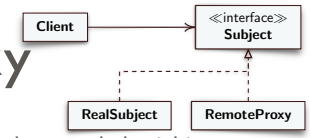
Remote Proxy

[Buschmann]

- Contesto: nella variante Remote Proxy, il client necessita di accedere a servizi di un componente che risiede su un altro host
- Problema: l'accesso diretto da parte del client all'oggetto remoto non è appropriato. I client non dovrebbero conoscere gli indirizzi di rete ed i protocolli di comunicazione inter-processo (IPC)
  - Inoltre, l'accesso al servizio deve essere trasparente e semplice per il client. Il client non deve cambiare la sintassi usata per le chiamate al servizio, sebbene sia in remoto
  - Il client dovrebbe sapere di possibili penalità nelle prestazioni, a causa di tempi di accesso più elevati. Piena trasparenza può oscurare le differenze di prestazioni fra servizi
    - Una comunicazione da un host a un altro è molto più lenta di una chiamata di metodo nello stesso host: almeno **20.000 ns** contro (anziché) **1 ns** (in base a distanze, velocità rete, tecnologie)

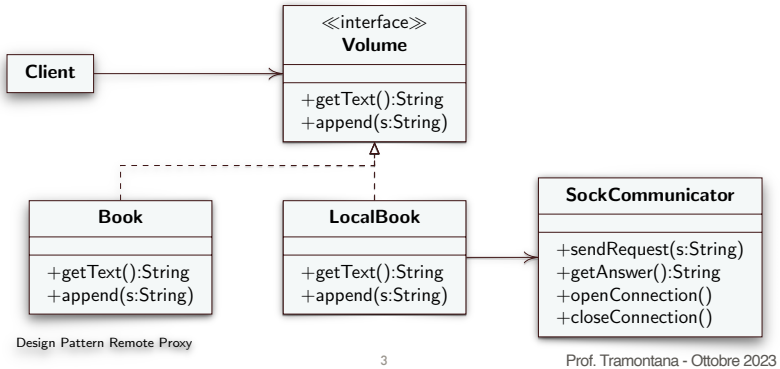
Remote Proxy



- Soluzione: il client comunica con il **RemoteProxy**, che manda le richieste ad un host remoto, dove il RealSubject risiede, e riceve risposte da remoto
  - Il **RemoteProxy** tiene la posizione (indirizzo IP, porta) ed identità del RealSubject, ed implementa la comunicazione fra le due macchine e i due processi (per es. usa le socket per comunicare). Per ciascun RealSubject, vi è un RemoteProxy in ciascun host in cui occorre usare i servizi del RealSubject
  - Per meccanismi IPC complessi si può usare un componente apposito lato client, e analogamente un componente ricevente per il lato RealSubject. I client non sono a conoscenza dei dettagli della comunicazione remota
  - Il RemoteProxy può far comunicare due processi presenti sullo stesso host o su host diversi
  - Il RemoteProxy rende più facile analizzare le prestazioni (e i ritardi) della comunicazione in rete

Remote Proxy

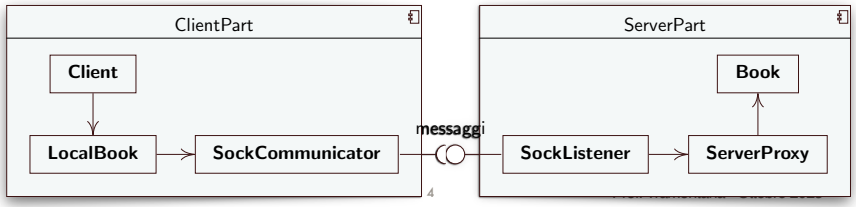
- Per l'applicazione di esempio
  - Volume è il ruolo **Subject**
  - Book è il ruolo **RealSubject**
  - LocalBook è il ruolo **RemoteProxy**



