P2P-DiscoveryService-WCF

Vincenzo Morgante

Indice

1 Introduzione

Il presente progetto ha avuto come obiettivo la realizzazione di un sistema distribuito in grado di permettere la *ricerca* e l'uso di risorse di calcolo utilizzabili da remoto. Trattandosi di un progetto didattico, in particolare pensato per approfondire WCF[1], sono stati stabiliti i seguenti vincoli per lo sviluppo:

- il peer-to-peer come infrastruttura di comunicazione tra i nodi, ai fini della ricerca di risorse; tale ricerca non deve essere predeterminata o basata su un elenco disponibile a tutti i nodi;
- utilizzo di WCF quale soluzione implementativa per la comunicazione tra i nodi della rete P2P, ma escludendo in ogni caso le soluzioni preconfezionate messe a disposizione dal .NET stesso o da eventuali librerie di terze parti;
- realizzazione di un'app per Windows Phone che interagisca con l'applicazione distribuita;
- eventuale uso del pattern MVVM[2] per disaccoppiare la/le UI dal codice che implementa la business logic; per quanto riguarda il design della/le UI, si tenga presente che non sono stati definiti requisiti di usabilità ai fini del presente progetto.

Il software sviluppato comprende tre applicazioni:

- il server di elaborazione, che rappresenta una risorsa di calcolo per l'elaborazione remota dei dati ed è identificato univocamente dal proprio URI;
- il generico nodo della rete P2P, che monitora uno o più server di elaborazione e consente di effettuare la ricerca delle risorse di calcolo;
- il client con il quale è possibile, prima, avviare la ricerca di una risorsa di calcolo tramite un nodo di ricerca, dopo, richiedere l'elaborazione remota dei dati su uno dei server di calcolo trovati.

1 WCF, acronimo per Windows Communication Foundation, è un modello di programmazione utato per lo sviluppo di applicazioni orientate ai servizi.

2 MVVM, acronimo per *Model-View-ViewModel*, è un modello per la progettazione di applicazioni, grazie al quale è possibile separare il codice dell'interfaccia grafica dal resto del codice.

2 Sistema proposto

Il sistema proposto nel presente progetto è stato ideato come strumento in grado di gestire la ricerca e l'uso delle risorse di calcolo remote per portare a termine diverse tipologie di elaborazione. Data la complessità e i tempi di sviluppo richiesti per realizzare un sistema di elaborazione distribuita, per il presente progetto sono state introdotte alcune semplificazioni, tenendo conto dei vincoli stabiliti.

Per ottenere una maggiore flessibilità nell'uso del sistema e per ridurne la complessità di sviluppo, le funzionalità di elaborazione remota e di ricerca delle risorse di calcolo sono state implementate in due distinte applicazioni, ovvero il server di elaborazione è stato realizzato in un eseguibile separato rispetto al nodo di ricerca. In ogni caso, nulla vieta di mettere in esecuzione entrambi gli eseguibili sullo stesso sistema.

Per comunicare tra loro, le applicazioni realizzate adoperano i cosiddetti *web service*, implementati tramite WCF. Nel presente documento, si userà il termine *servizio* per indicare un web service.

Il sistema è destinato a tre categorie di utenti: i fornitori delle risorse di calcolo, i gestori dei nodi di ricerca e i fruitori delle risorse di calcolo; in quel che segue, per fare riferimento a questi ultimi si userà il termine *utente/i*, lasciando intendere che si tratta degli utenti del sistema distribuito e quindi di coloro che sfruttano le risorse di calcolo.

I fornitori delle risorse di calcolo hanno il compito di installare, configurare e mettere in esecuzione i server di calcolo su risorse hardware da loro amministrate. Ogni server ospita un web service, che permette di interagire con esso: per conoscere le risorse di calcolo presenti, per inviargli un *task* da elaborare, per verificare lo stato di elaborazione di un task e per effettuare il download dei risultati ottenuti al termine dell'elaborazione di un task.

Poiché potrebbe essere disponibile un gran numero di server di calcolo e poiché ogni singolo server potrebbe fornire risorse di calcolo differenti, gli utenti sfruttano i cosiddetti *nodi di ricerca* per poter conoscere quali server di calcolo forniscono una determinata risorsa. Ogni nodo di ricerca si occupa di monitorare uno o più server di calcolo per verificare se sono in linea e quali risorse mettono a disposizione, mantenendo così un elenco delle “proprie” risorse, cioè delle risorse disponibili sui server da esso monitorati. I vari nodi di ricerca formano anche una rete P2P in base agli accordi tra i relativi gestori: i responsabili dei nodi di ricerca decidono quali tra questi *peer* possono comunicare tra loro e li configurano di conseguenza, in particolare l'amministratore di un nodo di ricerca ha il compito di redigere tre file di testo, di cui il primo contenente gli URI dei servizi di elaborazione remota da monitorare, il secondo contenente una stringa che lo identifica univocamente, il terzo contenente la lista degli identificatori dei propri vicini nodi di ricerca con i relativi URI. In figura 1 è mostrato un esempio di struttura del sistema distribuito proposto.

Un utente, mediante l'applicazione client installata sul proprio dispositivo, interroga uno dei nodi di ricerca (ciò avviene tramite un web service) per trovare eventuali server di calcolo che dispongono della risorsa di elaborazione cercata, quindi il nodo interrogato mette in atto le seguenti azioni:

- verifica se le “proprie” risorse soddisfano la richiesta dell'utente ed in caso positivo, invia al client una risposta contenente l'elenco degli URI che forniscono la risorsa cercata;
- inoltra la richiesta dell'utente ai propri vicini; in tal modo, grazie al meccanismo di flooding implementato nei nodi di ricerca, essa si diffonde nella rete P2P, in attesa che qualche altro nodo possa rispondere alla richiesta; le eventuali risposte vengono inviate lungo il percorso seguito dalle relative richieste.

Quando la ricerca si conclude, se il client ha ricevuto almeno un URI che può soddisfare la richiesta iniziale, l'utente potrà contattarne direttamente uno per richiedere l'elaborazione del task.

Nella release 7.1 di Windows Phone non è possibile sfruttare i socket, ma soltanto chiamate HTTP. Coerentemente con questa limitazione, un'applicazione sviluppata per questo sistema operativo è in grado di consumare soltanto servizi WCF che impiegano un binding di tipo *BasicHttpBinding*.

Per garantire la compatibilità con l'applicazione client sviluppata per Windows Phone, il servizio di ricerca ospitato sul nodo di ricerca e quello di elaborazione ospitato sul server di calcolo sono stati progettati e configurati per usare il suddetto tipo di binding. Inoltre, si è pensato di sviluppare anche una versione di tale applicazione che fosse compatibile con i sistemi desktop Windows, sfruttando la possibilità di poter utilizzare quasi tutto lo stesso codice di base che implementa la business logic nell'applicazione per Windows Phone.

Infine, per assicurare l'interoperabilità con applicazioni future che potrebbero essere realizzate in ambienti differenti dal .NET, anche la comunicazione tra i nodi di ricerca è stata implementata con un servizio WCF che usa il binding *BasicHttpBinding*.

