Relazione di Sistemi.
Progetto : Telecontrollo.

Milano (MI) 10 / 06 / 2010

Reaper

Applicativo Server per l'erogazione di:

- Servizi HTTP - PHP + MySQL
- Servizi di Telecontrollo

Realizzato da: Vanzo Luca Samuele.
Software usati: Visual Studio 2008 Team Edition, MySQL Workbench.

- Prefazione..... Pag. 1
- Jennic. Il Progetto..... Pag. 2
  - Cenni sul Progetto: Telemedicina.
  - WI-FI: Protocollo ZigBee.
  - Applicazione: Mirò.
- Windows Services..... Pag. 4
  - Cos'è un Servizio e perché usarlo.
  - Gestione di un Servizio Windows.
  - Il Registro Eventi di Windows: EventLogger.
- Applicativi: Reaper Server..... Pag. 9
  - Il Socket. Carneade! Chi è costui?
  - Protocollo TCP / IP.
  - Protocollo HTTP.
  - Struttura di un'applicazione Server: Mono / Multi – Utenza.
    - ✓ Reti di Calcolatori: Logica Client / Server.
    - ✓ Reti di Calcolatori: Logica P2P.
  - Un controllo per monitorarli tutti… Il Keep-Alive (Time-To-Live).
- Gerarchie di Utenze..... Pag. 12
  - Struttura Server per Autenticazioni Multi-Utenza: Il Database.
  - Una funzione per limitarli tutti… Le Policy.
- Logiche di Ottimizzazione..... Pag. 13
  - Il Multi-Threading.
  - Logica di Queuing Asyncrono.
  - La Garbage Collection.
- Database del Server..... Pag. 15
  - L'Enhanced ER-Diagram e le Tabelle risultanti.
  - Commenti.
- Software utilizzati..... Pag. 16
  - Visual Studio Team Edition 2008: C-Sharp Compiler.
  - MySQL 5 e MySQL Workbench 5.2 OSS.
  - Dreamweaver CS4.

L'esposizione dei contenuti è pensata secondo il modello "Manuale Utente" di modo da poter affrontare il più efficacemente possibile gli argomenti trattati e sopra citati. Inoltre ho ben pensato di seguire una scaletta ordinata, secondo grado di difficoltà, sicchè ogni argomento esposto risulti, dove possibile, collegato e d'introduzione ai seguenti.

Premetto sin da subito che il Progetto in questione, risultando molto ampio da illustrare, verrà suddiviso in due sezioni che verranno, secondo accordi precedenti, trattate separatamente dal sottoscritto, Cerizza Davide e Ricotta Marco.

Nel particolare, e per quanto mi riguarda, mi soffermerò nei vari aspetti di seguito elencati:

- Programmazione e architettura Server Multi – Utenza.
- Server e Protocollo HTTP.
- Reti Client / Server e P2P.
- Implementazione di un Database.
- Ottimizzazioni successive.
- Commenti su Software, Linguaggi e Realizzazione.

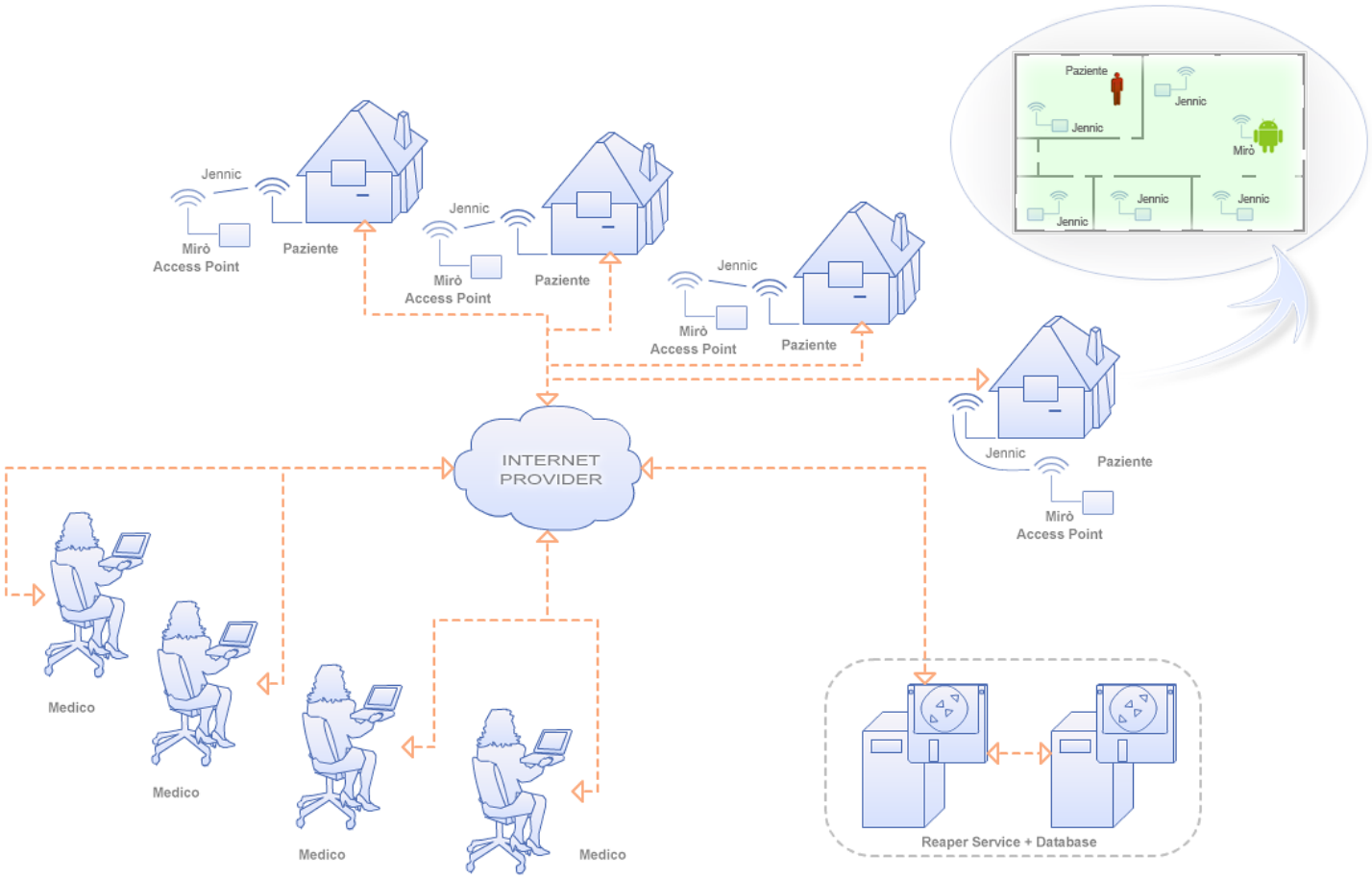
La realizzazione della parte server è in parte dovuta ad esperienze precedenti, quali sviluppo "**Web** / **Database**" e "**Servizi Web 3.0** (*Applicativi di Massa Online*)", e in parte frutto di uno studio individuale dei testi inglesi:

- *Titolo:* **Professionl C# 2008.**
  Casa editrice: **Wrox**.

- *Titolo:* **Professional Multicore Programming.**
  Casa editrice: **Wrox.**

Cenni sul Progetto: Telemedicina.

Il progetto di presenta come una possibile soluzione al controllo di pazienti e abitazioni domotiche, a distanza, mediante l'utilizzo di apposite schede WI-FI operanti ed espandibili su scala gerarchica. Nel particolare il progetto presenta un "**Server Remoto**" (*Reaper*) con "**Database MySQL**" e una serie di "**N client**" (*Medici e Pazienti*) connessi fra loro mediante Internet. Per quanto riguarda la rete locale del paziente il Client, in funzione Server (*Locale*), gestisce il **Jennic** e **Mirò** per poi limitarsi al solo invio di temperature o misurazioni varie al "**Server Remoto**" per la registrazione. Il medico in remoto potrà, così, gestire i pazienti e visionare i dati ottenuti.



WI-FI: Protocollo ZigBee.

ZigBee è il nome di una specifica per un insieme di protocolli di comunicazione che utilizzano antenne a bassa potenza basate su standard **IEEE 802.15.4** per Reti **WPAN**: Wireless Personal Area Network. Il modello opera nelle frequenze radio assegnate per scopi industriali, scientifici e medici (ISM): da 868 MHz a 2,4 GHz. Questa tecnologia ha lo scopo di essere più semplice ed economica rispetto altre WPAN come, ad esempio, il Bluetooth trovando applicazione in ambienti **Embedded** che richiedano un basso **Transfer Rate** a bassi consumi.

