qCloud Manager

Progetto di Programmazione ad Oggetti

Mariano Sciacco (1142498)

2018 — 2019

Corso di Laurea in Informatica, Scuola di Scienze, Dipartimento di Matematica, UniPD

Indice

1 Introduzione al progetto

1.1 Abstract

Si vuole realizzare un programma per la gestione dell'infrastruttura hardware di una azienda denominata **qCloud Solutions** che offre servizi Web. L'azienda possiede una **server farm** in cui vengono erogati dei servizi che si differenziano per varie tipologie in base alle necessità dei clienti. Ciascun server ha un costo base fissato e un costo di risorse che varia in base alla configurazione.
I server messi a disposizioni si differenziano per tre tipologie principali:

Hosting: soluzione mirata per uno o più siti web e completamente gestita lato server da parte dell'azienda. Le risorse richieste sono molto basse.

VPS (virtual private server): soluzione intermedia che permette di avere un server con risorse scalabili ad un costo contenuto. Usato su larga scala e solo su ambienti virtualizzati.

Server Dedicati: soluzione molto onerosa che viene impiegata soprattutto per grossi progetti che richiedono molte risorse.

Ogni servizio erogato fa parte di un nodo hardware. Un *nodo hardware* costituisce una serie di macchine fisiche che si differenziano per la presenza di *virtualizzazione* del sistema operativo e per il quantitativo totale di risorse massime.

Per agevolare la gestione ed efficienza dei servizi, l'azienda ha adottato un sistema a punteggi per ogni singolo servizio erogato in relazione al nodo di appartenenza: un server può *pesare* da 1 a 4 punti risorse. Quando un nodo hardware supera il numero di punti risorse disponibili, viene segnalato tramite *warning*, così come quando non viene minimamente usufruito. Sfruttare un nodo più del suo potenziale è permesso, ma ciò può causare troppo carico sulle macchine fisiche e provocare downtime.

Infine, ciascun nodo hardware ha un *consumo variabile di KW/H* basato sulla quantità di macchine attive presenti. Il tutto si differenzia sia a carico minimo (*idle*) che a pieno carico (*full*). Tali statistiche permettono di fare previsioni sul costo di mantenimento dei servizi in relazione al guadagno.

1.2 Funzionalità del programma

Il programma permette di gestire i nuovi server erogati e di assegnarli a nodi hardware predefiniti. In particolare, sono presenti le seguenti funzionalità:

- Caricamento, salvataggio ed esportazione dei dati correnti
- Aggiunta di un nuovo servizio (server)
- Modifica di un servizio correntemente attivo
- Eliminazione di un singolo servizio
- Eliminazione multipla basata su una ricerca specifica
- Ricerca specifica per nodo, tipo, etichetta e indirizzo IP del servizio
- Visualizzazione veloce delle caratteristiche di un server tramite tabella
- Visualizzazione dei nodi hardware e della relativa percentuale di completamento
- Visualizzazione degli avvisi (*warning*) dei nodi
- Visualizzazione delle statistiche globali in tempo reale in base alla configurazione corrente

1.3 Premesse: dall'idea al progetto

L'idea di fondo del programma si basa sul fatto che l'applicazione permetta di gestire tutto il servizio aziendale principale attraverso una infrastruttura di rete già predisposta, che ha le seguenti premesse:

- l'infrastruttura ricava in automatico i nodi e i server presenti con le relative specifiche e genera dei file XML appositi;
- l'aggiornamento di un server in termini di scalabilità è fatto con subroutine interne automatizzate;
- l'accesso è protetto e l'usufrutto è dedicato solo ai dipendenti aziendali.

Da queste premesse si è costruito e adattato il progetto affinché si potesse ottenere il risultato più vicino a quello pensato. A livello implementativo si è scelto di lasciare in *sola lettura* il file XML che contiene i *nodi*, mentre i server possono essere caricati con un file esterno, di cui fa uso la parte di gestione file.

2 Breve manuale d'uso

A prima apertura del programma, viene richiesta la creazione (*File, Nuovo*) o il caricamento (*File, Apri un nuovo file*) di un file XML che contiene le configurazioni dei servizi erogati. In alto a destra, troviamo i **contatori** di avvisi, server e nodi, aggiornati in tempo reale. Un pochino più sotto al logo è presente una **sezione informativa** che spiega brevemente le funzionalità del tab corrente ed eventuali Hot-Keys. In basso a destra, invece, è presente una **barra di gestione file** che emette notifiche circa la lettura e la scrittura del file.