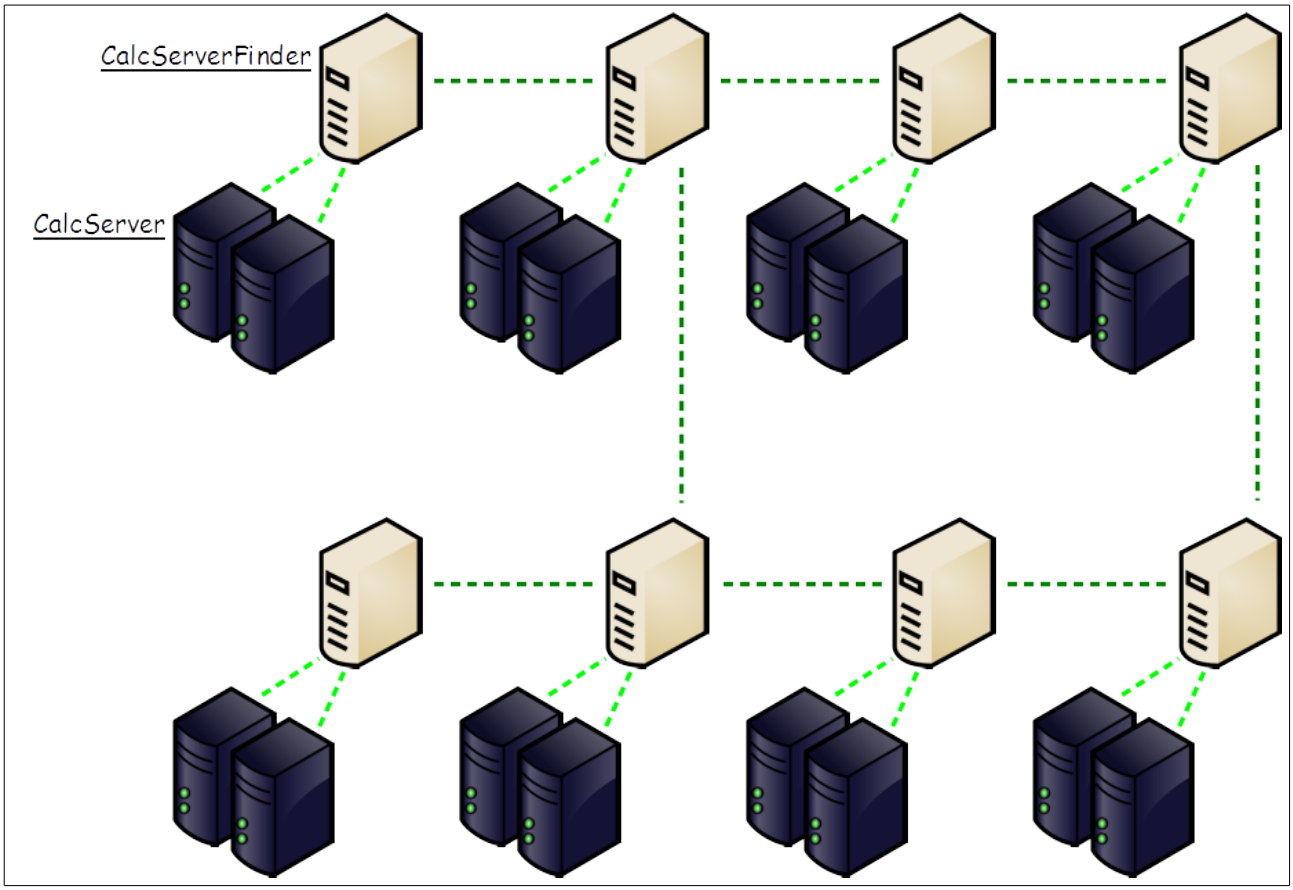

Figura 1. Esempio di struttura del sistema distribuito proposto: in questo esempio sono stati rappresentati 8 nodi di ricerca, ognuno dei quali monitora 2 server di calcolo. Le connessioni tra i peer sono state evidenziate in verde scuro, mentre quelle con i server di calcolo sono in verde chiaro.

3 Strumenti utilizzati

Gli strumenti utilizzati per lo sviluppo del progetto sono i seguenti:

- C# .NET 4.0;
- Visual Studio 2010;
- Windows Phone SDK 7.1.1.

Il software realizzato, per funzionare, necessita dell'ambiente di esecuzione .NET Framework 4.0.

4 Applicazione P2P per la ricerca dei servizi di calcolo

La comunicazione tra i nodi che effettuano la ricerca dei servizi di elaborazione trae ispirazione dal ben noto protocollo Gnutella [1], ma presenta una differenza sostanziale: mentre nelle reti Gnutella i messaggi vengono inviati direttamente su socket TCP, nel presente progetto vengono impiegati i web service sviluppati con WCF, come stabilito dai vincoli definiti nel capitolo 1, pertanto l'unica analogia con Gnutella è il meccanismo adoperato per lo scambio di richieste e di risposte tra i nodi di ricerca.

La ricerca è di tipo BFS (breadth-first search), ossia una ricerca in ampiezza: quando un nodo riceve una richiesta, la inoltra anche ai suoi vicini (cioè quei nodi con cui è direttamente connesso), e così essa si diffonde nella rete. Logicamente non è realistico esplorare "tutta la rete", poiché il numero dei messaggi potrebbe divenire eccessivo, pertanto il meccanismo di ricerca si basa essenzialmente su un flooding limitato: regolando in questo modo la profondità di ricerca, l'individuazione della risorsa cercata non è garantita, però si evita di sovraccaricare la rete.

Per quanto riguarda la condivisione delle risorse, il meccanismo sviluppato prevede che un nodo, dopo aver trovato le risorse (gli URI dei servizi di elaborazione remota) attraverso gli altri nodi, ne renda disponibile l'elenco al client ha avviato la ricerca, che potrà poi interagire direttamente con esse senza usare l'overlay network, che perciò ha soltanto funzione di ricerca. Lo schema in figura 2 riassume il funzionamento del nodo.

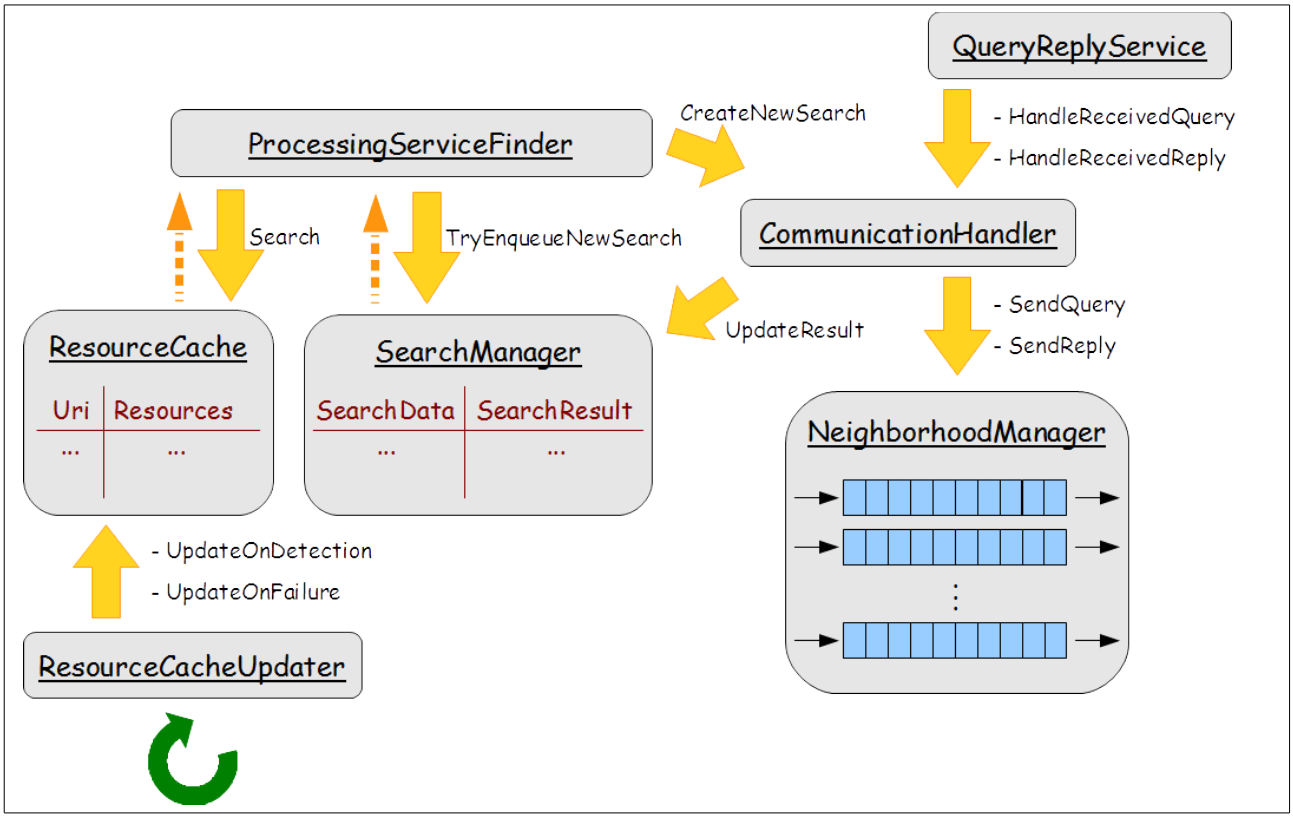

Figura 2. Funzionamento di un nodo di ricerca.