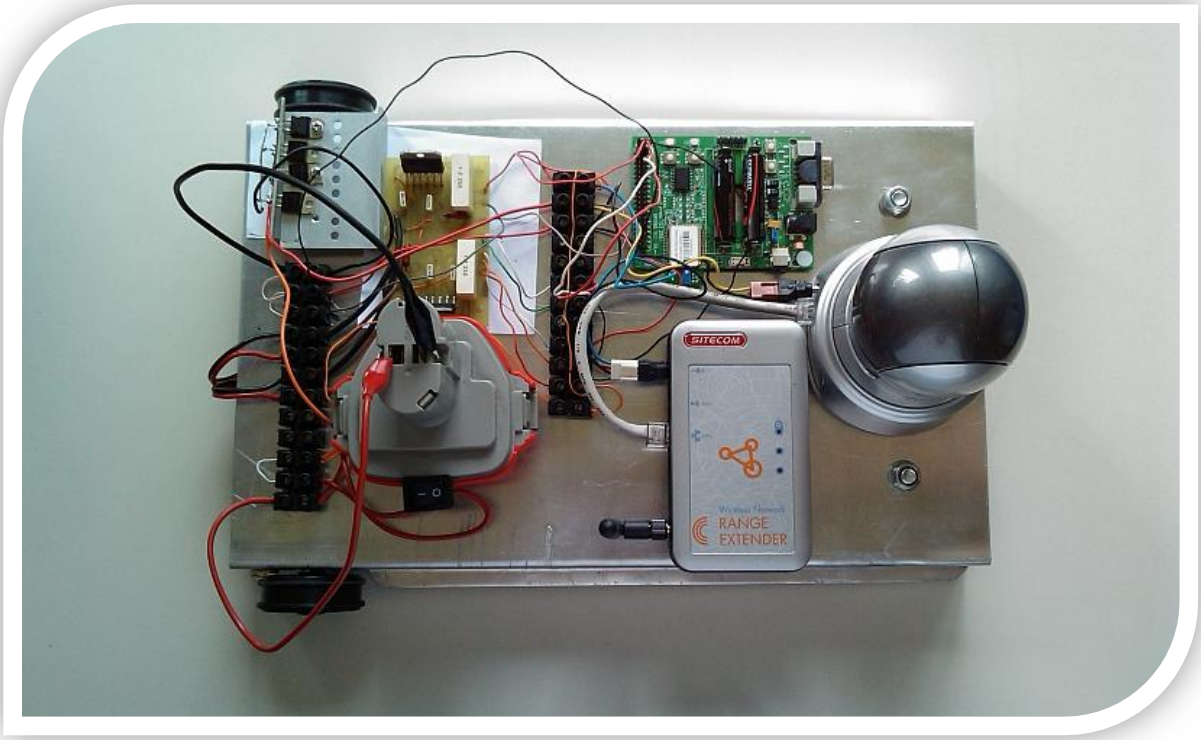
L'obiettivo quindi è quello di fornire i mezzi necessari alla creazione di una Rete Magliata Wireless dinamica, economica e autogestita che possa essere utilizzata per scopi industriali, reti di sensori, domotica e mobili.
Ci sono tre differenti tipi di dispositivo ZigBee:

- **ZigBee Coordinator** *(ZC)*: costituisce la radice di una rete ZigBee e può operare da ponte tra più reti. Ci può essere un solo Coordinator in ogni rete ed è in grado di memorizzare informazioni circa la sua rete di lavoro.

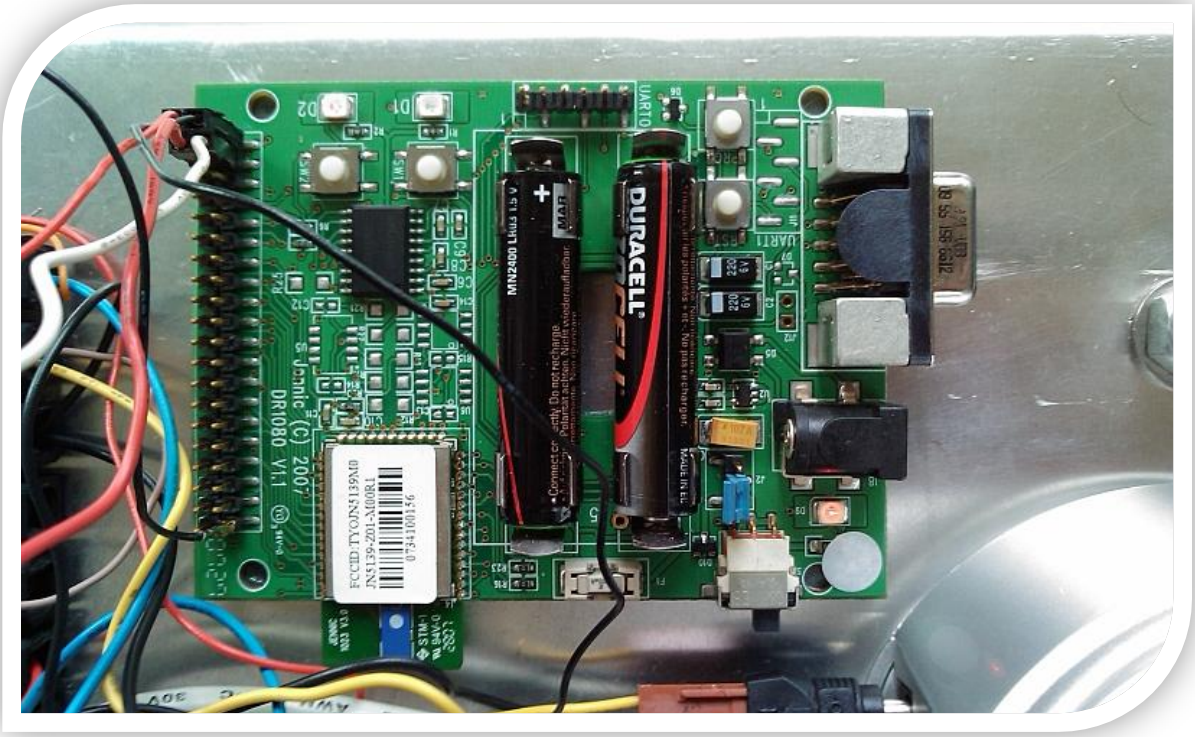
- **ZigBee Router** *(ZR)*: questi agiscono come router (instradatori), o meglio ponti, per i dati da e verso altri dispositivi.

- **ZigBee End Device** *(ZED)*: limitato alle sole funzionalità di dialogo con il suo nodo padre (Coordinator o Router a seconda della tipologia in uso), non possono trasmettere dati in entrata e, quindi, sono spesso più economici rispetto ai ZR o ai ZC a causa della poca memoria necessaria al loro funzionamento.

Applicazione: Mirò.
Visione completa del Robot Mirò. Sullo chassis (telaio) è possibile notare l'AccessPoint WI-FI, Jennic (in seguito ingrandita), una Web Cam-IP, una batteria da 12v ed infine il cablaggio e la componentistica necessaria al funzionamento del Mirò.



Scheda JNE-5139 (Jennic) per il controllo remoto del Robot Mirò alimentata da 2 batterie.
Il controllo dei motori avviene mediante segnali inviati tramite pin IDE.



Cos'è un Servizio e perché usarlo.
Secondo Microsoft, neanche lei sa esattamente cosa sia, un servizio è:

- **1° Definizione:** *" Applicazione, routine o processo che esegue una funzione di sistema specifica per supportare altre applicazioni, in modo particolare a basso livello, ovvero un livello prossimo all'hardware ".*
- **2° Definizione:** *" Un processo, o insieme di processi, che aggiunge funzionalità a Windows atto a fornire supporto ad altri programmi "*.
- **3° Definizione:** *" Un servizio è un'applicazione eseguita in background (in sottofondo), indipendentemente da qualunque sessione di utente ".*
- **4° Definizione:** *" Un servizio è un oggetto eseguibile, installato in un registro e gestito dal Service Control Manager "*.

Consci delle definizioni di mamma Microsoft una sintesi accettabile potrebbe essere la seguente:

> ***" Un servizio è un'applicazione, routine o processo eseguito in background in grado di fornire funzionalità di basso livello, strettamente integrate con il sistema operativo, a componenti di Windows o/e Applicazioni di terze parti indipendentemente da qual si voglia sessione e/o livello (privilegi) utenza… "***

Soddisfatti della definizione appena sfornata passiamo ai vari livelli operativi di un servizio:

- **Local System**
  Il servizio opererà con il massimo dei privilegi utente del sistema locale, ma presenterà un'utenza anonima nella rete.
- **Network Service**
  Similare a "Local Service" non opera con privilegi nel sistema locale. Questo implica che il suddetto account - service dovrà essere usato solo per servizi richiedenti risorse dalla rete.
- **Local Service**
  Presenterà le credenziali ed informazioni del PC in uso a tutti i vari server remoti a cui sarà collegato.
- **User**
  Permette la selezione del tipo di account da utilizzare (*Service Control Manager)*.

Alla luce dei fatti sappiamo dunque che un **Servizio** è un **Processo** ovvero, sempre secondo Microsoft, "*uno spazio degli indirizzi, virtuale, e informazioni di controllo necessari per l'esecuzione di un programma o/e di uno o più* ***Thread***".

La domanda ora sorge spontanea: **Cosa sarai mai un Thread?**

Un **Thread**, molto nel generico, si può definire come una parte di programma eseguita in modo indipendente, in alcuni casi anche contemporaneamente ad altri **Thread**, così da meglio sfruttare le risorse hardware a nostra disposizione.
Molte grosse applicazioni adoperano soluzione in **Multi-Threading**, cioè usano più **Thread** per eseguire lavori simultanei su più CPU o più "Unità di Calcolo" di una singola CPU.

