Programmazione di Sistema, a.a. 2015/2016

# Progetto d'esame: Documentazione

## Panoramica

Il progetto è composto di 3 componenti principali:

- Il **server**. Dato che deve fare molte chiamate alle API a basso livello di Windows, dev'essere una app Win32 "nativa". È realizzata come soluzione Visual Studio, e la sua compilazione è interamente gestita dall'IDE.
- Il **client.** È un'applicazione GUI scritta in C++ e portabile su una grande varietà di OS e architetture hardware. Fa largo uso delle librerie Qt.
- Il **protocollo di rete**. Si basa sulla libreria Protocol Buffers di Google.

Librerie utilizzate:

- Google Protocol Buffers (a.k.a. protobuf). Consente di definire formati dati strutturati, pensati per l'uso come formati di file o protocolli di rete. Funziona convertendo la rappresentazione del protocollo/formato, formulata in un apposito linguaggio, in codice sorgente che effettua serializzazione e parsing. La versione usata della *libreria* è 3.0.0 beta2 (Release GitHub), ma la versione del *linguaggio* è la 2. La nuova versione 3 del linguaggio era ancora in fase alpha al momento della creazione del nostro progetto, e abbiamo seguito le raccomandazioni sul sito ufficiale della libreria.
- **Qt**. (Solo per il client.). Si tratta di una libreria estremamente completa, versatile, portabile e performante per la costruzione di ogni genere di applicazioni. Noi usiamo il modulo per la costruzione di GUI e quello per il networking. La scelta di Qt è motivata principalmente dalla familiarità che uno degli autori ha con essa. La portabilità della nostra applicazione è dovuta proprio a Qt, ed è ulteriormente facilitata da strumenti da essa forniti quali QMake, che gestisce l'intero processo di compilazione.

Per la compilazione e lo sviluppo abbiamo utilizzato alcuni ulteriori strumenti, che quindi costituiscono dipendenze di sviluppo (ma sono totalmente superflui a runtime):

- **Ruby**. Abbiamo scritto in Ruby un generatore di codice per la conversione delle combinazioni di tasti fra Qt, API di Windows e il nostro protocollo di rete. Abbiamo scelto Ruby per approfittare dei moduli YAML e ERB della libreria standard. Nessun'altra libreria Ruby è necessaria.
- **CMake**. È necessario per compilare Protobuf su tutte le piattaforme supportate.

## Note di piattaforma

Dato che ogni programma va compilato necessariamente con lo stesso compilatore e le stesse librerie "di base" (C Runtime, o CRT) di tutte le librerie che esso usa, è stato necessario organizzare alcuni dettagli del processo di compilazione del progetto.
Il compilatore scelto è **MSVC 19.0**, semplicemente perché già parte di Visual Studio 14.0 2015, comunque usato per lo sviluppo del server. Riguardo la libreria CRT, siamo vincolati

da Qt, che va compilata necessariamente con la versione del CRT **Multithreaded Debug/Release DLL** [1]. Inoltre, da MSVS 2015/MSVC 19.0 in poi, il CRT è stato riorganizzato. Ora è necessario linkare e aggiungere l'include path dello **Universal CRT**, che è separato dalle librerie incluse con VS. Visual Studio fa queste operazioni in automatico, senza nessuna novità dal punto di vista dello sviluppatore, mentre invece gli strumenti offerti da Qt (Qt Creator e qmake) sono stati necessariamente configurati esplicitamente, includendo alcune direttive nel file .pro.
Come la maggior parte delle app desktop su Windows (come Chrome, Firefox, Visual Studio, …), stiamo compiliando tutto in **32 bit**. Questo perché i vantaggi dati dal compilare in x64 sono irrilevanti per i nostri scopi, mentre compilando in 32 bit otteniamo un uso di RAM e disco leggermente più ridotto, e maggior compatibilità con macchine datate. La conseguenza di questa scelta è che tutte le librerie devono essere in versione 32 bit.

# Protocollo

Il protocollo è abbastanza semplice, specificato dai due file *protocol.proto* e *keys.proto*. I file C++ generati a partire da essi sono linkati o inclusi sia nel progetto del server che in quello del client.

## Codifica delle icone

Ogni icona viene serializzata in un messaggio di tipo *Icon*, dove se ne specifica la dimensione (campi *width*, *height*) e il contenuto. L'immagine è serializzata in un array di byte nel campo *pixels*, in un semplice formato in cui i pixel dell'immagine occupano 4 byte ognuno in formato BGRA8 (il valore dei canali blu, verde, rosso e alpha, in quest'ordine, un intero unsigned di un byte per canale), e sono disposti come in una matrice in ordine row-major (ogni riga da sinistra a destra; righe da quella in cima verso sotto).