In quel che segue, si farà riferimento ad un generico nodo di ricerca, oltre che come nodo, anche come peer, ed in senso lato questo indicherà anche l'applicazione distribuita realizzata e oggetto di questo capitolo.

4.1 Gestione della rete e dei messaggi in arrivo e in uscita

In uno scenario peer-to-peer basato sui web service, ogni nodo espone un servizio a tutti i potenziali peer e al tempo stesso utilizza un servizio uguale, offerto però da qualche altro nodo della rete. Per accedere alla rete di ricerca, bisogna conoscere uno o più URI su cui sono in ascolto altrettanti peer con cui stabilire una connessione.

Ogni nodo possiede un proprio identificatore, costituito da una stringa che lo identifica in maniera univoca nella overlay network; l'assegnazione dell'identificatore esula dal presente progetto, quindi si può assumere che gli amministratori dei nodi di ricerca abbiano in qualche modo assegnato degli identificatori unici ai nodi di cui sono responsabili (per esempio, l'operazione di generazione e di assegnazione dell'identificatore potrebbe essere centralizzata e gestita con la registrazione presso un servizio che fornisce un identificativo unico ad ogni nodo).

Considerando il sistema proposto nel capitolo 2, gli URI dei vicini non vengono acquisiti tramite la rete, ma l'applicazione richiede che vengano specificati in un file di testo[3] predisposto per lo scopo; ogni rigo di tale file serve per l'immissione delle informazioni (separate da virgola) relative ad un vicino, cioè il suo *identificatore univoco* e l'URI su cui è in ascolto per la ricezione dei messaggi dai vicini: si ipotizza che l'amministratore responsabile della configurazione di ogni nodo abbia fornito questa coppia di informazioni agli amministratori dei nodi scelti come vicini, così che tutti i nodi possano essere configurati in maniera corretta.

Il web service dedicato alla ricezione dei messaggi provenienti dai vicini è stato implementato nella classe `QueryReplyService`: essa contiene i due metodi `Query` e `Reply` specificati nel contratto di servizio definito nell'interfaccia `IQueryReplyService`, nei quali occorre specificare non solo il messaggio, ma anche l'identificatore del nodo che lo sta inviando, essenziale per le operazioni di routing dei messaggi di query e di reply. Per ogni messaggio ricevuto da un vicino, questi metodi invocano due distinti metodi della classe `CommunicationHandler`, il cui compito è proprio di gestire le operazioni di routing dei messaggi.

Ogni nodo associa a ciascun vicino le seguenti informazioni e strutture dati, utili per poter gestire il traffico in ingresso e in uscita dalle relative connessioni:

- una stringa contenente l'identificatore univoco del vicino;
- una coda per la memorizzazione temporanea dei messaggi in uscita diretti verso il vicino.

Su tale coda di uscita agiscono almeno due thread concorrenti: quello avente il compito di prelevare i messaggi contenuti al suo interno per inviarli al vicino, ed uno o più thread che, dovendo inviare un messaggio al vicino, lo inseriscono nella coda di uscita.

La gestione dei messaggi in uscita verso un vicino, ossia l'accodamento degli stessi e il conseguente invio al vicino, avviene in un'istanza della classe `NeighborClient`, che quindi implementa sia il *proxy* per comunicare col servizio `QueryReplyService` del vicino, sia la coda di uscita. Pertanto, in ogni nodo saranno attivi tanti oggetti `NeighborClient` quanti sono i vicini definiti nell'apposito file di testo.

4.2 Comunicazione tra i peer

Non appena un nodo ha allocato correttamente tutte le strutture dati per la comunicazione, esso può comunicare con gli altri utilizzando gli appositi messaggi. Sono previste due tipologie di messaggi, ognuno con caratteristiche ben precise: la *query* è un messaggio di richiesta e serve per cercare le risorse di calcolo disponibili nella rete, la *reply* è la relativa risposta contenente gli URI dei servizi

3 Le informazioni sui vicini di un nodo vanno specificate all'interno del file "Neighbors.txt", che deve trovarsi nella stessa cartella dell'eseguibile.

che possiedono la risorsa cercata; ciascuno di questi messaggi presenta almeno i seguenti campi di base, essenziali per il routing nella overlay network[4]:

- *MSGID* (*message identifier*), una stringa contenente l'identificatore del messaggio;
- *TTL* (*time-to-live*), un numero intero non negativo e inizializzato con un valore positivo;
- *HOPS*, un numero intero non negativo, inizializzato a zero.

Il campo MSGID contiene l'identificatore univoco di un messaggio all'interno della rete ed il suo ruolo principale è quello di rendere possibile il routing dei messaggi nella overlay network: quando un nodo vuole rispondere ad un messaggio, la risposta deve essere inoltrata a ritroso lungo lo stesso percorso seguito dalla relativa richiesta e per individuarlo si usa proprio l'identificatore. Inoltre esso viene adoperato ogni qualvolta una risposta giunge al mittente della relativa richiesta, in modo da poterle associare tra loro. Infine, un ruolo molto importante assunto da questo identificatore è quello di prevenire la diffusione di eventuali messaggi duplicati, poiché con esso è possibile evitare che un messaggio, già inoltrato da un nodo ai propri vicini, venga da quest'ultimo nuovamente immesso in rete come duplicato e sprecando banda (i dettagli sul routing sono affrontati nel paragrafo 4.2.1).

Per creare un nuovo identificatore, i nodi sfruttano un algoritmo[5] per generare *UUID* (*Universally Unique IDentifier*, [2]): si tratta di un numero pseudo-casuale di 128 bit che, sebbene l'algoritmo di generazione non ne garantisca l'effettiva unicità, il numero di combinazioni ($2^{128} \approx 3.4 \times 10^{38}$) è così elevato da rendere praticamente impossibile una coincidenza tra due identificatori. Un nodo genera un nuovo identificatore per ogni nuovo messaggio di richiesta creato, quindi non per quelli ricevuti che deve inoltrare ad altri nodi.

Il TTL è il numero di volte che il messaggio può essere ancora inoltrato dai nodi e stabilisce il cosiddetto raggio (o profondità) di ricerca, fissando così un limite alla diffusione dei messaggi nella rete. Il campo HOPS contiene il numero di salti, cioè il numero di volte che un messaggio è stato inoltrato da un nodo lungo un certo percorso, e permette di conoscere la distanza dal mittente lungo tale percorso. Ogni volta che un messaggio giunge presso un nodo, quest'ultimo ne aggiorna i campi TTL e HOPS decrementando di 1 il primo e incrementando di 1 il secondo, così da conteggiare l'ultimo link logico attraversato: la *fase di ricezione* è la prima ad essere eseguita all'arrivo di un qualsiasi messaggio, quindi precede sempre ogni possibile altra azione (elaborazione e/o inoltro) su di esso. Di conseguenza, la somma dei campi TTL e HOPS è costantemente uguale al TTL iniziale impostato dal mittente.

$$TTL(0) = TTL(n) + HOPS(n)$$

TTL(0) è il valore iniziale assegnato al TTL, mentre TTL(n) e HOPS(n) sono i valori dei campi rispettivamente TTL e HOPS dopo *n* salti. Poiché non sarebbe molto realistico esplorare tutta la rete, di solito è preferibile limitare il flooding, assegnando un valore adeguato al TTL iniziale[6].