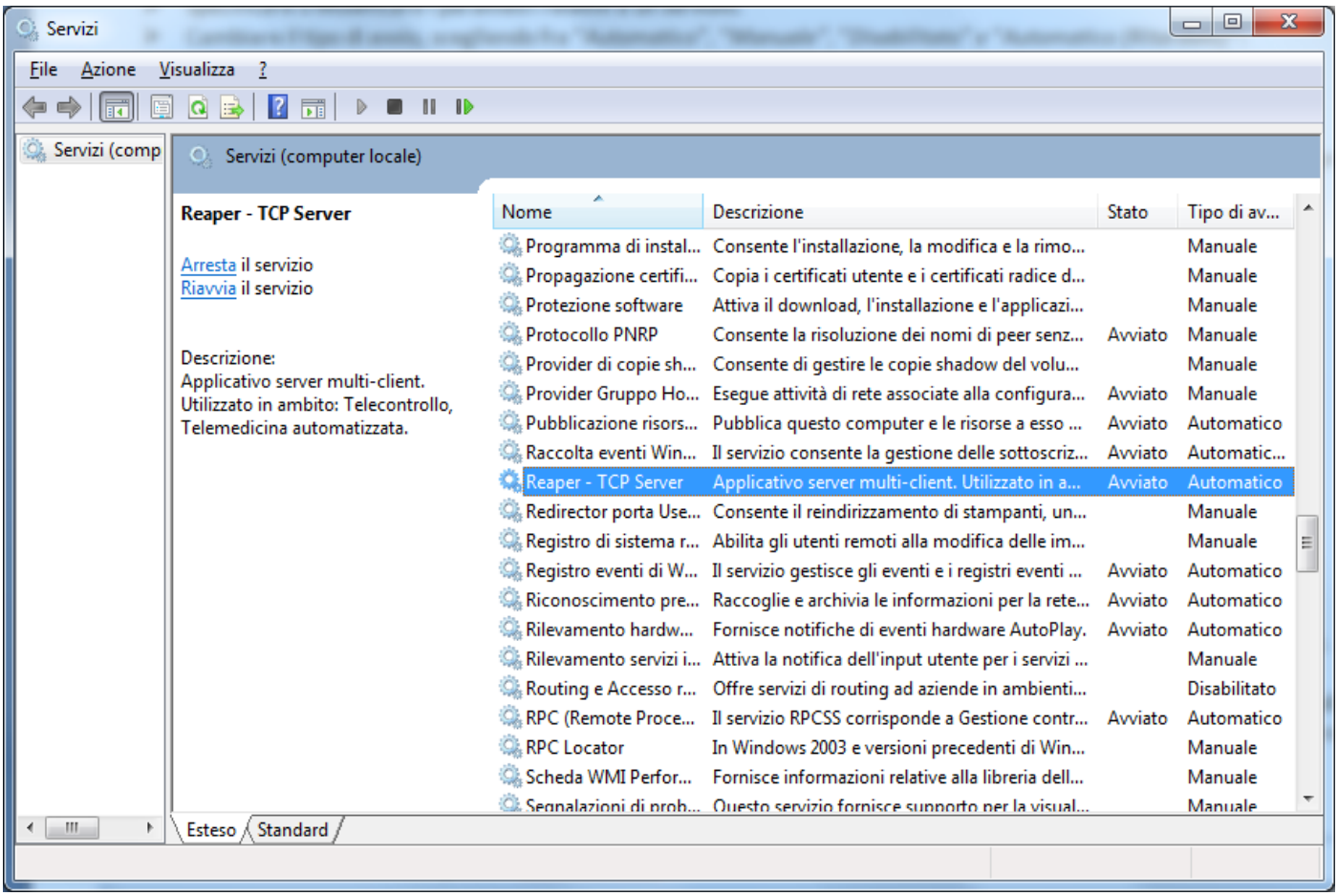
Ora, compreso cos'è e come funziona un Servizio Windows, possiamo finalmente discutere circa il perché un Servizio risulta più indicato, rispetto alla semplice "*Windows Form Application* (WPF)", in alcuni contesti d'utilizzo. Per prima cosa i Servizi consentono di creare applicativi la cui esecuzione perdura per tutta la sessione di Windows. Possono essere avviati automaticamente all'avvio del computer, sospesi e riavviati e non mostrare alcuna interfaccia utente riducendo, quindi, i costi di esecuzione su RAM e CPU.

Grazie a queste caratteristiche, i Servizi Windows sono perfetti oggetti da utilizzare per applicazioni server o in generale quando si ha la necessità di funzionalità a esecuzione prolungata che non interferiscano con altri utenti dello stesso computer.
Infine l'interazione del servizio Windows con l'utente o con altri oggetti finestra (Form) deve essere progettata con accortezza considerando scenari in cui, ad esempio, non vi siano utenti connessi o l'utente disponga di un set di oggetti desktop limitato.
In questi casi risulta più indicato scrivere una "*Windows Form Application* (WPF)" che venga eseguita sotto il controllo dell'utente riducendo notevolmente errori umani che potrebbero causare l'arresto del servizio.

Gestione di un Servizio Windows.

Dopo aver installato un Servizio può essere gestito mediante "***Pannello di controllo***" di Windows, oppure digitando "**Services.msc**" nella casella dei comandi di "***Esegui***" del menù "***Start***". La console di gestione "***Servizi***" fornisce una breve descrizione delle funzioni eseguite ed il percorso del file eseguibile "**.exe**", lo stato corrente del servizio, il tipo di avvio, le dipendenze e l'account scelto per l'esecuzione. Mediante il pannello l'utente quindi si può:

- Avviare, Arrestare, Sospendere (*pausa*) o riavviare un Servizio.
- Specificare o Modificare i parametri relativi a un servizio.
- Cambiare il tipo di avvio, scegliendo fra "*Automatico*", "*Manuale*", "*Disabilitato*" e "*Automatico (Ritardato)*":
    - **Automatico:** avvia il servizio durante il caricamento del desktop utente.
    - **Manuale:** avvia il servizio quando richiesto dall'utente o quando richiesto da un'applicazione (*se si hanno i privilegi*).
    - **Disabilitato:** disabilita completamente il servizio ed impedisce l'esecuzione delle relative dipendenze.
    - **Automatico (Ritardato):** introdotto da "*Windows Vista*" avvia il servizio dopo un breve intervallo di tempo "*Post-Operazioni critiche di avviamento*" di modo che l'avviamento del Servizio avvenga più velocemente.
- Cambiare l'account di esecuzione (*se installato come "User"*).
- Configurare le opzioni di ripristino in caso di crash dell'applicazione.



Fin qui nulla di complicato poiché seguiti passo a passo da una qualche interfaccia utente fornita da Windows stesso.
Ma come tutte le cose apparentemente facili hanno anche un volto complicato, il quale, risulterà più efficace e reattivo.

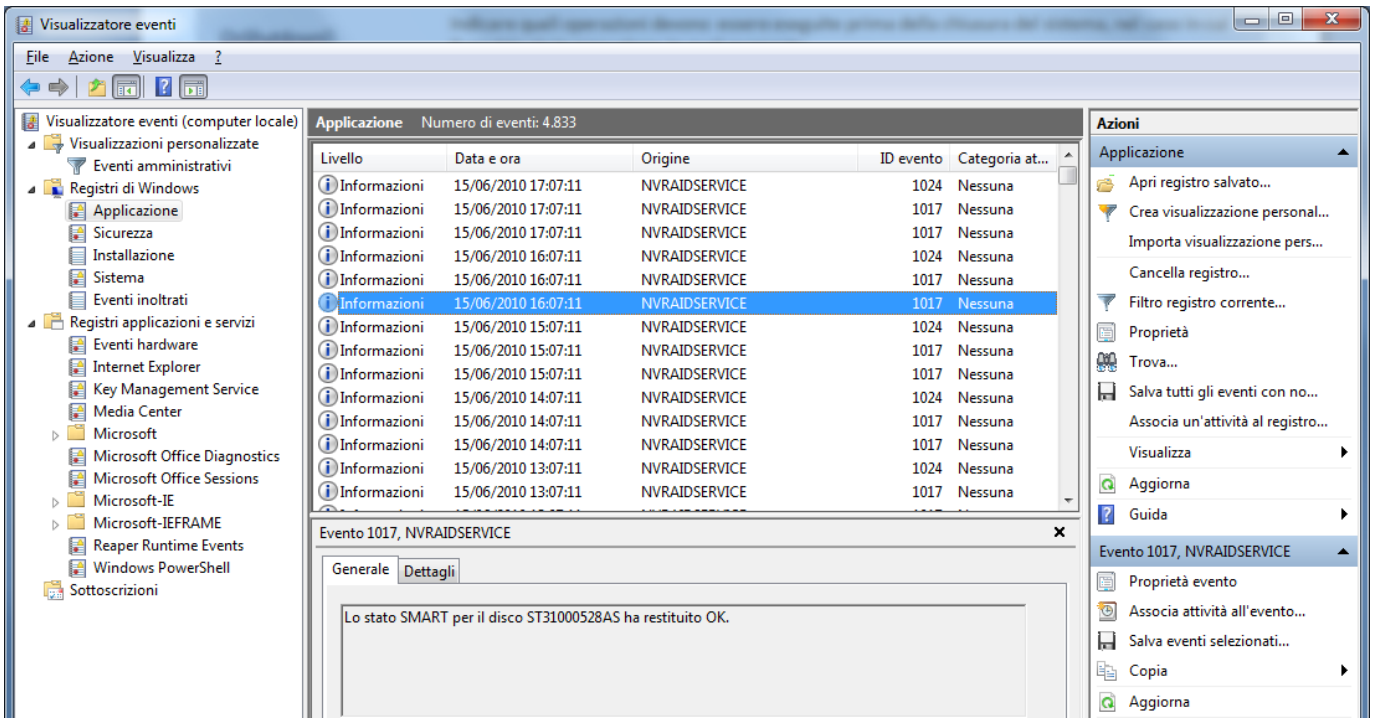
Questo "*volto*" è una classe appartenente allo Spazio "***System.ServiceProcess***": il suo nome è **ServiceController**.

Mediante questa classe è possibile, infatti, accedere per intero al servizio desiderato; che sia più o meno di nostra produzione.
Nel particolare esso rappresenta un **Servizio Windows** consentendo a noi poveri programmatori l'esecuzione, l'arresto, la sospensione e la manipolazione del servizio stesso. La classe **ServiceBase** invece, interna al copro del servizio, espone i metodi di cui dovremo eseguire l'override (*sovrascrittura*) necessario al controllo autonomo e personalizzato del servizio in base al verificarsi, o meno, di particolari eventi riportati di seguito per comodità.