2.1 Dashboard

La dashboard contiene gli avvisi che riguardano i nodi e le statistiche. Per quanto riguarda gli **avvisi**, è opportuno tenere sotto controllo i *Warnings* al fine di aumentare l'efficienza dei singoli nodi. Le **statistiche**, invece, sono aggiornate in tempo reale e permettono di visualizzare i guadagni e il costo di mantenimento.

2.2 Gestione servizi

Nella gestione servizi è possibile eseguire azioni di aggiunta, modifica e cancellazione dei server.

- Cliccando su un singolo server è possibile aprire i **dettagli** che compariranno sulla destra, a fianco della tabella.
- Con un dettaglio aperto, è possibile eseguirne la **modifica** e **cancellazione** con i relativi bottoni che compariranno.
- Per **chiudere un dettaglio** è sufficiente cliccare sul bottone *Chiudi* in alto a destra così da poter riprendere anche la ricerca.
- Per **aggiungere un server** si deve cliccare sull'apposito bottone *Aggiungi* posto in alto a sinistra sulla tabella; ad esso seguirà un form laterale con i relativi campi da compilare.
- Per **ricercare** dei servizi specifici è sufficiente riempire la barra di ricerca e/o selezionare un nodo dal menù a tendina posti sopra la tabella. Dai risultati ottenuti è possibile eseguire la **cancellazione multipla** cliccando sul bottone in basso a sinistra rispetto alla tabella.

2.3 Gestione nodi

La gestione dei nodi permette solamente di **visualizzare** in tempo reale i dettagli di un nodo, tra cui il quantitativo di risorse impiegate e la percentuale di completamento. *Per vedere più dettagli di un nodo* è sufficiente cliccare su una entry della tabella e, successivamente, compariranno sulla destra i dettagli, come per la gestione servizi.

2.4 Opzioni file

Il file che contiene i dati può essere salvato a seguito di modifiche, salvato con nuovo nome o esportato. Il **salvataggio con nome** comporta la creazione e la futura modifica di un nuovo file con i server correntemente modificati. **L'esportazione**, invece, crea semplicemente un nuovo file a parte, mantenendo le modifiche sul file correntemente aperto. Per eseguire queste operazioni si può usare la barra superiore del menù *File* o le **hotkeys** segnalate nel menù. Infine, per avere maggiori dettagli sul file correntemente aperto, basta cliccare sul menù *Informazioni*, *File (corrente)*.

2.5 Listino prezzi, punti risorse e consumi

Tabella 1: Listino prezzi fisso

Categoria	**Oggetto**	**Prezzo mensile unitario**
Generale	vCore (1)	2.5 €
Generale	Ram (0.5GB)	2.0 €
Generale	Disco (10GB)	1.5 €
VPS	Base	3,5 €
VPS	Snapshot	2,0 €
VPS	AntiDDoS	10,0 €
Hosting	Base	7,5 €
Hosting	Accesso SSH	6.0 €
Hosting	Turbo boost	5.0 €
Dedicato	Base	35,0 €
Dedicato	vCore (1)	3,5 €
Dedicato	Ram (1GB)	4,5 €
Dedicato	Disco (240GB)	10,0 €
Dedicato	Backup attivo	15,0 €
Dedicato	Ip aggiuntivo (1)	5,0 €

Tabella 2: Punti risorse dei server usati dai nodi

Server	**Punti risorse**	**Nota**
Hosting	1	-
VPS	2-4	Varia in base ai vCores
Dedicato	4	-