Ogni volta che un nodo riceve un messaggio da un vicino, dopo averne aggiornato TTL e HOPS, lo elabora per mandare a quest'ultimo un'eventuale risposta (se prevista); contemporaneamente, prima di inoltrarlo ai suoi vicini (ad esclusione del vicino di provenienza), verifica se è scaduto (cioè se il suo TTL vale 0): in tal caso il messaggio non viene inoltrato e così se ne limita la diffusione oltre il raggio di ricerca stabilito.

---

4 I messaggi di query e reply sono implementati rispettivamente nella classe `QueryData` e nella classe `ReplyData`: entrambe derivano dalla classe astratta `MessageData`, che include i campi comuni usati per il routing dei messaggi.

5 Per generare un nuovo identificatore, viene adoperato il metodo statico `NewGuid()` della classe `Guid` del .NET: ciò avviene nel metodo `CreateNewSearch()` della classe `CommunicationHandler`, che viene invocato ogni volta che è necessario avviare una nuova ricerca.

6 Nelle simulazioni eseguite col software implementato, il valore del TTL iniziale è stato impostato a 5.

4.2.1 Il routing dei messaggi

Il meccanismo di instradamento dei messaggi nella overlay network varia in base alla tipologia del messaggio: le query vengono inviate in broadcast, in modo che le possano ricevere tutti i nodi entro il raggio di ricerca fissato dal mittente, mentre le reply vengono instradate lungo il percorso inverso delle relative richieste per evitare che il mittente debba accettare troppe connessioni in ingresso per ogni singola risposta.

L'invio in broadcast comporta che un nodo, quando riceve una query proveniente da un vicino, se questa non è scaduta (cioè se il suo TTL è positivo), la inoltra agli altri suoi vicini ed eventualmente invia una risposta anche a quello che gli ha passato la richiesta. Questo meccanismo consente di raggiungere tutti i nodi entro il raggio di ricerca fissato, tuttavia, in base alla struttura della overlay network, si potrebbero generare dei cicli, quindi un nodo potrebbe ricevere una richiesta che ha già inoltrato, producendo così traffico inutile. Per riconoscere i duplicati dei messaggi di query o quelli generati da un possibile ciclo, si sfrutta il fatto che ogni messaggio ha un MSGID unico: ciascun nodo conserva all'interno della propria *tabella di inoltro* la cronologia di quelli che ha ricevuto ed inoltrato, associandoli ognuno al riferimento del vicino di provenienza (cioè alla stringa contenente l'identificatore del vicino) oppure ad un riferimento *vuoto* se la query è stata creata dal nodo stesso; se dovesse arrivare un messaggio di query con MSGID già presente nella tabella di inoltro, vuol dire che è un duplicato oppure che si è formato un ciclo, quindi viene scartato.

Mediante questa tabella di inoltro, implementata nella classe `ForwardingTable`[7], ogni nodo funge anche da "router" per le risposte relative alle richieste da esso inoltrate ai propri vicini: infatti le reply vengono costruite con lo stesso MSGID della richiesta a cui si riferiscono, per cui quando un nodo le riceve, grazie a questa tabella può conoscere il vicino a cui deve inviarle per farle giungere fino al mittente; ciò assicura che ogni risposta venga instradata nel percorso inverso rispetto a quello di trasmissione della relativa richiesta. Visto che le reply vengono instradate su un percorso "scelto prima dell'invio", non possono generare cicli o duplicati.

Naturalmente non è possibile conservare a tempo indefinito tutta la cronologia dei MSGID inoltrati, ma limitandone la quantità si otterrebbe una soluzione poco scalabile; per queste ragioni, ogni entry della tabella viene associata al tempo di arrivo della query e rimossa alla scadenza di un timer[8]. Nel momento in cui arriva una reply, il nodo legge il relativo MSGID e verifica se è presente dentro la tabella suddetta:

- se trova un riferimento vuoto, significa che la reply è arrivata a destinazione, quindi, grazie al *gestore delle ricerche*, i risultati di ricerca contenuti nella reply possono essere associati ai dati di ricerca iniziali, diventando disponibili per il client che aveva avviato la ricerca;
- se, invece, trova un riferimento ad un vicino, la reply deve essere inoltrata a questo, quindi la inserisce nella relativa coda di uscita dedicata ai messaggi diretti ad esso;
- se non trova nessun riferimento, scarta la reply; in questo caso, la risposta ricevuta potrebbe riferirsi ad una richiesta mai inoltrata, per cui potrebbe essere stata recapitata a causa di un errore, oppure più probabilmente il MSGID è scaduto e quindi non è più presente in tabella.

I messaggi di risposta includono anche i campi TTL e HOPS: il primo viene impostato col numero

7 In ogni nodo, un'istanza della classe `ForwardingTable` viene creata dalla classe `CommunicationHandler`, che, come accennato all'inizio del capitolo, è responsabile del routing di tutti i messaggi.

8 Il tempo massimo di conservazione di ogni entry nella tabella di inoltro dovrebbe dipendere dal traffico, cioè dalla frequenza delle reply, nello specifico dovrebbe dipendere in maniera inversamente proporzionale da tale frequenza. Per semplicità, il parametro impostato nel software realizzato non si basa sul traffico, ma è un valore costante. La classe `CommunicationHandler` sfrutta un oggetto `Timer` del .NET configurato in base al periodo stabilito per la pulizia periodica della tabella di inoltro, così da poter avviare un thread con il compito di cercare e rimuovere gli elementi scaduti.

di HOPS relativo al messaggio di richiesta, mentre al secondo campo viene ovviamente assegnato il valore 0. Nonostante questi due campi possano sembrare superflui nelle risposte, visto che il routing avviene mediante la tabella di inoltro, sono invece molto importanti in termini di scalabilità, poiché permettono di conoscere la distanza percorsa da una risposta e i link logici mancanti per arrivare a destinazione, pertanto, in un'ipotetica versione futura dell'applicazione, potrebbero essere utilizzati per stabilire delle priorità quando il traffico è elevato: per esempio, una risposta con TTL basso è quasi giunta a destinazione, oppure un valore di HOPS elevato indica che ha attraversato già molti nodi, ecc., quindi si potrebbe dare priorità alla trasmissione di questi messaggi, così da non rendere vano il lavoro dei nodi che le hanno trasmesse.

4.2.2 Il gestore delle ricerche

Il gestore delle ricerche è quel modulo che si occupa di gestire i risultati ottenuti dalle ricerche che vengono di volta in volta effettuate attraverso la rete dei peer: esso conserva i dati di ricerca dentro una propria *cache* e vi associa i risultati che vengono via via ricevuti dai vicini a partire dall'istante di inizio di una nuova ricerca, ma li rimuove non appena giungono a scadenza. La presenza di una cache evita l'avvio di una nuova ricerca quando ne è già presente una con le stesse opzioni, pertanto riduce il traffico di rete; tuttavia, gli elementi memorizzati al suo interno dovrebbero rimanervi per un tempo limitato, poiché le risorse di elaborazione disponibili possono variare nel tempo.

Il gestore delle ricerche è implementato nella classe `SearchManager` e viene interrogato ogni volta che l'applicazione client invoca il metodo `Search` del servizio di ricerca[9]. In questa occasione, dopo la costruzione di un nuovo oggetto `SearchData` a partire dalle opzioni specificate dal client, viene invocato il metodo `TryEnqueueNewSearch` del `SearchManager` per provare ad inserire la nuova ricerca; tuttavia, se già ne è presente una uguale, i risultati vengono subito inviati al client.

Per impattare di meno sulle prestazioni, le ricerche scadute non vengono eliminate subito, ma in due possibili occasioni, cioè durante la fase di pulizia periodica della cache[10] o durante un tentativo di accesso ad un elemento scaduto.

