzato, è un **Singleton**, ovvero la sua istanza (unica per tutta la durata dell'attività del programma) può essere invocata in ogni momento da un qualsiasi componente, e i suoi metodi pubblici consentono a quest'ultimo di ricavare lo status globale del parcheggio. Questo quindi vale nel caso in cui volessimo, per esempio, cambiare la modalità del Raspberry in **Headed**, importando il Core del progetto in una applicazione UWP. Una *Page* avente il compito di mostrare all'utente lo stato del parcheggio, potrebbe semplicemente richiamare il controller ed interrogarlo senza dover modificare nulla dei livelli inferiori, dediti alla gestione dei messaggi e delle connessioni coi i dispositivi remoti.

La medesima cosa varrebbe anche se non si volesse visualizzare direttamente i dati del parcheggio, ma, ad esempio, inviarli ad un server online in un formato codificato(XML, JSON, etc.). Sarebbe sufficente realizzare un componente dedito alla codifica e all'invio dei dati, ricavati sempre attraverso l'invocazione dell'istanza del controller.

Capitolo 5

Conclusioni

In quest'ultimo capitolo viene espressa un'opinione riguardo la tecnologia studiata e applicata (Windows IoT Core). Il motivo di tali considerazioni è quello di fornire un'idea generale dei vantaggi e degli svantaggi che l'introduzione di tale tecnologia in ambiti Internet of Things (quali, ad esempio, Smart City) potrebbe riscontrare.

5.1 Vantaggi dell'utilizzo di Windows IoT Core

Nelle seguenti sezioni sono elencati i vantaggi dell'utilizzo della tecnologia.

- **Programmazione ad oggetti universale**: utilizzare IoT Core significa programmare applicazioni UWP o BackgroundTask comodamente realizzabili tramite linguaggi orientati agli oggetti (quali *C#*). Anche per i device come Arduino vale lo stesso, in quanto, attraverso Remote Wiring, diventa possibile gestire input e output dei singoli pin direttamente dall'Applicazione. Vi è dunque un interessante risvolto: Windows IoT Core fornisce agli sviluppatori strumenti per programmazione multiThread e concorrenza anche in ambito di microcontrollori, estendendo le potenzialità del controllo di un Arduino, dotato nativamente di un solo processore e senza un sistema operativo che gestisca i processi (se non creato ad hoc, chiaramente). Infine, ciò porta un grande vantaggio anche in fase di **Progettazione**. Permette infatti di pensare più dispositivi, connessi insieme e distanti tra loro,

come un unico sistema. Analogamente al nostro progetto Smart Parking, la progettazione e l'implementazione non va diversificata tra i singoli componenti hardware. Unire in questo modo la progettazione fornisce grandi vantaggi, quali **minor complessità** e **possibilità di maggiore astrazione**.

- **Instaurazione veloce della connessione grazie a Remote Wiring**: uno degli aspetti più utili ad uno sviluppatore della libreria in questione, è certamente quello che concerne la connessione tra un dispositivo dotato di sistema operativo Windows 10 e un Arduino. In alternativa, con un dispositivo Linux (per esempio Raspberry dotato di *Raspbian*), la connessione dei due dispositivi risulterebbe sicuramente più lunga e complessa da implementare. Con Windows IoT Core e Remote Wiring, una volta settati i parametri di connessione, basta invocare il metodo *Begin()* per attivare in automatico i vari strati dell'architettura, allacciando autonomamente la connessione al dispositivo remoto (via Usb, Bluetooth o Internet).

- **Portabilità delle applicazioni sviluppate**: grazie all'avvento delle applicazioni UWP, la totalità delle applicazioni windows che non utilizzano API specifiche per IoT (che **non** comprendono Remote Wiring, per esempio, ma soltanto per API di GPIO su Raspberry, Galileo, etc.) sono portabili su desktop, mobile e tutti quei dispositivi operanti tramite Windows 10. Questo comporta la possibilità, per esempio, di gestire degli Arduino da remoto da un qualunque tipo di device Windows e, questo, può risultare pratico in alcuni ambiti.

- **Supporto a linguaggi UWP-compatibili**: oltre al linguaggio ad oggetti C# (trattato in precedenza), un'applicazione IoT Windows può essere scritta anche in tutti quei linguaggi di programmazione che la piattaforma universale permette agli sviluppatori:

  - **C++**
  - **Node.js**
  - **Python**
  - **Visual Basic**
  - **Javascript**

- **Utilizzo gratuito**: per agevolare il mercato di IoT, Microsoft ha deciso di rilasciare il sistema operativo gratuitamente agli sviluppatori. Questo permette ad Aziende e a Makers di effettuare delle prove e di realizzare prototipi IoT senza doversi preoccupare dell'acquisto di una licenza.