Tabella 3: Velocità CPU dei server

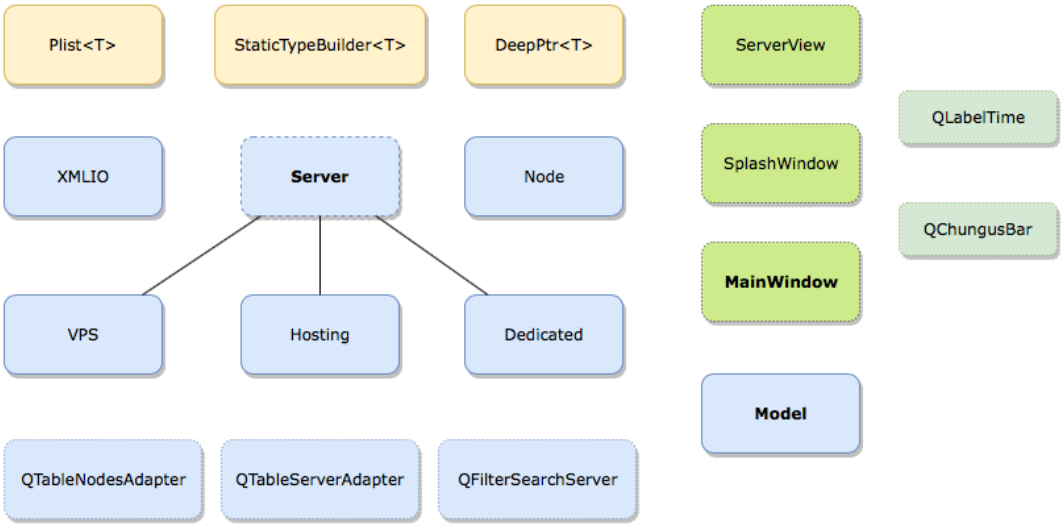
Server	**Velocità**	**Nota**
Hosting	1.2 Ghz	+25% con Turbo Boost
VPS	2.1 Ghz	-5% con AntiDDoS
Dedicato	3.4 Ghz	-

Tabella 4: Consumi ipotetici dei Nodi

Base approssimativa	Stato	Costo
250 W / 4 punti risorsa	Idle (25%)	0.089 KW/h
250 W / 4 punti risorsa	Full (100%)	0.089 KW/h

3 Progettazione

L'approccio di sviluppo per il progetto si è basato sul pattern **Model-View** di *Qt*. Oltre alla gerarchia e al gestore XML input/output, sono stati realizzati degli adattatori e un filtro di ricerca che comunicano alla view i cambiamenti del modello. In questa sezione verranno analizzati gli elementi principali del modello e della view, facendo chiarezza sulle opportune dipendenze.



3.1 Modello e Gerarchia

3.1.1 Gerarchia principale

La gerarchia principale del modello si costituisce di una *base astratta* `Server` da cui derivano direttamente 3 classi concrete: `VPS`, `Dedicated` e `Hosting`.

Ciascuna classe implementa dei metodi virtuali puri che riguardano la **serializzazione e deserializzazione dei dati**, **i limiti di cores, RAM e disco**, **la clonazione profonda**, **il calcolo della velocità della CPU**, **l'obbligo di virtualizzazione**, **il peso delle risorse** e **il calcolo del prezzo**.

3.1.2 La classe Node

La classe `Node` implementa un tipo molto basilare che viene utilizzato nel modello e che caratterizza un singolo nodo hardware di cui fanno parte dei server. Questa classe è pensata per rimanere indipendente, sebbene faccia uso della stessa deserializzazione della gerarchia e sia compatibile con il `DeepPtr`.

3.1.3 Il template ausiliario della gerarchia

Al fine di semplificare ed aumentare l'estendibilità della gerarchia si è optato di tenere traccia dei tipi delle classi concrete attraverso l'implementazione di una **mappa statica**. Essa contiene come **chiave** una stringa che è il nome della classe e come **valore** un oggetto dello stesso tipo.

Inoltre, la mappa viene utilizzata per la parte di serializzazione e deserializzazione dei dati e utilizza il polimorfismo per richiamare i metodi virtuali. La costruzione della mappa è la seguente:

- La classe base `server` ha una mappa statica come membro protetto.
- Ciascuna classe derivata ha un campo dati statico di tipo `StaticTypeBuilder<T>`, dove `T` identifica il nome della classe da cui viene richiamato.
- All'avvio del programma, vengono costruiti i campi dati statici e la mappa viene popolata per mezzo del template che esegue esattamente questa funzione ausiliaria per la gerarchia.

In questo modo, per estendere ulteriormente la gerarchia, garantendo il funzionamento autonomo della gestione dati, è sufficiente *dichiarare il campo dati statico e istanziarlo* con il costruttore di default, agevolando così la *conoscenza* da parte del builder di un nuovo tipo concreto. Nel caso in cui la nuova classe fosse astratta (e derivata dalla base) si potrà omettere semplicemente tale implementazione. Ovviamente i relativi metodi virtuali puri alla base dovranno essere implementati per garantirne pieno funzionamento.