4.2.3 Ricerca delle risorse di calcolo condivise

La procedura di ricerca delle risorse condivise avviene mediante l'invio in broadcast dei messaggi di query, che contengono le opzioni di ricerca inviate da un'applicazione client e ricevute mediante il servizio di ricerca. La chiamata al metodo `Search` della classe `ProcessingServiceFinder` può determinare l'inizio di una ricerca: ciò si verifica nel caso in cui le opzioni di ricerca[11] specificate non corrispondono a nessuna entry nella cache del gestore delle ricerche, pertanto saranno inserite in una nuova query e quest'ultima sarà inviata a tutti i vicini, diffondendo così la ricerca nella overlay network.

I nodi inoltrano questo messaggio basandosi sulle regole di routing stabilite: quando un nodo riceve una query da un vicino, oltre a rispondere eventualmente con una reply (una reply viene generata e inviata solamente dai nodi le cui risorse di calcolo note soddisfano le opzioni di ricerca specificate nella query), la inoltra agli altri vicini previa verifica che il TTL non sia scaduto e che non si tratti di

9 Il servizio di ricerca è stato definito nel contratto di servizio `IProcessingServiceFinder` e implementato nella classe `ProcessingServiceFinder`. Questo servizio viene sfruttato dalle applicazioni client per poter dare inizio alla ricerca delle risorse di calcolo necessarie per l'elaborazione del task specificato dall'utente.

10 La classe `SearchManager` sfrutta un oggetto `Timer` del .NET configurato in base al periodo stabilito per la pulizia periodica della propria cache interna, al fine di avviare un thread con il compito di cercare e rimuovere gli elementi scaduti.

11 In generale, le opzioni per la ricerca di una risorsa di calcolo potrebbero essere abbastanza articolate, ma poiché non avrebbero influenzato il meccanismo di comunicazione realizzato, si è preferito mantenerle semplici, riducendole alla semplice coppia *nome-versione* del toolbox cercato, cioè si cerca la corrispondenza esatta di nome e versione di un toolbox.

un duplicato e mantenendo inalterato il MSGID originario.

Quando un nodo può rispondere ad una query, costruisce una nuova reply basandosi sulle regole di routing stabilite: il MSGID viene copiato dalla query (in modo che possa seguire a ritroso lo stesso percorso di quest'ultima), il campo TTL viene impostato col numero di HOPS della query ricevuta, ed infine gli URI dei servizi di elaborazione[12] che soddisfano le opzioni di ricerca vengono prelevati dalla cache delle risorse di elaborazione (ovviamente, se le opzioni di ricerca non sono soddisfatte nessun servizio di elaborazione associato a questo nodo, non viene inviata alcuna reply).

4.3 La cache delle risorse di elaborazione

Ogni nodo di ricerca viene configurato per monitorare lo stato di uno o più server di calcolo, con lo scopo di verificare se sono in linea e le risorse che mettono a disposizione: ciò avviene in maniera periodica, invocando un metodo del web service in esecuzione sul server di calcolo[13], che restituisce i nomi dei toolbox attivi su quest'ultimo. Poiché i toolbox attivi su tali server possono cambiare nel tempo, la verifica periodica consente di gestire questa eventualità e mantiene pertanto aggiornata la cosiddetta *cache delle risorse*[14], che conserva le informazioni sulle risorse disponibili nei server di calcolo noti ad ogni nodo di ricerca.

La classe `ResourceCache` implementa la cache delle risorse usando una tabella indicizzata tramite URI: ognuno di essi è l'indirizzo web di un servizio di elaborazione in esecuzione su un server di calcolo ed è associato ad un elenco delle risorse di elaborazione disponibili su quest'ultimo, almeno finché viene rilevato. Inizialmente la cache delle risorse è vuota, ma inizia a riempirsi non appena giungono i primi risultati delle interrogazioni effettuate, per poi essere aggiornata successivamente, sostituendo periodicamente l'elenco delle risorse disponibili per ogni URI. È ovvio che per svariate ragioni le interrogazioni potrebbero non avere alcuna risposta (il server di calcolo potrebbe essere sovraccarico o essersi disconnesso oppure potrebbe esserci un problema di rete): in questo caso il relativo URI non viene rimosso subito dopo la prima interrogazione non andata a buon fine, poiché potrebbe trattarsi di un problema momentaneo, ma piuttosto viene rimosso dopo un certo numero di interrogazioni consecutive fallite.

L'aggiornamento della cache viene effettuato da un thread separato, gestito da un'istanza della classe `ResourceCacheUpdater`, che interroga periodicamente[15] tutti i server di elaborazione associati[16] al nodo di ricerca e li rimuove dalla cache quando non gli forniscono nessuna risposta per un certo numero di volte consecutive: in pratica, se si verificano problemi in fase di interrogazione, il contatore degli errori associato all'URI del servizio interrogato viene incrementato, altrimenti viene riazzerato e gli identificatori dei toolbox vengono aggiornati con quelli ricevuti; gli URI dei server dai quali non si riceve risposta vengono rimossi dalla cache non appena il numero dei tentativi falliti consecutivi di comunicazione supera il valore massimo stabilito.

Il servizio del nodo di ricerca con cui si interfacciano i client recupera dalla cache di ricerca gli URI

12 Chiaramente potevano anche essere incluse altre informazioni all'interno di ogni risposta, per esempio il throughput medio di elaborazione e relativo ad ogni toolbox, la velocità media di upload, il tempo medio di presenza in rete, ecc., in sostanza tutte caratteristiche utili per la scelta del servizio di elaborazione, ma sarebbero risultate superflue nel prototipo realizzato per il presente progetto.

13 I server di calcolo sono dotati di un web service con cui è possibile interagire: tra i vari metodi in esso presenti, uno permette di ottenere l'elenco dei toolbox utilizzabili per le elaborazioni remote dei dati. I dettagli implementativi riguardanti il server di calcolo sono descritti nel capitolo 5.

14 Chiaramente, la probabilità che, in un nodo di ricerca, la cache delle risorse rispecchi lo stato effettivo dei server di calcolo dipende dall'intervallo di tempo impostato per la verifica periodica.

15 La classe `ResourceCacheUpdater` sfrutta al suo interno un oggetto `Timer` del .NET impostato secondo il periodo di refresh stabilito per la cache e che permette di avviare uno o più thread che si occupano di interrogare i server di calcolo tramite un oggetto `ProcessingServiceMonitor`.

16 Gli URI dei server di calcolo associati ad un nodo di ricerca vanno specificate all'interno del file "Servers.txt", che deve trovarsi nella stessa cartella dell'eseguibile.

che corrispondono ai criteri di ricerca specificati. Ne consegue che alla cache di ricerca potrebbero accedere due o più thread differenti (quello che la aggiorna periodicamente ed uno o più thread su cui vengono gestite le richieste provenienti dai client), ragion per cui la classe che implementa la cache delle risorse è stata sviluppata in modo che i suoi metodi siano thread-safe, assicurando l'accesso esclusivo alle sezioni critiche, cioè a quelle porzioni di codice che leggono o scrivono sulla tabella interna che implementa la cache delle risorse.

4.4 Logging delle attività

Il nodo di ricerca include anche una class library con cui effettua il logging delle attività dei propri moduli. In questo caso si è adottato un approccio più semplice rispetto a quello usato per il server di elaborazione: la classe `Logger` funziona allo stesso modo di quella implementata per quest'ultimo[17], tuttavia, invece di scrivere i messaggi su file, li mostra sulla console, con l'unico scopo di valutare il funzionamento del software sviluppato.

17 Per ulteriori dettagli sulla libreria implementata per il logging del server di elaborazione, si legga il paragrafo 5.4.