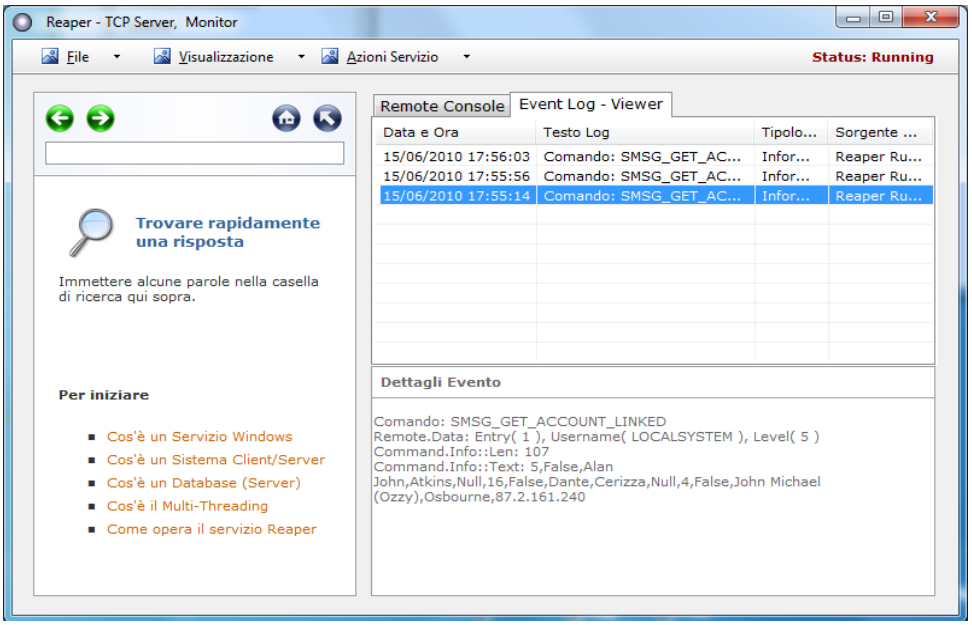
Metodo	***Eseguirne l'override per…***
OnStart()	Indicare quali operazioni devono essere eseguite quando il servizio viene avviato. Per un'esecuzione utile del servizio, occorre scrivere il codice in questa routine.
OnPause()	Indicare quali operazioni devono essere eseguite quando il servizio viene sospeso.
OnStop()	Indicare quali operazioni devono essere eseguite quando il servizio viene arrestato.
OnContinue()	Indicare quali operazioni devono essere eseguite quando il servizio riprende il normale funzionamento dopo la sospensione.
OnShutdown()	Indicare quali operazioni devono essere eseguite prima della chiusura del sistema, nel caso in cui il servizio sia in esecuzione in quel momento.
OnCustomCommand()	Indicare quali operazioni devono essere eseguite quando il servizio riceve un comando personalizzato.
OnPowerEvent()	Indicare quali operazioni devono essere eseguite quando il servizio riceve un evento di risparmio energia, come quelli che indicano lo stato di batteria scarica o sospensione.

Il Registro Eventi di Windows: EventLogger.

Bene siamo pronti per donare la parola al nostro **Servizio Windows**. Come possiamo fare? Possiamo, per esempio, reindirizzare l'output da console a una qualche applicazione di nostra creazione. Ma anche no!! Perché complicarci la vita inutilmente??
Usiamo piuttosto uno dei pochi strumenti utili a nostra disposizione su Windows: **Il Registro Eventi**.



Ebbene sì Windows ha un registro eventi e da bravi programmatori sarà ora di apprendere la via più breve per usarlo.
Giusto al caso nostro troviamo una classe del **.NET Framework** (*versione 2 o successiva*) adibita, guarda te che fortuna, alla gestione degli eventi: appartenente allo Spazio "**System.Diagnostics**" troviamo dunque la classe **EventLog.**



Utilizzando la classe prima citata, quindi, è possibile leggere dai log esistenti, inserire voci nei log, creare o eliminare origini eventi, eliminare log e rispondere alle voci. È inoltre possibile crearne di nuovi durante la creazione di un'origine eventi.

Codice di esempio "**Visual C-Sharp 2008 Framework .NET 2.0**"

```
// Spazio dei nomi (Framework)
using System;
using System.Diagnostics;


// Inizio Classe Applicativo
class MySample{

    public static void Main(){

        // Creo una nuova "Origine Eventi" se non esiste nel Registro Eventi.
        if(!EventLog.SourceExists("MySource"))
        {
            EventLog.CreateEventSource("MySource", "MyNewLog");
        }

        // Creo un'istanza (Handler) della Classe: EventLog.
        EventLog myLog = new EventLog();

        // Seleziono l'Origine Eventi dal Registro.
        myLog.Source = "MySource";

        // Scrivo un Log nell'Origine precedentemente selezionata.
        myLog.WriteEntry("Mio Log... Ciao Mondo!!");

    }
}
```

Inoltre è possibile specificare il tipo di Log fornendo informazioni aggiuntive per la voce. Ogni evento deve essere di un unico tipo, non è quindi possibile combinare i tipi di eventi per una voce. Il Visualizzatore eventi utilizza tale tipo per scegliere l'icona da visualizzare nella vista elenco del log. I valori accettati per la Tipologia sono:

Nome membro	*Descrizione*
Error	Evento errore. Indica un problema importante che l'utente dovrebbe conoscere. Solitamente la mancanza di funzionalità o dati.
FailureAudit	Evento di controllo non eseguito correttamente. Indica un evento di protezione che si verifica quando un tentativo di accesso controllato non ottiene esito positivo. Ad esempio un tentativo non riuscito di apertura di un file.
Information	Evento informativo.
SuccessAudit	Evento di controllo eseguito correttamente. Indica un evento di protezione che si verifica quando un tentativo di accesso controllato ottiene esito positivo. Ad esempio un collegamento eseguito con esito positivo.
Warning	Evento di avviso. Indica un problema non immediatamente significativo ma che può creare condizioni in grado di causare problemi in futuro.

Infine è possibile interrogare e gestire un Servizio anche sa "**Linea di Comando** (*CMD*)" mediante il comando: "**SC**".
Di seguito una schermata tipo del gestore testuale in console di Windows.

```
Microsoft Windows [Versione 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. Tutti i diritti riservati.

C:\Users\Samu>sc

DESCRIZIONE: (Controllo servizio) è un programma della riga di comando
             utilizzato per la comunicazione con Gestione controllo servizi e con i
             servizi.

SINTASSI: sc <server> [comando] [nome servizio] <opzione1> <opzione2>...

        L'opzione <server> è in formato "\\NomeServer"
        Per ulteriori informazioni sui comandi digitare: "sc [comando]"


        Comandi:
          query-----------Esegue una query sullo stato di un servizio
                          o enumera lo stato dei tipi di servizi.
          queryex---------Esegue una query sullo stato esteso di un
                          servizio o enumera lo stato dei tipi di servizi.
          start-----------Avvia un servizio.
          pause-----------Invia una richiesta di controllo PAUSE a
                          un servizio.
          interrogate-----Invia una richiesta di controllo
                          INTERROGATE a un servizio.
          continue--------Invia una richiesta di controllo
                          CONTINUE a un servizio.
          stop------------Invia una richiesta di controllo STOP a un servizio.
          config----------Modifica la configurazione di un servizio
                          (permanente).
          description-----Modifica la descrizione di un servizio.

        I comandi seguenti non richiedono un nome di servizio:
        sc <server> <comando> <opzione>
          boot------------(ok | bad) Indica se l'ultimo avvio
                          deve essere salvato come l'ultima
                          configurazione di avvio valida
          Lock------------Blocca il database del servizio
          QueryLock-------Esegue una query sullo stato LockStatus del database
                          di Gestione controllo servizi (SCManager)

...altri comandi rimossi da me per necessità...


ESEMPIO: sc start NomeServizio
```

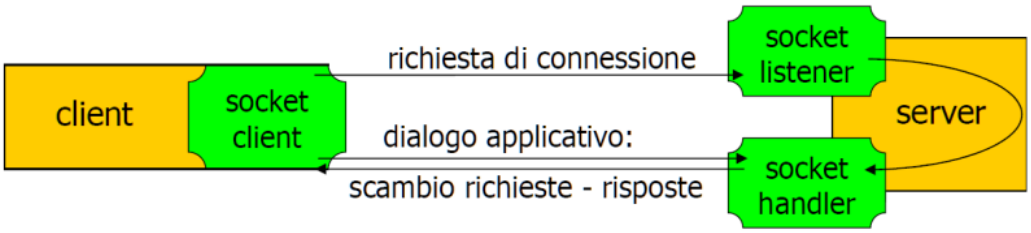
Il Socket. Carneade! Chi è costui?.

Un Socket è un'astrazione software progettata per poter utilizzare delle API Standard e condivise per la trasmissione e la ricezione di dati attraverso una rete. Ovvero, nel nostro caso, è il punto in cui un processo accede al canale di comunicazione per mezzo di una porta ottenendo un canale di comunicazione (*logico*) tra due macchine distinte. Ci sono due tipi fondamentali di socket:

- **Socket IP**: usati in molti sistemi operativi per le comunicazioni attraverso un protocollo di trasporto (*quali TCP/IP o UDP*).
- **Socket** in **Dominio Unix**: usati nei sistemi operativi POSIX per le comunicazioni tra processi residenti sullo stesso computer.

Inoltre troviamo anche due tipi di **Socket IP**:

- **Listener**: identificato dalla terna protocollo di trasporto, indirizzo IP e numero di porta permette l'accettazione di nuove connessioni in entrata da altre macchine in locale o attraverso una rete;
- **Established**: rappresentano una connessione attiva identificata dalla quintupla protocollo di trasporto, indirizzo IP sorgente, indirizzo IP destinazione, numero di porta sorgente, numero di porta destinazione.



Il Protocollo TCP / IP.

Il protocollo TCP / IP è, a volte, chiamato "**Suite di Protocolli TCP/IP**" in funzione dei due protocolli in essa definiti: il **Transmission Control Protocol** (*TCP*) e l'**Internet Protocol** (*IP*). Un indirizzo IP, nel particolare, mediante **4** (*IPv4*) o **6** (*IPv6*) **Gruppi** di numeri da 3 cifre (*Es: 192.168.1.23*) è in grado di identificare in modo univoco ogni scheda di rete connessa. In base a come vengono considerati questi gruppi di numeri (Host o Reti) possiamo definire delle Classi di IP: A, B, C, D, E.