5.2 Svantaggi nell'utilizzo di Windows IoT Core

Come ogni tecnologia, anche IoT Core di Windows presenta alcuni svantaggi.

- **Poche piattaforme fisiche supportate**: ad oggi il Sistema Operativo Windows 10 IoT Core supporta soltanto pochi tipi di Device: **Raspberry Pi 2**, **Intel Galileo**, **MinnowBoard MAX**, **Arduino UNO**(tramite Windows Remote Arduino). Limita, dunque, le possibilità di scelta per un eventuale progetto che desideri sfruttare questa tecnologia.

- **Rilasciato da poco: conseguenze**: il sistema è stato rilasciato da qualche mese e si potrebbe incappare, quindi, in qualche bug irrisolto nella release corrente. Un'altra conseguenza è la scarsa documentazione in rete. In caso di problemi nello sviluppo è ancora difficile reperire buon materiale specifico per IoT Core.

- **Per sviluppare è necessario Windows 10**: per poter scrivere un'Applicazione per Windows 10 IoT Core (o UWP App in generale), è necessario farlo utilizzando Visual Studio 2015 su Sistema Operativo Windows 10. Al momento è una limitazione, poiché con sistemi più recenti (quali Windows 7/8/8.1, ovvero una grande fetta di utenza) ciò non è possibile, e questo impone obbligatoriamente di fare un Upgrade del sistema a chiunque desideri sviluppare per IoT attraverso Windows.

5.3 Considerazioni Finali

Alla luce dell'esperienza effettuata, attraverso lo studio e l'utilizzo di Windows IoT Core, sono di seguito esposte delle considerazioni finali personali.

É innegabile che la tecnologia, così com'è allo stato attuale, risulti decisamente comoda in fase di sviluppo. Non ci si riferisce soltanto ad applicazioni IoT, ma anche l'implementazione di sistemi embedded autonomi (ad esempio operanti su Raspberry), una volta installato il sistema operativo, risulta più semplice e veloce (merito sicuramente da attribuire, oltre che alla piattaforma, anche al debugger remoto di visual studio) rispetto ad implementazioni alternative (quali programmi Java da eseguire su Sistema Operativo Raspbian).

Tornando a Internet of Things, è sicuramente una valida piattaforma (tenendo conto di pro e contro descritti in precedenza) da considerare per lo sviluppo di sistemi, alla pari delle tecnologie precedenti (e ancora in uso) per quanto riguarda la capacità di progettare in maniera "pulita" i sistemi e la complessità di sviluppo (spesso addirittura inferiore). Bisogna però soffermarsi bene sui requisiti di un dominio/caso specifico prima di scegliere Windows IoT Core, in quanto questa tecnologia al momento risulta ancora **limitata** (per i motivi elencati nelle sezioni precedenti) rispetto alle altre e potrebbe non coprire tutte le esigenze di un dato progetto.

Bibliografia

[1] Learning the Internet of Things, Peter Waher, Packt Publishing Ltd., 2015

[2] Internet of Things - Applications and Challenges in Technology and Standardization, Debasis Bandyopadhyay, Jaydip Sen, 2011

[3] IEEE P2413: https://standards.ieee.org/develop/project/2413.html

[4] MQTT.org: mqtt.org

[5] CoAP tecnology: coap.technology

[6] XMPP Standards Foundation: xmpp.org

[7] Data Distribution Service: https://it.wikipedia.org/wiki/Data_Distribution_Service

[8] AMQP: www.amqp.org

[9] Eclipse IoT: iot.eclipse.org

[10] Kaa Open Source: http://www.kaaproject.org/

[11] SiteWhere, the Open Platform for IoT: http://www.sitewhere.org/

[12] Guide to Universal Windows Platform: https://msdn.microsoft.com/it-it/library/windows/apps/dn894631.aspx

[13] Headed and Headless mode: http://ms-iot.github.io/content/en-US/win10/HeadlessMode.htm

[14] Windows Remote Arduino: https://github.com/ms-iot/remote-wiring/blob/develop/README.md

[15] Firmata Protocol Documentation: https://github.com/firmata/protocol

[16] Firmata Protocol for Arduino: https://github.com/firmata/arduino

[17] Architecture of Windows NT: https://en.wikipedia.org/wiki/Architecture_of_Windows_NT