5 Server di elaborazione remota

Il server di elaborazione remota dei task è stato realizzato in un eseguibile separato rispetto a quello che implementa il nodo di ricerca; alla base di questa scelta progettuale c'è una doppia motivazione, anticipata nel capitolo 2: *riduzione della complessità* e *maggiore flessibilità*.

Il prototipo realizzato per il presente progetto si limita alla semplice esecuzione sequenziale dei task per i quali è stata richiesta l'elaborazione, senza implementare meccanismi di prenotazione di una o più risorse di calcolo, senza la possibilità di annullare i task in coda o in esecuzione, non fornendo alcuna politica di gestione dei task elaborati per ridurre l'occupazione di memoria, non limitando il numero di elaborazioni concorrenti, ecc.. L'obiettivo è stato quello di sviluppare un server di calcolo che consentisse l'elaborazione di task piuttosto semplici e di diversa natura, come complemento alle funzionalità di ricerca implementate nei peer, quindi il compito di questi ultimi è di *trovare gli URI di quei server che sono in grado di elaborare un determinato task*.

Lo scambio dei dati tra l'applicazione client e il server di elaborazione avviene grazie ad un servizio ospitato da quest'ultimo e che, tra le varie funzioni, consente l'upload dei dati relativi ad un task che si desidera elaborare e il successivo download dei risultati. Entrambe queste operazioni prevedono lo scambio di una stringa di testo in formato XML che, nel primo caso, contiene e descrive i dati da elaborare e la tipologia di elaborazione, mentre nel secondo caso contiene e descrive i risultati.

5.1 Servizio per l'elaborazione remota dei task

Il contratto di servizio ideato per l'interazione remota con il server di elaborazione dei task è stato definito nell'interfaccia `IProcessingService`, mentre la sua implementazione risiede nella classe `ProcessingService`. I metodi dedicati allo scambio dei dati tra l'applicazione client e il server di calcolo sono tre:

- `SubmitData` – permette il trasferimento dei dati relativi al task dal client al server;
- `GetState` – consente la verifica dello stato di un task precedentemente inviato;
- `GetResults` – viene usato per effettuare il download dei risultati di un task completato.

Oltre ai tre suddetti metodi, dedicati allo scambio dei dati con l'applicazione client, è stato previsto il metodo `QueryForEnabledResources`, che viene adoperato dai nodi di ricerca per ottenere una lista con i nomi dei task performer disponibili sul server di elaborazione: grazie a questo metodo, un nodo di ricerca può sapere quali risorse di calcolo sono disponibili su ogni server di calcolo ad esso associato.

5.2 Elaborazione di un task

Tipicamente, l'elaborazione remota di un task prevede i seguenti tre passi, che corrispondono ai tre metodi elencati nel paragrafo 5.1.

1. Il client invoca il metodo `SubmitData` per inviare al server di calcolo il task da elaborare e attende che la stringa XML che lo descrive arrivi a destinazione, ricevendo come risposta di avvenuta ricezione una stringa che identifica univocamente il task all'interno del server, per poterne successivamente scaricare i risultati.
    - Non appena il server riceve la stringa XML contenente il task, questa viene salvata in un file temporaneo: infatti, in generale non conviene far partire subito l'elaborazione, in quanto le risorse fisiche (memoria, processore, ecc.) disponibili al momento della ricezione potrebbero non essere sufficienti, ragion per cui conviene conservare i dati

ricevuti in un file temporaneo e schedulare opportunamente l'elaborazione del task[18].

- Per ogni task ricevuto, viene creato un nuovo oggetto `TaskMetadata`, dedicato alla conservazione di tutte le informazioni necessarie per portare a termine l'elaborazione del task: ad esempio, il percorso (sul server) del file temporaneo contenente i dati del task, il percorso (sul server) del file temporaneo sul quale memorizzare i risultati, le informazioni sulla risorsa di calcolo che dovrà elaborarlo, lo stato dell'elaborazione, gli eventuali errori avvenuti durante l'elaborazione, ecc..
- Dopo aver creato un nuovo oggetto `TaskMetadata`, questo viene passato al *gestore di elaborazione dei task*, cioè all'istanza unica di `TaskProcessingManager`, che lo schedula.

2. L'applicazione client verifica periodicamente il completamento dell'elaborazione, invocando il metodo `GetState` del servizio di calcolo e specificando la stringa che identifica in modo univoco il task sul server.
3. Quando l'elaborazione del task è stata completata, il client richiede il download dei risultati, usando il metodo `GetResults` del servizio e specificando la stringa che identifica in modo univoco il task sul server.

Qualora si dovessero verificare errori durante le chiamate ai suddetti tre metodi del servizio, questi verrebbero restituiti al client come *fault exception*, per segnalare la tipologia di errore verificatosi: se necessario, lo specifico errore avvenuto sul server viene mascherato al client, cioè quest'ultimo riceve soltanto una spiegazione generica dell'errore, con lo scopo di nascondere l'implementazione interna del servizio (cfr. pattern Exception Shielding [4]); oltre a ricevere la spiegazione generica dell'errore, il client riceve anche un identificatore dell'errore[19], affinché l'utente possa comunicarlo ad un eventuale amministratore del server di calcolo (i dettagli di ogni eccezione vengono scritti su un file di log del server, in modo che un ipotetico amministratore possa visionarli).

5.2.1 Contesto di elaborazione di un task

Un'istanza della classe `TaskPerformerContext` ha il compito di gestire l'apertura e la chiusura di uno stream sorgente e di uno stream di destinazione associati ai relativi file, così che il task descritto nel file sorgente possa essere successivamente elaborato da un *task performer*, cioè da un oggetto che implementa l'interfaccia `ITaskPerformer`, sollevando così quest'ultimo dal dover gestire tali procedure.

In particolare, invocando il metodo `Execute` su un'istanza della classe `TaskPerformerContext`, questo si occupa di aprire uno stream in lettura dal file contenente il task da elaborare ed uno stream in scrittura sul file in cui si dovranno memorizzare i risultati dell'elaborazione; tali stream vengono poi passati al metodo `Execute` della specifica istanza che implementa `ITaskPerformer`: l'oggetto che implementa questa interfaccia non ha bisogno di gestire l'apertura dei file, ma sfrutta gli stream gestiti dall'istanza di `TaskPerformerContext` che lo contiene[20].

---

18 Per semplicità, nella presente implementazione, i task ricevuti vengono messi in coda in base all'ordine di arrivo, senza tenere conto delle risorse fisiche necessarie per la loro elaborazione. Inoltre, non sono state previste verifiche sulla dimensione dei file relativi ai task da eseguire o politiche di priorità per determinate tipologie di task, pertanto ogni task ricevuto viene subito messo in coda, in attesa di essere eseguito.

19 In fase di rilevazione dell'errore, il servizio di calcolo genera anche un identificatore e lo associa alle informazioni dettagliate sull'errore verificatosi: tale associazione viene memorizzata sul file di log del server. L'identificatore così generato viene inviato al client, ma senza i dettagli dell'errore.

20 L'interfaccia `ITaskPerformer`, grazie anche alla classe `TaskPerformerContext`, garantisce scalabilità, poiché potrebbe essere sfruttata per introdurre facilmente il supporto all'impiego dei plug-in per l'elaborazione dei task: infatti, basterebbe che in ogni plug-in fosse presente almeno una classe che implementa tale interfaccia.

Nel prototipo sviluppato come server di elaborazione sono presenti due classi che implementano la suddetta interfaccia:

- la classe `StatisticsToolbox` – rappresenta un toolbox con due funzioni statistiche, cioè *MeanStdDev* e *WordsCount*, di cui la prima calcola la media e la deviazione standard di un insieme di valori numerici, mentre la seconda calcola la frequenza di ogni parola contenuta in un testo;
- la classe `MathToolbox` – rappresenta un toolbox con le due funzioni matematiche *MinMax* e *MinMaxIdx*, che calcolano il minimo e il massimo di un insieme di valori numerici (la seconda restituisce anche gli indici della prima occorrenza del minimo e del massimo).