## Codifica delle combinazioni di tasti

Ci sono tre diverse codifiche usate nel progetto per le combinazione di tasti: quella usata da Qt per rappresentare l'input dell'utente del client; quella usata dal protocollo di rete per la serializzazione/deserializzazione; quella compresa dalle API di Windows, che il server usa per mandare comandi alle applicazioni.
Il file *keycodes.yaml* contiene la tabella di traduzione dei keycode fra le tre diverse codifiche. Tutti i componenti che devono usare i keycode dovrebbero usare le corrispondenze specificate in questo file. Dato che la codifica di Windows e di Qt consiste in costanti simboliche del codice sorgente (costanti VK_*, costanti Qt::Key), si è deciso di basarsi sulla generazione di codice sorgente.
Ad esempio, il file *keys.proto* contiene la codifica dei tasti usata nel protocollo, ma è generato dallo script *keys.proto.erb* mediante *erb* (Embedded Ruby).

---

[1] Vedi: http://stackoverflow.com/questions/25491916/why-does-qmake-select-the-multi-threaded-debug-dll-mdd-run-time-library-for

## Flusso di una sessione tipica

1. Il client si connette al server; il server manda immediatamente un messaggio *AppList*, contenente la lista delle applicazioni attive al momento della connessione nella forma di una sequenza di messaggi *Application*.
2. All'accadere di un evento (finestra distrutta, finestra creata, cambiamento della finestra in focus, ...) il server manda un messaggio di tipo *Event.*
3. Quando l'utente richiede di mandare la pressione di una combinazione di tasti all'applicazione in focus, il client codifica la richiesta come un messaggio *KeystrokeRequest* e la manda al server. Il messaggio è marcato con un identificatore numerico. Il server risponde con un messaggio *Event* contenente un campo di tipo *Response* marcato con lo stesso identificatore numerico della richiesta ed un codice d'errore/successo.

# Server

## Panoramica dell'architettura

Il problema principale da risolvere nell'implementazione del programma è quello di ricevere gli eventi dalle API di Windows e distribuirli in modo concorrente ai client. La soluzione scelta si basa sul *polling*, che consiste nell'interrogare le API di Windows a intervalli regolari e ogni volta confrontare la lista delle finestre ottenuta con quella ottenuta dall'interrogazione precedente, per ricavare l'insieme delle finestre create e distrutte durante l'ultimo intervallo. Quindi gli eventi appropriati vengono generati e mandati ai client.

Il programma è multithreaded. In particolare:

1. un thread è dedicato all'attesa di nuovi client;
2. per ogni connessione, un thread dedicato si occupa di mandare la lista iniziale delle finestre, per poi passare a servire le richieste provenienti da quel client (compresa la chiamata delle API di Windows necessarie a generare l'evento sulla finestra in focus);
3. un thread si occupa del monitoraggio delle finestre: chiama periodicamente EnumWindows, ottiene gli insiemi di finestre create e distrutte, manda gli eventi ai client.

I thread del gruppo 2 sono creati in numero fisso, e ognuno lavora in un ciclo infinito che inizia prelevando un socket connesso da una coda FIFO apposita (riempita dal thread (1)), per poi servirne le richieste fino alla sua chiusura, e infine ripetere il ciclo prelevando un nuovo client. Per questo motivo, il numero di thread nel gruppo 2 limita il numero di client contemporaneamente connessi che possono essere serviti. Inoltre, si noti che ogni socket è letto un thread del gruppo (2) che serve le richieste (combinazioni di tastiera) e letto dal thread (3) che trasmette gli eventi. Il thread (3) manda gli eventi su tutti i socket dei client connessi.