Il TCP per l'invio dei pacchetti usa il meccanismo detto di **Windowed**, ovvero seguendo delle regole ben precise:

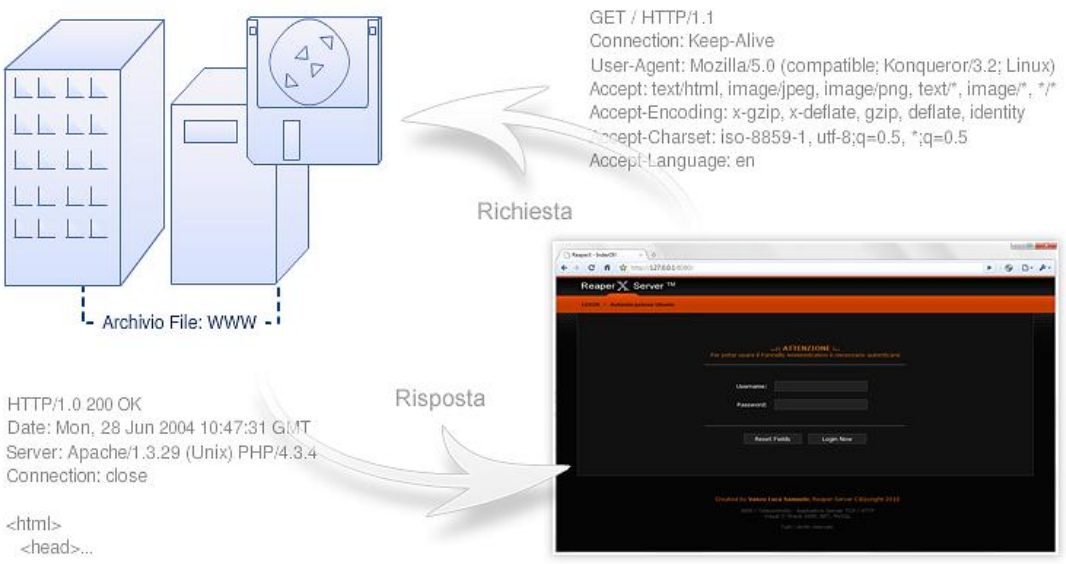
- Ad ogni pacchetto spedito il trasmettitore fa partire un **TimeOut**.
- Il Ricevitore invia per ogni pacchetto ricevuto un **ACK** indicando la sequenza dell'ultimo pacchetto ricevuto correttamente.
- Il trasmettitore considera quindi spediti tutti i pacchetti successivi.
- Se il timeout scade il TCP ritrasmette il pacchetto

Questa è una tecnica molto importante perché fornisce un canale di comunicazione affidabile includendo meccanismi per la congestione ed il controllo di flusso. Per esempio quando ci vogliamo collegare con il nostro browser a un **Server Web** (*Es Apache*), stabiliamo un collegamento (*Logico*) a livello applicativo il resto verrà gestito dal livello di trasporto e inferiori (*Stack ISO/OSI*).

Il Protocollo HTTP.

L'**Hypertext Transfer Protocol** (*HTTP*) è usato come principale sistema per la trasmissione di informazioni sul web.
Un **Server HTTP** generalmente resta in ascolto sulla porta (Es: 80 o 8080) usando il protocollo **TCP / IP** operando sullo stesso meccanismo botta e risposta di una rete **Client / Server**: il client (*Browser*) esegue una richiesta ed il server (*Apache per esempio*) restituisce una risposta. L'HTTP differisce da altri protocolli, come l'FTP, per il fatto che le connessioni vengono generalmente chiuse una volta che una particolare richiesta (*o una serie di richieste correlate: **HTTP 1.1***) è stata soddisfatta. Questo comportamento rende il protocollo HTTP ideale per la navigazione Internet (*Siti Web*) in cui le pagine, molto spesso, contengono dei **Link** ad altre pagine. Talvolta pone, però, problemi di sviluppo dei contenuti web poichè la sua natura **Senza Stato** (*Stateless*) costringe, per la conservazione dello stato utente, metodi alternativi sottili tramite linguaggi appositi (*PHP, ASP.NET*) o meno come i **Cookies**.

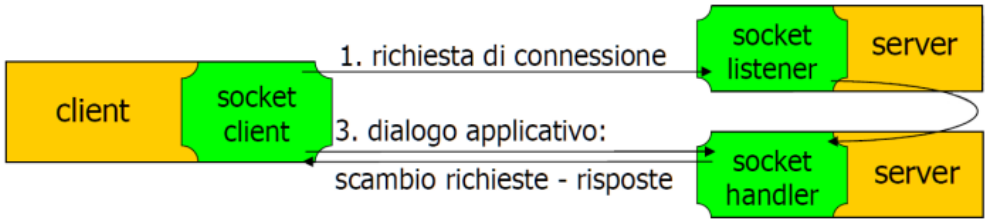
Un esempio pratico di **Richiesta** e **Risposta HTTP** lo si può avere visionando l'immagine seguente (*Pagina 10*).



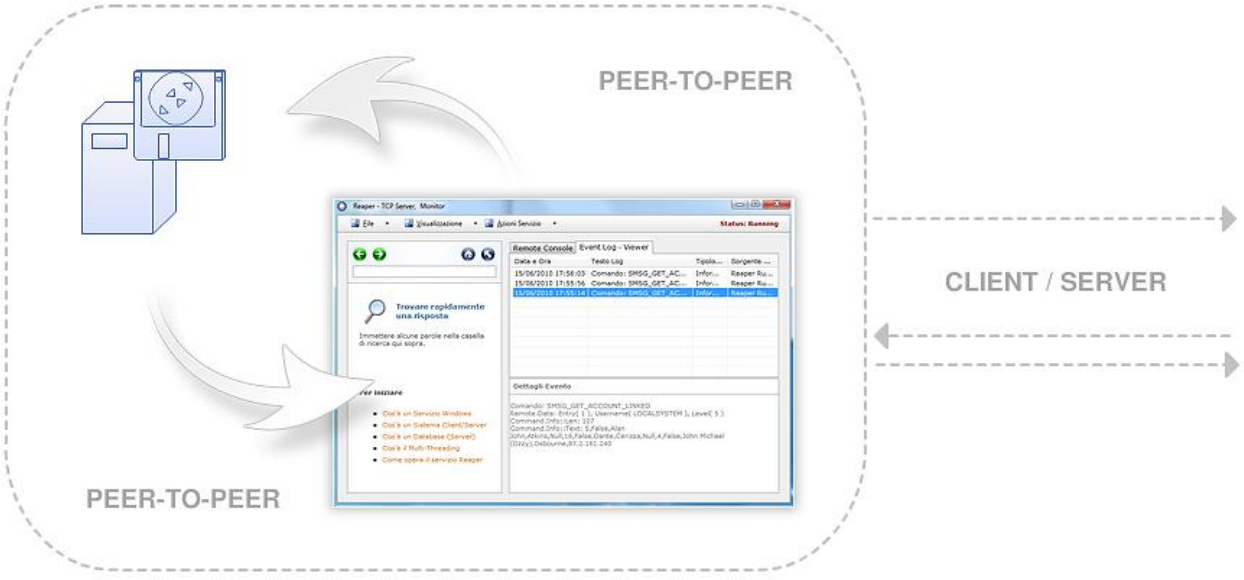
Struttura di un'applicazione Server Multi – Utenza.

Un server di base "**Client / Server**", come precedente illustrato nei **Socket**, ha un limite fondamentale: mentre è attivo il dialogo tra il server e un client il server stesso risulta inaccessibile ad altre richieste di connessione inoltrate da altri client. La soluzione a questo problema richiede un cambiamento radicale nella struttura che porterà a due applicazioni distinte:

- Un "**Server A**" in ascolto per richieste di connessione e che, per ognuna, ne attivi un'istanza nel "**Server B**".
- Un "**Server B**", infine, verrà gestito come un sotto-processo a esecuzione autonoma (**Thread**) dell'applicazione padre.



La struttura fin ora illustrata è la classica di un qual si voglia applicativo Client / Server che , però, può essere impiegata anche per creare applicativi in architettura **Peer-To-Peer** (*P2P*), in pratica introducendo nell'applicazione complessiva (*tratto grigio nella figura*) sia un client che un server (*vedi figura seguente*).



Solitamente quest'ultima opzione è usata, come nel mio caso, per monitorare attivamente (*come utente autenticato sul server*) il servizio offerto piuttosto che passivamente (*avvia, sospendi, spegni servizio*). In questo modo sarà possibile eseguire comandi come un qualunque utente registrato (*limitato al suo livello utente*), gestire le connessioni attive, i log eventi, il servizio stesso, ecc...

Un controllo per monitorarli tutti… Il Keep-Alive (Time-To-Live).

Ora compreso, si spera, la struttura di un **Server Multi – Utenza** sarà bene preoccuparci delle risorse a nostra disposizione: la Banda. Ebbene sì! L'idea di un'applicazione in grado di dialogare con più client contemporaneamente piace molto… un po meno se pensiamo che ogni Client occuperà una porzione di **Banda**, **RAM** e **Ciclo CPU**. Detto questo ci piacerebbe, quindi, sapere quali fra i tanti connessi sono degni di ricevere le nostre risorse, o meglio, di occuparle. Come fare??? Molto banalmente introduciamo un temporizzatore che ogni tot secondi controlli l'effettiva esistenza del Client al capo opposto della rete: il **Keep-Alive** (*Time-To-Live*).