Questi due toolbox sono piuttosto semplici, ma il loro unico scopo è stato quello di creare un po' di eterogeneità per quanto riguarda le tipologie di elaborazioni disponibili: infatti, tramite un apposito file di configurazione[21], è possibile decidere quali toolbox attivare su una precisa istanza di server di elaborazione, da cui il compito dei nodi di ricerca di *trovare gli URI dei server di calcolo con uno specifico toolbox attivo*.

La classe `TaskPerformerContextManager` è stata creata per la gestione di un insieme di oggetti `TaskPerformerContext`, al fine di potervi accedere specificando, nel metodo `TryGetContext`, il nome completo e la versione del task performer che fornisce le funzioni richieste. Questa classe di gestione implementa l'interfaccia `ITaskPerformerContextProvider`, che definisce soltanto il metodo `TryGetContext`: grazie a questa scelta, il *gestore di elaborazione dei task* può disporre di un'interfaccia di sola lettura per l'accesso ad uno specifico task performer.

5.2.2 Gestore di elaborazione dei task

La classe `TaskProcessingManager` fornisce un accesso centralizzato e thread-safe alle funzioni di accodamento e di elaborazione dei task. Ogni volta che un nuovo task giunge da un client, e dopo la memorizzazione su disco del documento XML che lo descrive, il servizio di elaborazione sfrutta il proprio backend per invocare `InsertUserTask` della classe `TaskProcessingManager`, al fine di schedulare il nuovo task:

- un nuovo oggetto `TaskMetadata`, con le informazioni inerenti l'elaborazione del task (cioè con i suoi metadati), viene inserito in una tabella e associato a un identificatore interno;
- l'identificatore interno associato ai metadati viene passato allo scheduler, affinché scheduli il task.

Quando lo scheduler decide di avviare il prossimo task, lo fa invocando il metodo `ExecuteTask` della classe `TaskProcessingManager` e passandogli l'identificatore interno: in questo modo, può recuperare i metadati del task e iniziarne l'elaborazione con il task performer corretto.

Durante l'elaborazione del task, cioè a partire dal momento in cui viene caricato il documento XML che descrive il task e fino al momento in cui viene scritto un nuovo documento XML per conservare i risultati dell'elaborazione, potrebbero verificarsi tre tipologie d'errore riguardanti il task performer usato e rappresentate da altrettante eccezioni[22]:

- `TaskDataReadException` – questa eccezione è dovuta ad un errore durante la lettura del file contenente la descrizione del task, che non è stato scritto nel formato corretto;

21 Il nome e la versione di ogni toolbox da attivare per un singolo server di calcolo vanno specificati all'interno del file "Resources.txt", che deve trovarsi nella stessa cartella dell'eseguibile.

22 I tre tipi di eccezione che possono essere lanciate durante l'elaborazione di un task, vengono rilanciate all'interno di un'eccezione `TaskPerformerException` ed impostate come eccezioni interne ad essa.

- `TaskProcessingException` – descrive gli errori riguardanti l'elaborazione vera e propria del task, cioè quelli a carico del task performer;
- `TaskResultWriteException` – infine, questa eccezione dipende dal task performer, che ha commesso qualche errore durante la codifica dei dati nel formato XML.

Poiché questa tipologia di errori si riferisce all'elaborazione di un task, nel caso in cui si dovessero verificare, la descrizione ad essi relativa verrebbe inserita nei metadati associati al task, in modo che l'utente possa leggerla dopo aver effettuato il download dei risultati relativi al task.

5.3 Formato dei dati trasferiti

La struttura del documento XML che descrive un task deve comprendere un tag radice denominato `task` e i seguenti tag annidati:

- il tag `component` permette di indicare il task performer necessario per elaborare il task e richiede che siano specificati gli attributi `className` e `classVersion`, in cui specificare rispettivamente il nome e la versione del task performer necessario per l'elaborazione;
- il tag `function` serve per specificare la funzione da applicare ai dati che costituiscono il task e deve essere specificata nell'attributo `functionName`;
- il tag `data` contiene i dati relativi al task e il formato del suo contenuto è arbitrario, ovvero dipende dallo specifico task performer; ciò assicura flessibilità nella descrizione di un task.

La struttura del documento XML contenente i risultati dell'elaborazione di un task dipende dal task performer che lo ha elaborato.

5.4 Logging delle attività

Il server di elaborazione comprende anche alcune classi con cui effettua il logging delle attività dei propri moduli, creando un nuovo file di log ogni volta che viene avviato e scrivendo su di esso tutti i messaggi di log che vengono via via passati all'unico oggetto `Logger` istanziato: la scrittura su file avviene grazie alla classe `FileLogHandler`[23], che però non effettua immediatamente la scrittura di ogni messaggio, ma la posticipa fino a quando non raccoglie un certo numero di messaggi, al fine di ridurre il numero di accessi al disco.

23 La classe `FileLogHandler` implementa l'interfaccia `ILogHandler`.

6 I client per interagire con l'applicazione distribuita

Per poter usufruire delle risorse di calcolo monitorate dalla rete di ricerca, sono state realizzate due versioni di un'applicazione client, entrambe in grado di interagire sia con i nodi di ricerca sia con i server di calcolo, ma di cui una è stata sviluppata per poter funzionare su sistemi desktop Windows, mentre l'altra è dedicata ai dispositivi mobili Windows Phone 7.1: entrambe condividono lo stesso codice di base[24] (con differenze minime) che implementa la business logic, ma usano accorgimenti diversi per l'implementazione della UI (di cui la figura 3 ne mostra gli screenshot) e dei proxy per la comunicazione con il servizio di ricerca e con quello di elaborazione, affrontati rispettivamente nel capitolo 4 e nel capitolo 5.

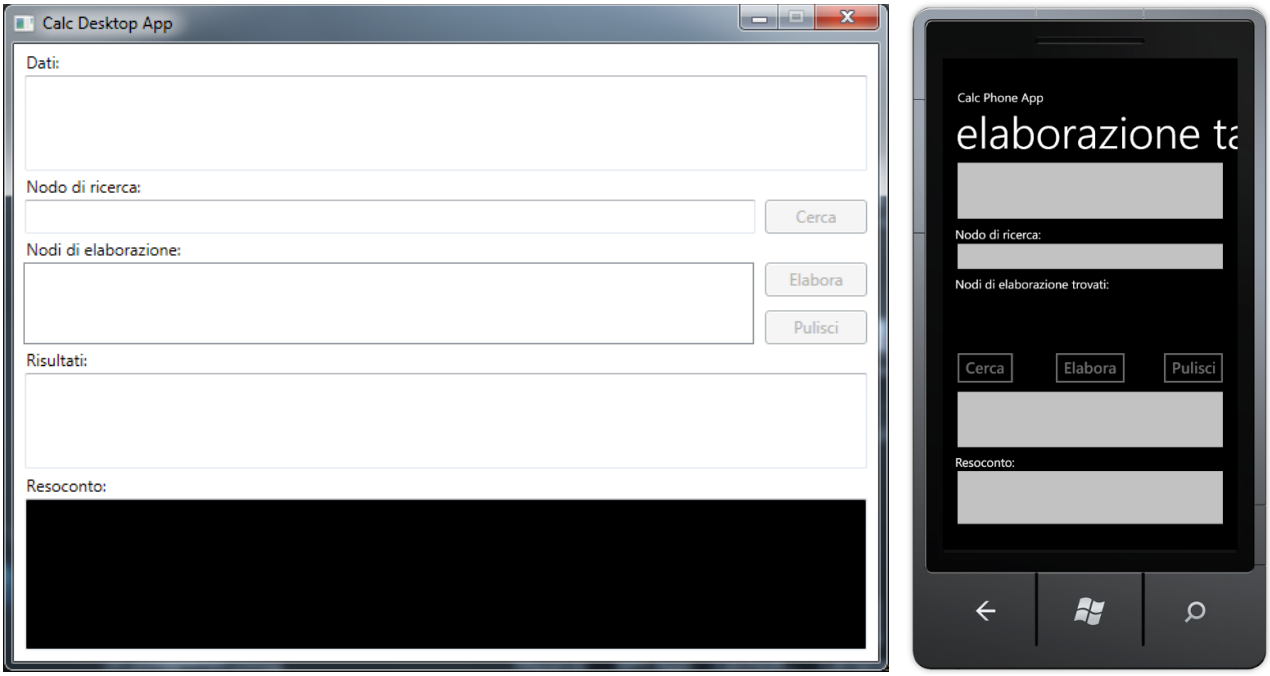

Figura 3. UI dell'applicazione client su Windows 7 (a sinistra) e sull'emulatore per Windows Phone 7.1 (a destra).