### Alternativa scartata: hooks

Una soluzione alternativa è quella di servirsi degli *hooks*. Si tratta di un meccanismo offerto dalle API di Windows in cui un processo può provocare il caricamento di una DLL nello spazio di indirizzamento di uno o più altri processi in modo che al verificarsi di un determinato evento ogni processo chiami una procedura handler presente nella DLL "iniettata".
La soluzione è stata ben testata realizzando degli esperimenti concreti. In particolare, il nostro programma di test chiama *SetWindowsHookEx* per installare una procedura handler sull'hook WH_SHELL (è un hook globale, e quindi cattura gli eventi di tutti i processi del desktop). La procedura handler della DLL poi usa una Mailslot per comunicare l'evento al nostro programma server.
Il vantaggio di questo sistema è che ogni evento viene catturato immediatamente quando si verifica, ma hanno anche molti svantaggi. Innanzitutto, in generale, presentano molti evidenti problemi di sicurezza: possono facilmente indurre il crash o addirittura corrompere lo spazio di indirizzamento di applicazioni altrimenti correttamente funzionanti. Inoltre, per usarli correttamente bisogna fare grande attenzione alla differenza di bitness (32- o 64-bit) tra la DLL da installare e il processo "agganciato". La documentazione dice che in caso di bitness diverse fra DLL hook e processo vittima, si installa la DLL sul processo agganciante mentre sul processo agganciato viene iniettata una callback. Tuttavia, dalla nostra esperienza emerge che questo meccanismo non funziona nel nostro caso (almeno non senza ulteriori accorgimenti da prendere, che tuttavia non sono documentati). L'unico workaround documentato, e di cui abbiamo potuto verificare l'efficacia, è quello di compilare due diverse varianti del processo agganciante e della DLL hook, a 32 e 64 bit rispettivamente, ognuno responsabile di agganciare processi della propria bitness. Nel nostro caso, bisogna anche risolvere la questione di adottare un protocollo di IPC per far comunicare la DLL iniettata con il processo del server.
Abbiamo deciso che il grande aumento di complessità portato da questa soluzione supera di gran lunga i vantaggi, e abbiamo quindi optato per una più semplice e sufficientemente efficace soluzione basata su polling.

## Principali unità

**ProcessWindow**. Rappresenta una singola finestra/applicazione. Nel suo costruttore, estrae e immagazzina le informazioni relativa alla finestra (titolo, icona, ecc…) attraverso le API Win32 a partire dall'HANDLE corrispondente.
**WindowsList**. Una collezione thread-safe di handle a finestre (HWND). Offre il metodo *update* che imposta l'insieme delle finestre contenute, e manda le notifiche appropriate (*nuova finestra*, *finestra chiusa*, ...) ai client connessi.
**Client**. Espone un'interfaccia thread-safe ad un socket corrispondente a un client connesso. Offre metodi per inviare/leggere messaggi e chiudere il socket. In particolare, le scritture sono protette da un mutex.
**ClientList**. Una coda FIFO thread-safe contente oggetti Client. Il programma ne mantiene due: la coda dei client *pending*, in attesa di un thread che ne serva le richieste; la coda dei client *active*, i quali devono ricevere gli eventi.

# Le soluzione scelte: vantaggi, limitazioni, compromessi

## Iniezione di input da tastiera

Dai client provengono richieste di mandare una data combinazione di tasti alla finestra in focus. Oltre agli errori segnalati e/o gestiti direttamente dalle API di Windows, l'unica condizione problematica da gestire è quella in cui il focus è cambiato dal momento in cui la richiesta è stata mandata a quella in cui è stata ricevuta e processata: in altre parole, bisogna assicurarsi di non mandare la combinazione di tastiera ad applicazioni oltre quella per cui era originariamente intesa.
Le soluzioni testate e valutate sono: mandare direttamente il messaggio al processo interessato (API *PostMessage*); forzare il focus verso la finestra che l'utente remoto vuole controllare, e poi mandare l'input alla finestra con focus usando *SendInput*; rilevare e segnalare al client il cambio di focus, lasciando all'utente la facoltà di ripetere il comando. Le prime due soluzioni si sono rivelate non particolarmente complesse ma comunque non sufficientemente efficaci: la prima soluzione non funzionain tutti i casi [2], mentre la seconda soluzione non elimina la race condition, ma ne riduce solo le probabilità.

Rappresentare le combinazioni di tasti e mandarle a un applicazione è, nella forma più generale, un problema estremamente complesso, a causa di varie race condition e casi particolari. Ciò è dimostrato, ad esempio, dal codice sorgente di AutoHotKey, un programma per automatizzare l'input dato alle applicazioni di Windows e che pertanto deve risolvere il nostro stesso problema [3].

## Elencare le finestre/applicazioni aperte

La API da usare è *EnumWindows*, che però ritorna più finestre di quelle interessanti per la nostra applicazione. Dopo qualche ricerca, abbiamo deciso per un'euristica che esclude finestre invisibili, finestre che hanno per titolo una stringa vuota, finestre di strumenti (ad es. floating toolbar) e finestre che non hanno la finestra radice come "genitore" (*parent*) diretto o indiretto.

[2] https://blogs.msdn.microsoft.com/oldnewthing/20050530-11/?p=35513/
[3] https://github.com/Lexikos/AutoHotkey_L/blob/master/source/keyboard_mouse.cpp