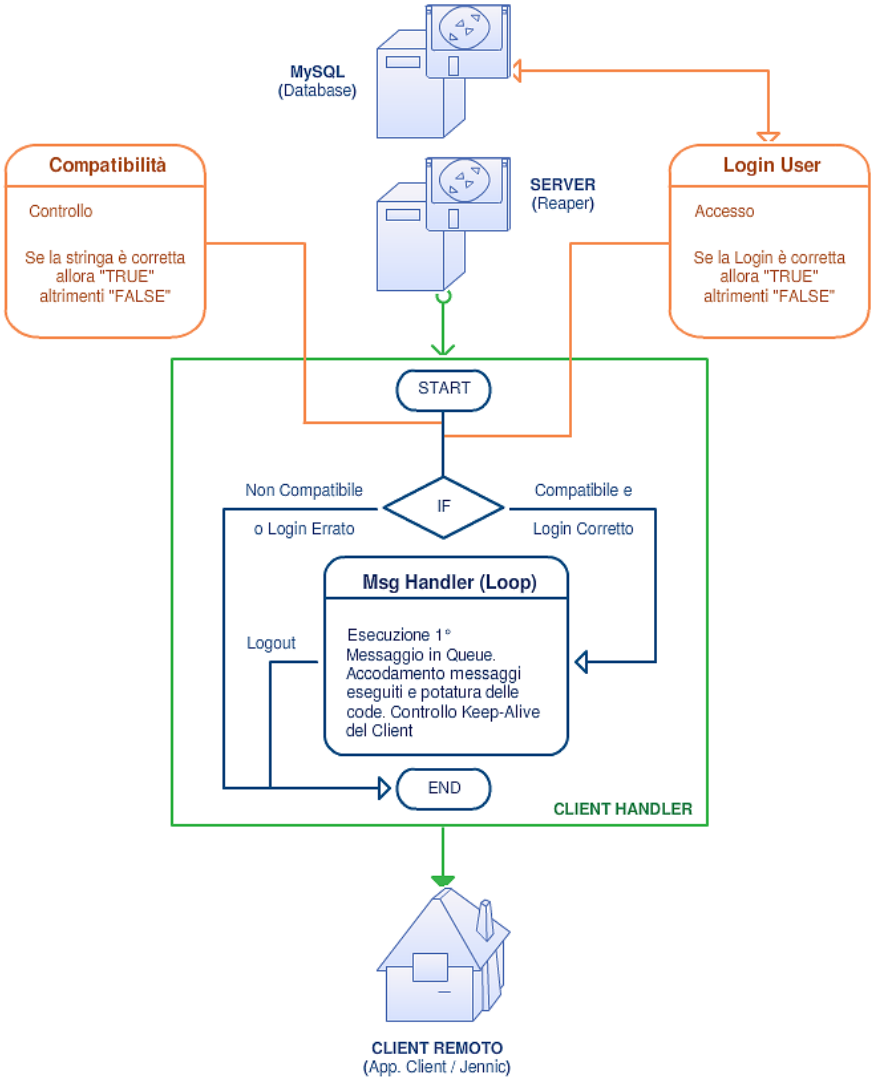
In sintesi si tratta di un **Timer** che, per esempio, dopo 30 secondi di inattività del client forza la chiusura della sessione (*logout del client, chiusura socket, pulizia della memoria usata*). Viceversa ogni comando eseguito dal Client provocherà il reset del Timer in quanto proverà l'esistenza di un'applicazione connessa al Server. Un disegno di massima dell'immagine complessiva del Client sul Server, compreso di **Keep-Alive**, risulterebbe quindi la seguente (*Operatore = Applicazione Server, Casa = Client*).

Struttura Server per Autenticazioni Multi-Utenza: Il Database.

Nelle pagine precedenti abbiamo visto come strutturare un'applicazione Server di modo che possa gestire più utenze contemporaneamente e nel caso, grazie al Keep-Alive, come potare quelle esistenti in base alla reale presenza o meno del Client.
Ora invece sorge il problema di come mantenere una serie di dati e informazioni associate a un Client in particolare.
Per fare ciò abbiamo bisogno di un secondo servizio server adibito a database: un **Relational DBMS**.
Nel nostro specifico caso useremo **MySQL** in quanto gratuito e **OpenSource** (*codice libero*).

Per prima cosa definiamo **RDBMS**. Un **Relational Database Management System** (*abbreviato in **RDBMS***) è un sistema software progettato per consentire la creazione e manipolazione efficiente di database (*ovvero di collezioni di dati strutturati*) solitamente da parte di più utenti contemporaneamente.

Detto questo possiamo apportare alcune semplici modifiche alle procedure di accettazione di un Client da parte del Server.
In sintesi si tratta di aggiungere, post **Accept()** della connessione, un controllo di **Login** magari preceduto da una stringa di validazione: giusto per essere sicuri che il Client in questione sia compatibile con i protocolli (*regole*) di comunicazione (*messaggi*) del Server in uso (*vedi immagine a blocchi seguente*).



Una funzione per limitarli tutti… Le Policy.

Ora che abbiamo implementato **MySQL** nella nostra applicazione ci piacerebbe renderla un pochino dinamica, magari che a livelli diversi di utenti corrispondano più o meno comandi secondo regole precise: le **Policy**. Nel particolare si creerà una coda di comandi che, durante l'esecuzione dei messaggi in **Queue** (*vedremo più avanti cosa sono*), compareremo per accertare se l'utente può o meno eseguire una data istruzione (*Es: SMSG_GET_ACCOUNT_PROFILE*) al suo livello attuale (*paziente, medico, admin, ecc…*).
Riprendendo la figura precedente, per esempio, potremmo trovare la suddetta funzione dentro il blocco "**Msg Handler (Loop)**".

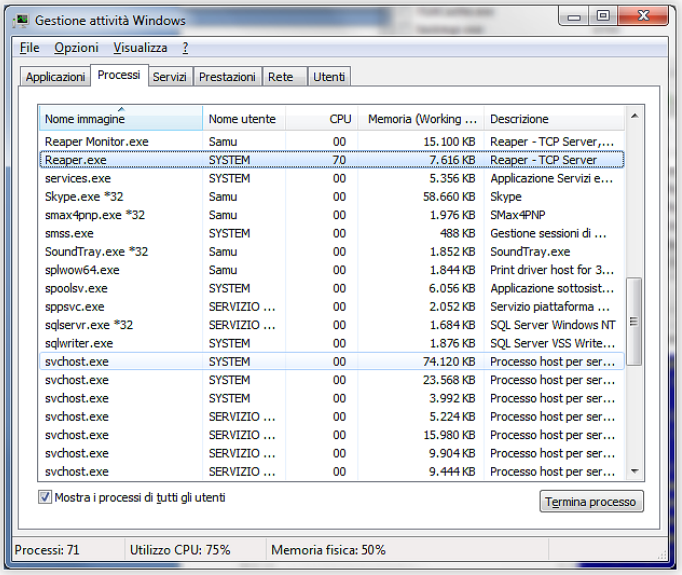
Il Multi-Threading.

Nelle puntate precedenti abbiamo introdotto i **Thread** come sotto-processi eseguiti parallelamente al processo padre.
Questi oggetti ci sono stati utili per ottenere un Server in grado di supportare più Client in contemporanea, ma come tutto hanno un lato negativo: il **Carico di Lavoro** (*o Cicli/s dei Core*). Infatti supponendo di avere un **Quad-Core** (4 Core di calcolo) e 4 Client connessi noteremmo **Carichi di Lavoro** prossimi al **80/90%** del totale della CPU. Questo è dovuto dalla presenza di **Loop Infiniti** che, per loro natura, essendo associati, per esempio, un **Core** ciascuno manterranno quest'ultimi occupati sino a sessioni terminate.

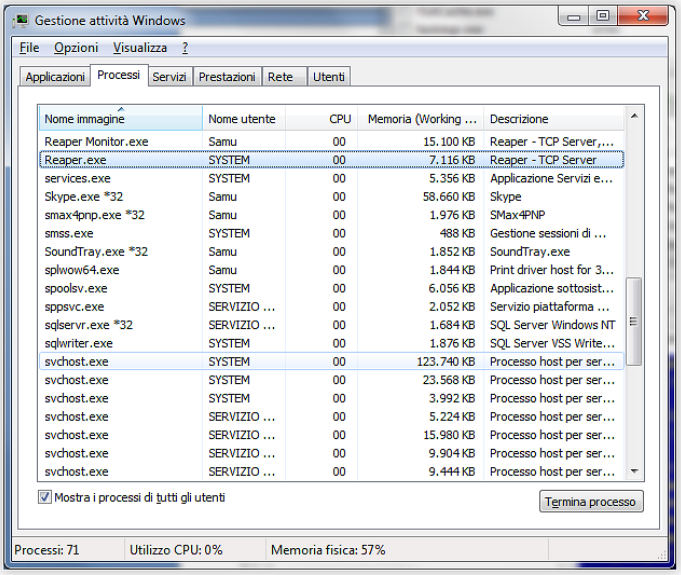
Seppur nativo in ormai tutte le CPU attuali il **Multi-Threading** (*ovvero la possibilità di eseguire più Thread in parallelo*) rimane uno strumento per i programmatori e resta a loro usarlo al meglio. Per questo motivo, ereditando dal programmazione d'intrattenimento (*Giochi*), introduciamo un piccolo espediente: la **Sleep()**. Mediante questa funzione infatti è possibile porre in stato di "**pausa**", per **1ms**, un **Thread** ad ogni nuovo ciclo in modo che la **CPU** possa eseguire altro più o meno associato al Server.

Questo trucchetto gioverà molto a livello di prestazioni poiché non solo avremo **Carichi di Lavoro** fissi al **0/1%,** ma avremo anche la possibilità di mantenere un maggior numero di Client connessi rispetto prima (*vedi immagine seguente*).

Multi-Threading **senza** System.Threading.Thread.Sleep(1);

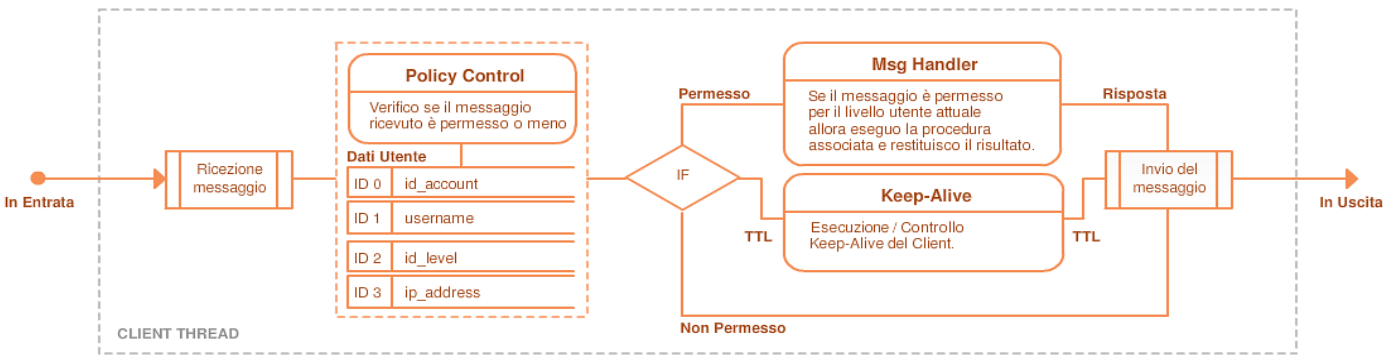


Multi-Threading **con** System.Threading.Thread.Sleep(1);



Logica di Queuing Asyncrono.