3.1.4 Il container e il deep pointer

Il container realizzato per questo progetto è una lista concatenata singolarmente con le principali funzioni di *inserimento, rimozione, ricerca, swap, clonazione profonda e distruzione profonda.* Viene fatto uso di un contatore degli oggetti all'interno della lista e anche di due puntatori all'inizio e alla fine della lista. Il container, inoltre, fa uso di un iteratore costante, mentre per l'iterazione non costante viene usato direttamente l'operatore di subscripting (`operator[]`) e la funzione `Select(int i)`, in accoppiata al metodo ausiliario privato `Jump(unsigned int)`.

In aiuto alla gerarchia, infine, si fa uso del **Deep Pointer** implementato tramite template in maniera indipendente che semplifica le operazioni di eliminazione e di clonazione degli oggetti della gerarchia inseriti all'interno del container. In aggiunta, il `DeepPtr<T>` è compatibile anche con la classe `Model` e `Node` per garantire distruzione profonda degli oggetti, dove richiesto.

3.1.5 La classe Model

Il modello (`Model`) si compone di tutte le classi precedentemente menzionate e principalmente fa uso di due container di oggetti `DeepPtr<Node>` e `DeepPtr<Server>`. Sulla base di questi contenitori, vengono create anche delle **mappe ausiliarie** basate sui nodi che sono usate per ricavare *dati statistici e avvisi.*

Il modello si occupa anche della gestione dei dati tramite file XML, controllandone le funzioni di creazione, salvataggio, caricamento ed esportazione, di cui verrà menzionato meglio il funzionamento in seguito.
Questa classe, inoltre, si occupa anche di **aggiungere ed eliminare oggetti** nei contenitori principali, gestendo gli esiti di ciascuna operazione.

3.1.6 Adattatori delle tabelle e filtri di ricerca

Al fine di far comunicare direttamente il *modello* con la *view*, sono stati creati due *adattatori* derivati dalla classe `QAbstractTableModel` e un *filtro* utilizzato per la ricerca dei server che si interpone tra l'adattatore e la view della tabella dei server, con classe derivata da `QSortFilterProxyModel`.

Entrambi gli adattatori possiedono un *puntatore semplice al modello* che fa uso dei suoi metodi in base alle necessità. Le funzioni implementate per le tabelle permettono la visualizzazione, l'aggiunta e l'eliminazione di righe nella tabella. Al fine di semplificare l'usabilità, la modifica delle entry è disponibile solamente tramite l'apposito modulo *Modifica* che compare a destra a fianco della tabella e non tramite modifica alle singole celle.

I filtri di ricerca permettono la ricerca testuale su 4 campi distinti del tipo `Server` e, separatamente tramite menù a tendina, sul nodo di appartenenza. In questo modo è possibile visualizzare tutti i server di un singolo nodo ed eseguire delle ricerche testuali più approfondite.

3.2 Graphical User Interface (GUI)

3.2.1 La splash window

Eseguendo il programma, viene visualizzata una *splash window* durante il caricamento delle impostazioni, ossia una piccola finestra a tempo che mostra il logo del programma e le informazioni di caricamento. La classe `SplashWindow` deriva direttamente da `QSplashScreen` e inizializza il banner insieme allo stylesheet (CSS) dell'applicazione. La classe prende per riferimento un oggetto `MainWindow` che viene poi *visualizzato* alla fine del caricamento nel metodo `execute`.

3.2.2 La finestra principale

La finestra principale (`MainWindow`) è una classe derivante da `QWidget` ed è la classe che gestisce tutta la schermata principale della GUI, i cui membri privati sono principalmente puntatori a oggetti di tipo `QWidget`, `QLayout` e, ovviamente, un puntatore profondo al modello. Questa implementazione permette l'aggiornamento degli elementi visualizzati solo strettamente necessari e stabilisce una comunicazione con il modello per le principali richieste di informazioni o di aggiornamento dei due contenitori.
Nota particolare è il metodo statico `cleanLayout(QLayout *)` che permette la cancellazione ricorsiva dei widget e dei layout figli, garantendone la cancellazione prima di un eventuale refresh.