Per la realizzazione delle due versioni dell'applicazione sono stati impiegati i due framework WPF[25] e Silverlight[26] for Windows Phone (entrambi basati su XAML) ed il pattern MVVM per favorire la separazione del codice di business logic dall'interfaccia grafica.

Per quanto riguarda il design delle UI, visto che i vincoli del progetto non hanno imposto particolari requisiti di usabilità, si è data priorità ai tempi di realizzazione delle stesse.

6.1 Funzionamento del client

L'applicazione client è in grado di sfruttare entrambe le funzionalità implementate nell'applicazione distribuita, ovvero la ricerca di una risorsa di calcolo in base al task specificato, e l'elaborazione del task su uno dei server di calcolo trovati durante la fase di ricerca.

Dopo aver immesso, nell'apposito campo, la stringa XML contenente i dati del task che si desidera

24 Poiché si desiderava sperimentare fino a che punto fosse possibile condividere lo stesso codice di base, ovviamente non è stato possibile adottare particolari accorgimenti nello sviluppo dell'applicazione per Windows Phone.

25 WPF, acronimo per Windows Presentation Foundation, è un framework per la realizzazione di interfacce grafiche.

26 Silverlight è un framework che permette di creare applicazioni multimediali interattive per browser, le cosiddette *Rich Internet Applications* (RIA). Silverlight for Windows Phone è una versione Silverlight adattata per lo sviluppo di applicazioni destinate ai dispositivi mobili Windows Phone.

elaborare, occorre specificare anche l'indirizzo del nodo di ricerca che si vuole sfruttare per avviare la ricerca, ed infine fare clic sul pulsante *Cerca*: la risorsa di calcolo necessaria per elaborare il task è indicata all'interno della stringa XML che lo descrive e verrà cercata dalla rete dei nodi di ricerca. Mentre la ricerca è in corso, tutti i pulsanti vengono disabilitati: per semplicità, non è stata prevista la possibilità di interrompere la ricerca, ma la sua durata è stata fissata ad un intervallo di tempo di 30 secondi, durante il quale il client effettua polling ogni 5 secondi per recuperare gli eventuali URI trovati dalla rete dei nodi di ricerca.

Quando la fase di ricerca si conclude, i pulsanti vengono abilitati nuovamente e tutti gli URI relativi ai nodi di elaborazione trovati, se presenti, possono essere visualizzati nell'apposita sezione, disposti in ordine di ricezione. Il resoconto viene aggiornato con le informazioni principali sugli eventi che si verificano durante la comunicazione col nodo di ricerca (gli eventuali errori di comunicazione, il numero di risultati ricevuti, ecc.).

Per elaborare il task specificato, occorre selezionare uno dei nodi di elaborazione trovati e fare clic sul pulsante *Elabora*. Ciò determina di fatto il trasferimento del task sul server di calcolo, sul quale ha luogo l'elaborazione, e il successivo download dei risultati[27]: questi ultimi sono anch'essi descritti in formato XML e vengono mostrati nell'apposito campo. In tale fase, il precedente resoconto viene sovrascritto con le informazioni sugli eventi che avvengono durante la comunicazione col server di calcolo.

6.2 Separazione del codice della UI dal resto del codice

Per implementare le UI del client, sono state seguite le regole del pattern MVVM [3], in base alle quali si è proceduto con la suddivisione dello strato di presentazione dell'applicazione in tre parti, ognuna con una specifica responsabilità:

- il *Model* è formato da tutte quelle entità che conservano i dati dell'applicazione, che nel caso dell'applicazione in oggetto vengono mantenute solamente in memoria, quindi la persistenza (su file, database, ecc.) non è stata prevista;
- la *View* rappresenta l'interfaccia grafica ed in questo caso è stata implementata all'interno del file "MainPage.xaml"; nel code-behind viene creata una nuova istanza della classe che implementa il *ViewModel*;
- il *ViewModel* assicura gli scambi informativi tra la View e il Model ed è stato implementato classe `MainViewModel`, che deriva dalla classe astratta `ViewModelBase`.

Il binding dei comandi associati ai pulsanti della UI è assicurato da altrettante istanze di una classe che implementa l'interfaccia `Icommand`; un oggetto di questo tipo serve per descrivere i due aspetti principali di un comando, cioè la *condizione* che lo abilita e l'*azione* da compiere:

- la condizione necessaria per abilitare il comando deve essere valutata all'interno del metodo `CanExecute`, che quindi deve restituire `true` se la condizione stabilita è verificata, `false` se invece non è verificata; ciò fa sì che il corrispondente pulsante della UI venga abilitato o disattivato in maniera del tutto trasparente per lo sviluppatore;
- l'azione da compiere in risposta al trigger del comando deve essere specificata all'interno del metodo `Execute`; questa azione viene ovviamente eseguita nel momento in cui si fa clic sul corrispondente pulsante.

Nel presente progetto, per semplificare la definizione degli specifici comandi, è stata creata la classe

27 Dal momento in cui il task viene inviato al server di calcolo, il client interroga periodicamente (ogni 5 secondi, per un massimo di 6 tentativi) quest'ultimo, per verificare se l'elaborazione del task è stata completata ed eventualmente procedere con il download dei risultati.

`CommandHandler<T>`, che implementa l'interfaccia `ICommand` e definisce anche un costruttore in cui specificare l'azione da associare al comando e la condizione per abilitarlo: l'azione va specificata come oggetto `Action<T>`, mentre la condizione come oggetto `Predicate<T>`. In tal modo, i vari comandi da associare ai pulsanti della UI possono essere inizializzati in poche righe di codice nel costruttore della classe che implementa il ViewModel.

6.3 Comunicazione con i servizi

Per le motivazioni spiegate nel capitolo 2, l'applicazione client sfrutta due proxy configurati con il *BasicHttpBinding* per dialogare con i servizi remoti: il primo di questi proxy è stato generato per comunicare con il servizio di ricerca attivo sul nodo di ricerca, mentre il secondo è stato generato per le comunicazioni col servizio di elaborazione remota. Per generare i proxy sono stati utilizzati i due strumenti forniti con l'ambiente di sviluppo: *SvcUtil.exe* [5] e *SlSvcUtil.exe* [6].

Bibliografia

[1]	Wikipedia. “Gnutella”. http://it.wikipedia.org/wiki/Gnutella
[2]	Wikipedia. “Universally Unique Identifier”. http://en.wikipedia.org/wiki/Universally_unique_identifier
[3]	Microsoft. “WPF Apps With The Model-View-ViewModel Design Pattern”. http://msdn.microsoft.com/en-us/magazine/dd419663.aspx
[4]	Microsoft. “Web Service Security. Scenarios, Patterns, and Implementation Guidance for Web Services Enhancements (WSE) 3.0”, Exception Shielding. https://msdn.microsoft.com/en-us/library/aa480591.aspx
[5]	Strumento ServiceModel Metadata Utility Tool (Svcutil.exe). https://msdn.microsoft.com/it-it/library/aa347733(v=vs.100).aspx
[6]	Using SLsvcUtil.exe to Access a Service. https://msdn.microsoft.com/it-it/library/cc197958(v=vs.95).aspx