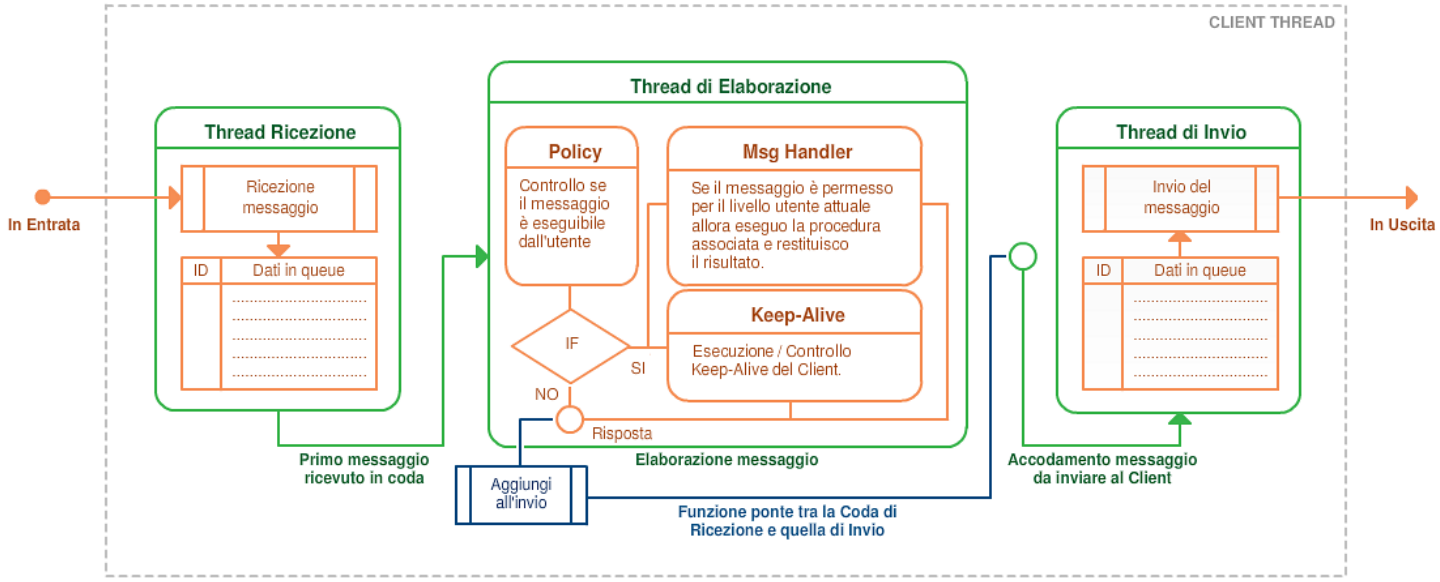
Un detto recita "*chi fa da sè fa per tre*", ma non sempre è così: la logica del **Queuing** che andrò ora spiegandovi ne è la prova.
Prima di partire, però, sarà meglio fare mente locale! Siamo partiti da un semplice Server Mono-Utente, abbiamo introdotto un secondo Server per poter accettare più Client, abbiamo implementato un **Database** per poter meglio gestire dati e informazioni di ogni utente e infine con la coppia **Multi-Threading** / **Keep-Alive** siamo riusciti a ottimizzare il tutto. Sempre sulla linea "*detti popolari*" abbiamo "*fatto 30… facciamo 31*". La struttura attuale, seppur buona e ben pensata, ha una pecca non trascurabile degna di essere approfondita. Prendiamo, per capirci, il seguente schema rappresentante la gestione di un messaggio inviato dal Client.



Come si evince dalla figura precedente il grosso problema che sorge da una struttura statica sono i tempi di esecuzione.
Infatti il ciclo di vita del **Thread Client** in uso è soggetto a tre fattori ritardanti: la **ricezione** (*varia con il timeout*), l'**esecuzione** della procedura (*se permesso*) ed infine l'**invio** (*varia con il timeout*). Per fare un esempio pratico si ipotizzi che invio e ricezione siano impostati con 5 secondi di **timeout** e che mediamente l'esecuzione impieghi un secondo per elaborare un risultato. Sulla base di questi dati la durata massima di un ciclo per singolo utente sarebbe di **11 secondi**. Ora, riprendendo il detto prima citato, nel nostro caso "*chi fa da sé **NON** fa per tre*"! Pertanto potremmo pensare di dividere i vari compiti di modo che nessuno influenzi l'esecuzione degli altri ottenendo, così, una struttura **Asyncrona** (*parallela*) a **3 Thread** perfettamente in sintonia fra loro: il **Queuing Asyncrono**.
Nel particolare quali vantaggi comporta l'accodamento?
Molto semplicemente una gestione a **Thread** paralleli risulta molto più efficace rispetto a un unico blocco poiché i tempi di ritardo di invio e ricezione non influenzeranno i tempi dell'esecuzione e viceversa. Cosa vuol dire tutto ciò. Riprendiamo l'esempio di prima.
Schematizzando l'esempio precedente con la nuova struttura avremo:

- Il Client invia 5 messaggi al secondo.
- L'elaborazione dura 1 secondo a messaggio
- L'invio è trascurabile in questa struttura poiché una coda distinta e indipendente.

Sulla base di questi dati notiamo un notevole vantaggio, quale? Mi pare ovvio!! Nel tempo in cui un messaggio viene elaborato, per poi essere spedito, il server contemporaneamente riceve ben 5 messaggi dal client, cosa che con la vecchia struttura non era nemmeno pensabile con conseguenti perdite di dati dal Client. Quindi con l'accodamento abbiamo risolto non uno, ma ben due problemi: il **Ritardo di Esecuzione** e la possibile **Perdita di Dati** (*vedi figura seguente per un'idea di funzionamento*).



La Garbage Collection.

Per **Garbage Collection** (*letteralmente raccolta dei rifiuti*) si intende una modalità autonoma di gestione della memoria, mediante la quale un sistema operativo o un compilatore, liberano le porzioni di memoria inutilizzate dalle applicazioni. In altre parole, il **Collector** annoterà le aree di memoria che non sono più referenziate (*allocate da un processo attivo*) e le libererà automaticamente.
Quali vantaggi comporta l'utilizzo della GC in termini di errori?

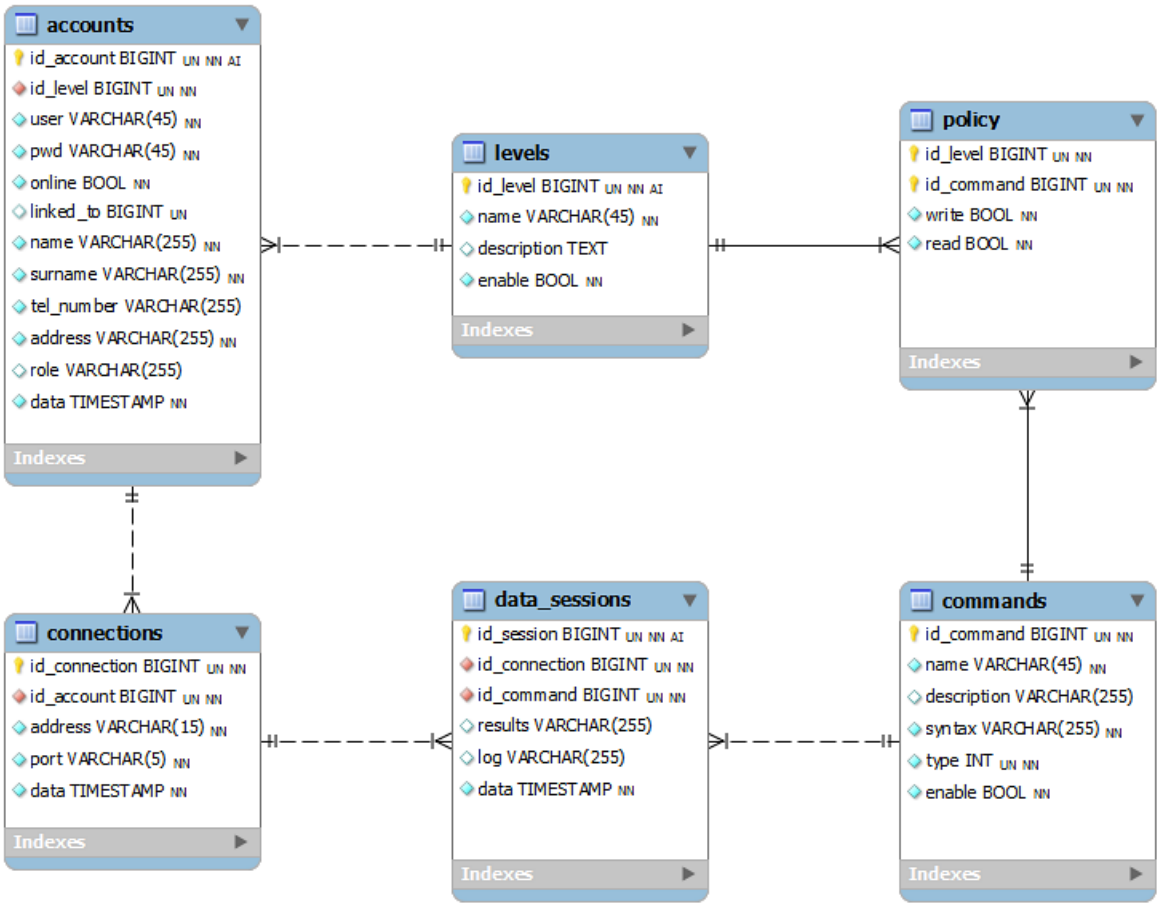
- **Null Reference**: si verifica quando, liberato uno spazio di memoria, viene richiamato un puntatore ad esso.
- **Double Bug-Free**: si verifica quando si tenta di liberare una zona di memoria già libera.
- **Dangling Pointer**: si verifica al rilascio erroneo di memoria ancora in uso.
- **Memory Leak**: si verifica in presenza di mancato rilascio di memoria non più utilizzata.