La finestra principale, infine, fa uso degli oggetti di due classi ausiliarie (`QLabelTime` e `QChungusBar`) per la generazione di un label con l'orario attuale (usato per la gestione file in basso a destra) e per la gestione dinamica dei colori nelle barre di completamento.

3.2.3 La server view

La classe `ServerView` è una classe ausiliaria a `MainWindow` che permette la creazione delle pagine di visualizzazione, aggiunta e modifica della parte dei servizi. Questa view comunica direttamente col `Model` per aggiornare i dati e con `MainWindow` per notificare modifiche ai dati. All'interno di questa classe ci sono anche dei metodi di verifica e controllo dell'input inserito nei form, che vanno a segnalare anche eventuali valori duplicati (come l'IP e l'etichetta).

3.3 Polimorfismo

L'uso del polimorfismo all'interno del progetto è fortemente presente nei contenitori per mezzo degli oggetti `DeepPtr<Server>` della gerarchia. In diversi punti del programma, specialmente nella GUI, si utilizzano **chiamate polimorfe** a metodi virtuali, che permettono il recupero di determinate informazioni, come ad esempio il calcolo del prezzo o la velocità della CPU in relazione ai membri privati usati.

Analogamente, viene fatto largo uso del polimorfismo anche nella parte di *creazione degli oggetti* al caricamento dei file XML, in cui si gestisce attraverso una chiamata di deserializzazione l'inserimento nel contenitore degli oggetti appena creati.

A favore di queste implementazioni, è stato reso necessario il riconoscimento del tipo di classe usando un metodo virtuale denominato `staticType()` che fa uso del polimorfismo per ritornare a stringa il tipo corrente, così da eliminare l'esigenza del *type checking dinamico* e permettendo il *cast statico*degli oggetti nelle operazioni più specifiche.

3.4 Gestione dati

Il programma fa uso di **XML** ai fini di salvataggio e caricamento dati. I dati vengono gestiti ad *alto livello* dal modello che esegue caricamento e salvataggio delle modifiche effettuate sulla istanza correntemente aperta e comunica visivamente all'utente i singoli cambiamenti per mezzo del box messaggi posto in basso a destra nell'applicazione.

A *basso livello*, invece, è stata implementata una classe denominata `XMLIO` che esegue la deserializzazione e la serializzazione per server e nodi. Questa classe fa uso dei **metodi virtuali** appositamente realizzati nella gerarchia (`dataUnserialize`, `dataSerialize`, `createObjectFromData()`) al fine di poter leggere o scrivere i file XML in maniera autonoma e distinta per ogni singola classe concreta, così da evitarne la necessità di estensione o modifica nel momento in cui si va ad estendere la gerarchia.

3.5 Note generali sulle implementazioni

- Si è scelto di usare un contenitore di tipo **lista** in quanto il più adeguato da usufruire per questo ambito, prevedendo di avere un numero contenuto di elementi. Da questa motivazione, inoltre, si è scelto di risparmiare byte anche per quanto riguarda il tipo di alcuni elementi, preferendo `short` a `int` in molti casi.
- Il template `StaticTypeBuilder<T>` di fondo andrebbe contro il principi *SOLID* dal momento che viene **esplicitata la costruzione della mappa dei tipi**. In realtà, tale soluzione è stata ritenuta la meno onerosa per implementazioni future dal momento che, qualora si volesse estendere la gerarchia, sarebbe necessario menzionare nella documentazione l'obbligo di creare una sottoclasse interna (protetta o privata) da usare per questo scopo. Omettere completamente questa funzionalità, d'altra parte, richiederebbe come conseguenza l'aggiornamento del XMLIO e del modello, il che va a sfavore dell'estendibilità. Altre alternative sarebbero state probabilmente più adeguate (es: avere una classe che fa esattamente da listener/builder e genera la mappa), ma per questioni di tempo richiesto dal progetto e per poca conoscenza sull'argomento si è preferito dedicarsi ad altre funzionalità.
- Per quanto riguarda l'utilizzo del **Model-View**, è opportuno ammettere che a programma avviato l'istanza degli adattatori e del modello rimangono sempre le stesse, pertanto la cancellazione del modello verrà fatta con la distruzione della classe `MainWindow`. Ciò è stato rafforzato ulteriormente con l'ausilio del `DeepPtr`, impiegato appositamente anche per il modello. Infine, ciascun adattatore fa uso di un puntatore normale al modello in base alle informazioni che deve ritornare e, allo stesso modo, ad ogni distruzione di queste classi si distrugge solamente il puntatore e non l'oggetto puntato, come corretto che sia.
- La **GUI** del programma si è basata in modo particolare sull'usabilità, tale per cui fosse semplice, sufficientemente intuitiva e al contempo informativa. Per questo motivo, sono state aggiunte icone colorate, tooltips, hotkeys e messaggi informativi dove è stato possibile, senza peccare sulle funzioni essenziali richieste dal progetto. L'applicazione, dunque, parte da una base minima, ma solida e aperta a future funzionalità, nei limiti del possibile.
- Nel **Modello** e nella **Gerarchia**, dal momento che si fa comunque uso della libreria di Qt per la gestione del XML, è stato preferito l'utilizzo di `QString` rispetto a `string`. Questo ha reso l'implementazione finale più leggera e con meno conversioni di tipo esplicite, senza intaccare sulla separazione modello / vista.