Specialmente quest'ultima categoria è fortemente nota nei linguaggi sprovvisti di un **Garbage Collector** automatico come ad esempio il **C** o il **C++**, mentre, è meno presente in linguaggi come il **Java**, il **C#** o **Lisp**(*origine dei GC*) anche se non ne sono immuni.
Nonostante l'avvento del **GC** rimane compito dello sviluppatore gestire e pulire la memoria prima e dopo l'utilizzo poiché seppur il **Collector** è in grado di recuperare memoria irraggiungibile (*inutile*) non è detto che possa recuperare memoria potenzialmente utile.

L'Enhanced ER-Diagram e le Tabelle risultanti.

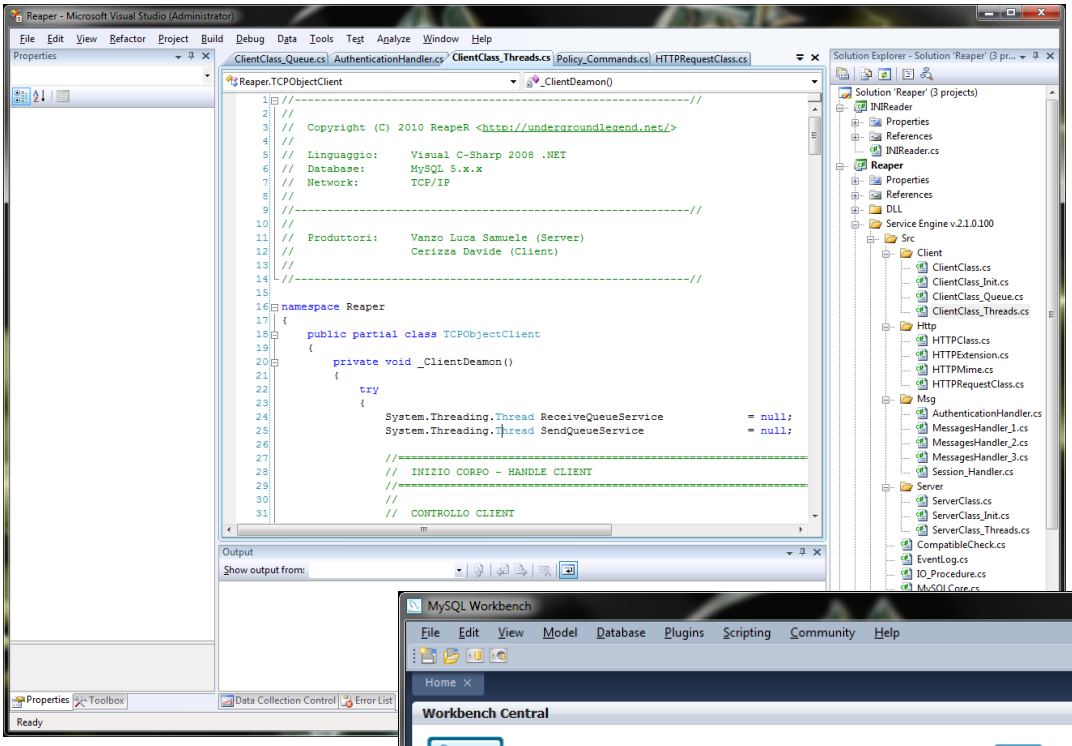
Per il **Database del Server** si è scelto di utilizzare 6 tabelle: **accounts** (*dati e profilo utenti*), **levels** (*livelli utente come medico, admin, paziente*), **policy** (*per stabilire quali livelli posso eseguire quali comandi*), **commands** (*lista dei comandi eseguibili*), **connections** (*informazioni aggiuntive dei singoli utenti come indirizzo IP e ora di connessione*), **data_sessions** (*storico dei comandi e risultati*).

Enhanced ER - Diagram

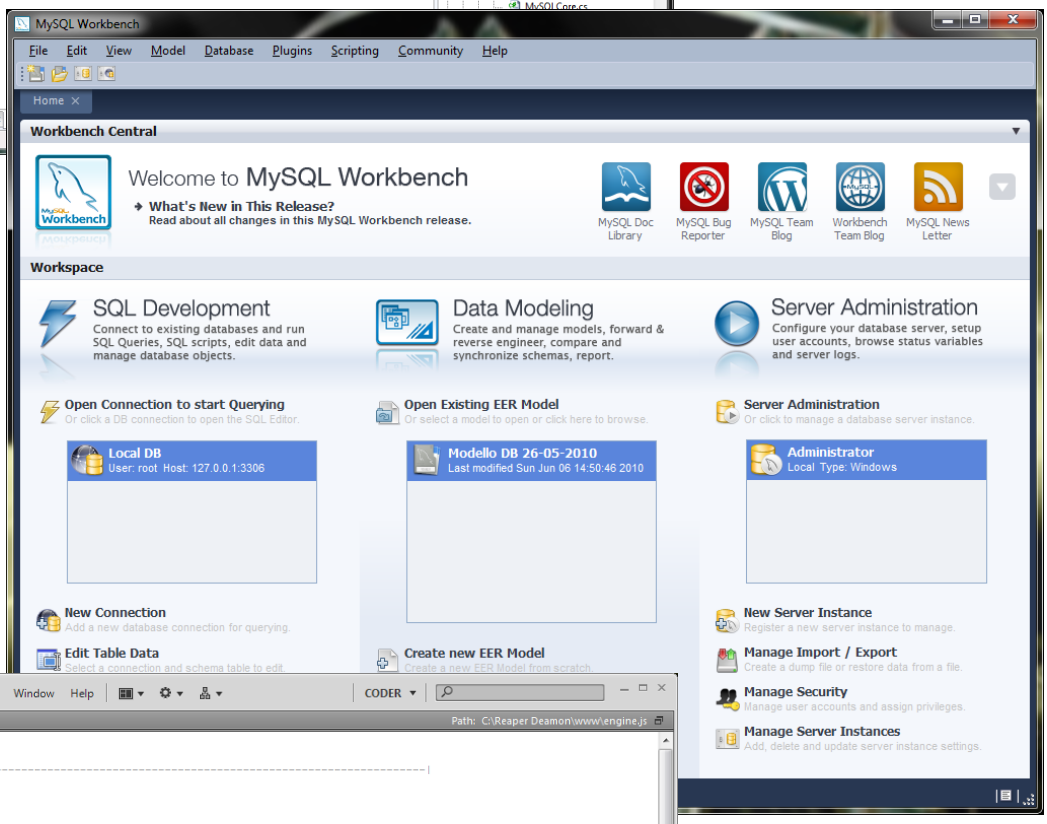


Commenti.

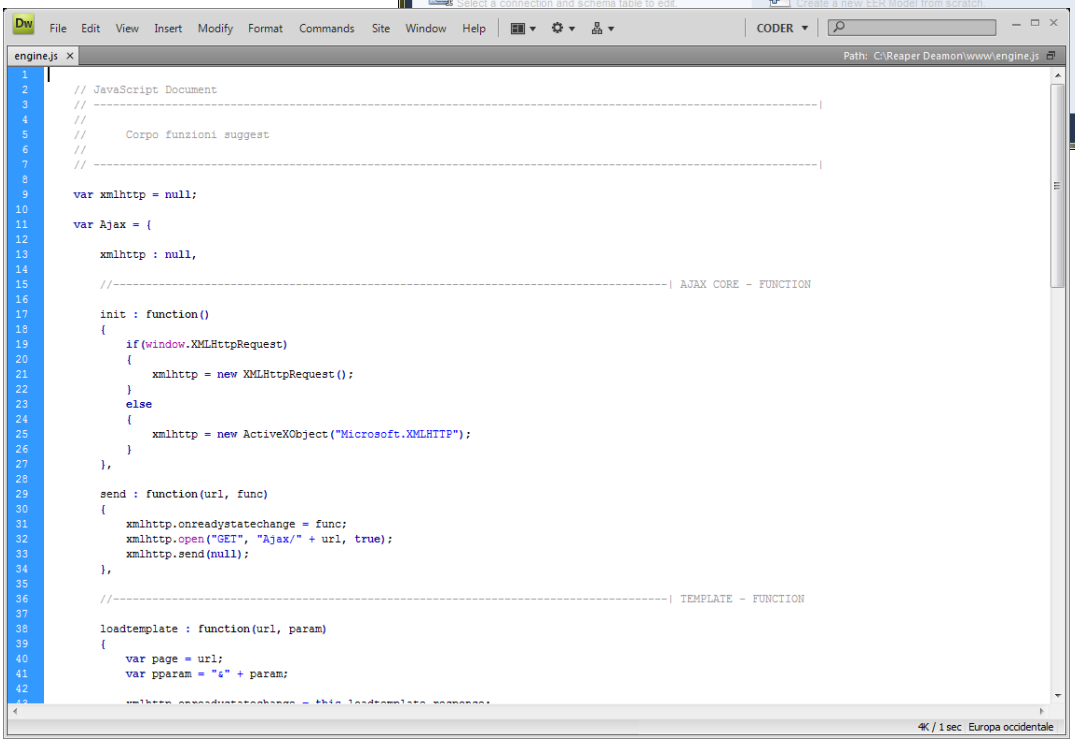
Dalla figura precedente si può notare, nonostante sia da evitare, vi sia la presenza di un **Loop** referenziale legato a tutte e 6 le tabelle. Questo perché a seconda del verso di percorrenza otteniamo un'informazione diversa. Per esempio attraversando il **Loop** partendo da **accounts**, passando per **levels**, verso **commands** sapremo quali utenti, legati a un certo livello, possono eseguire determinati comandi. Viceversa passando per **connections** sapremo quale utente, durante una determinata sessione, ha eseguito determinati comandi.



Visual Studio Team Edition 2008 Framework .NET



MySQL Workbench 5.2 + MySQL DBMS



Dreamweaver CS4 Editor