4 Conclusioni

4.1 Timeline e riassunto ore di lavoro

Il progetto è stato inizialmente realizzato con una gerarchia differente verso dicembre 2018 - gennaio 2019. A marzo il modello è stato riprogettato, ne sono state eseguite le relative reimplementazioni ad aprile e si è concluso del tutto a maggio.

A livello di *ore lavorative*, si può dire che ci sono state circa un **40% di ore lavorative in più** rispetto alle **50 ore** previste dalle specifiche, dovute alla riprogettazione iniziale e ad un maggiore affinamento della GUI a livello grafico.

Fase del progetto	**Ore lavorative**
Analisi del problema	3
Progettazione modello e GUI	9
Apprendimento libreria Qt	8
Codifica e implementazione modello	14
Codifica e implementazione GUI	20
Test generali e Debugging	10
Stesura Relazione in LaTeX	5
Ore totali	**69 ore**

4.2 Ambiente di lavoro

Per sviluppare si è fatto uso di **Qt creator** nella versione `5.9.5`, usando come compilatore `clang 8.1.0`. Lo sviluppo è avvenuto principalmente su **MacOs 10.13 "High Sierra"** (3/4 del tempo) e **Windows 10** (1/4 del tempo).

A tal proposito è stato integrato un processo di versionamento e building per mezzo di **Github** e **Travis CI** tale per cui ad ogni singola modifica è stata fatta una compilazione automatica via cloud su una macchina **Linux**. Con questa configurazione si è potuto garantire il funzionamento del programma anche su **Ubuntu 14.04 "Trusty"** con compilatore `g++ 4.8.4`

4.3 Compilazione del progetto ed esecuzione

Il progetto prevede la compilazione tramite file `.pro` dal momento che contiene la posizione di tutti i file e identifica la versione specifica usata per C++, ossia C++11, di cui se ne fa uso nel modello con keywords apposite.

```
$ qmake qCloudManager.pro
$ make
```

A primo avvio, è possibile creare un nuovo file dal menù oppure può essere caricato un file dati compatibile. Nella cartella del progetto, sono stati messi dei **file XML** di esempio che possono essere usati per il primo caricamento. Alternativamente si può creare un nuovo file con dentro `<serverList/>` e selezionare il file appena creato.

4.4 Note conclusive dello sviluppatore

In linea di massima, il progetto ha richiesto un impegno moderato, dal momento che gran parte delle ore sono state spese per la fase di apprendimento delle classi della libreria di Qt. Una volta capito il funzionamento però, l'implementazione è stata abbastanza *straight-forward*.

Una buona porzione di tempo, inoltre, è stata usata nella fase di progettazione e test della GUI così da curarne l'usabilità, l'aspetto grafico e le funzionalità. Questo ovviamente non ha tolto ore di lavoro alla gerarchia, in cui si è cercato di garantire massima estendibiità. In modo analogo, come richiesto dalle specifiche, è stata curata molto la parte di *interazione* tra modello e view, al fine di ottenere la massima indipendenza senza trascurarne le funzionalità.

Concludendo, si può affermare che il progetto è stato abbastanza lungo, ma il processo di sviluppo è stato preso con molta tranquillità e con tutti i possibili accorgimenti al fine di ottenere un prodotto adeguatamente funzionante e stabile.