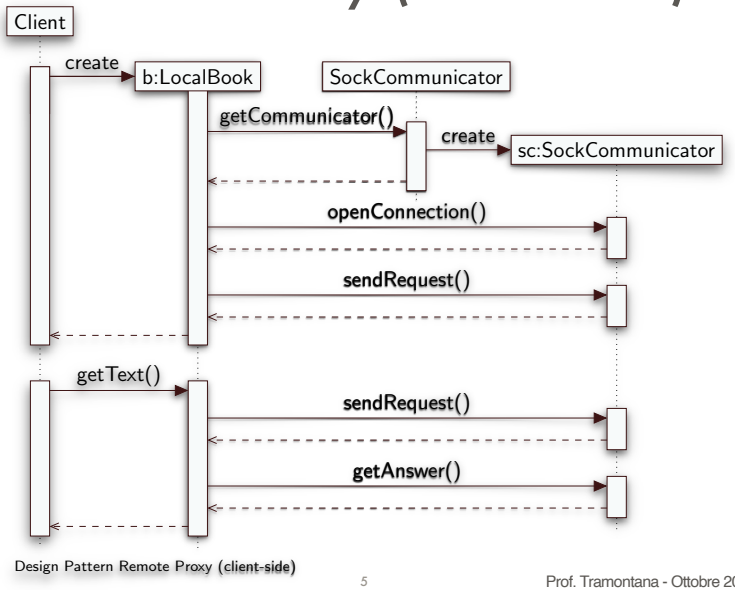
Design Pattern Remote Proxy

Remote Proxy

- Soluzione: diagramma UML delle componenti
  - **RemoteProxy** (LocalBook) traduce chiamate a metodi in messaggi, inviati tramite SockCommunicator, e implementa il protocollo di comunicazione con l'altra parte
  - **SockCommunicator** fornisce un'astrazione per la comunicazione (tramite Socket), semplificando il RemoteProxy e permette di cambiare facilmente la tecnologia di comunicazione [vedi dopo]
  - **SockListener** viene avviato sull'host server per mettersi in ascolto di possibili messaggi che arrivano dalla rete (da uno o più host remoti)
  - **ServerProxy** traduce i messaggi appena arrivati lato server in chiamate a Book

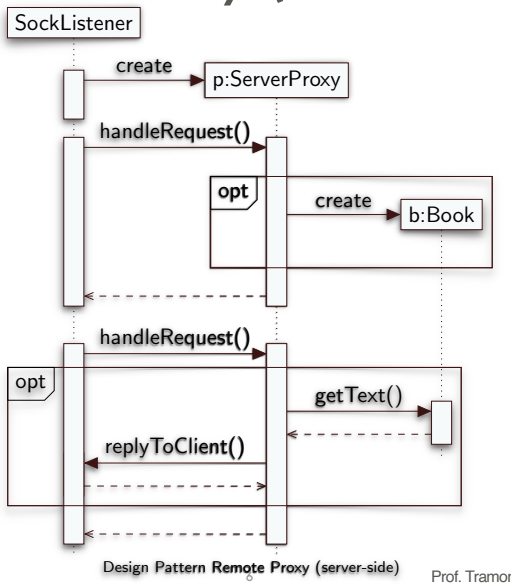


Remote Proxy (Client-Side)



Design Pattern Remote Proxy (client-side)

Remote Proxy (Server-Side)



Design Pattern Remote Proxy (server-side)

Forwarder-Receiver

[Buschmann]

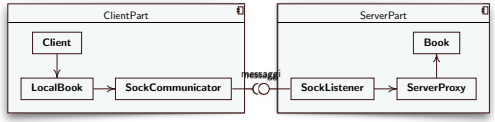
- Il design pattern Forwarder-Receiver fornisce comunicazione trasparente fra processi per sistemi che interagiscono tramite il modello peer-to-peer. Introduce *forwarder* (spedizionieri) e *ricevitori* per disaccoppiare i peer (ovvero le classi client) dai meccanismi di comunicazione sottostante
- Problema: nello sviluppo di applicazioni distribuite si usano meccanismi di comunicazione fra processi come TCP/IP, socket, code di messaggi. Questi sono efficienti ma introducono dipendenze con il sistema operativo ed i protocolli di rete
- Le forze che si vogliono bilanciare
  - Il sistema dovrebbe consentire di cambiare i meccanismi di comunicazione
  - La cooperazione segue il modello peer-to-peer, in cui il mittente deve solo conoscere il nome dei riceventi
  - La comunicazione fra peer non dovrebbe avere un notevole impatto sulle performance

Forwarder-Receiver

- Soluzione: i peer collaborano ed un peer può agire da client (richiede un servizio), o da server (fornisce un servizio), o entrambi. I dettagli del meccanismo di comunicazione sottostante sono nascosti ai peer, ed incapsulati in componenti separati
  - Funzionalità specifiche da incapsulare separatamente sono: mappatura fra nomi e locazioni specifiche, instaurare il canale di comunicazione, inviare e ricevere messaggi
- Struttura: il design pattern consiste dei ruoli **Forwarder**, **Receiver**, **Peer**
- I **Peer** sono responsabili dei compiti dell'applicazione e inviano messaggi ad altri peer o ricevono messaggi da altri peer
- **Forwarder** invia messaggi verso altri processi e fornisce un particolare meccanismo di comunicazione. Contiene il mapping fra nomi e indirizzi fisici. Nel messaggio trasmesso include il nome del proprio peer, così che il peer remoto possa inviare una risposta ad esso

Forwarder-Receiver

- Soluzione per l'applicazione mostrata, ruoli svolti dalle classi per il pattern
  - `LocalBook` (RemoteProxy) e `ServerProxy` sono **Peer** (classi che devono comunicare fra loro)
  - `SockCommunicator` è un **Forwarder**
  - `SockListener` è un **Receiver**



- Considerazioni
  - Usando entrambi i pattern Remote Proxy e Forwarder-Receiver si disaccoppiano meglio client, proxy e meccanismi di comunicazione
  - Per il pattern Remote Proxy, il RemoteProxy deve tenere la locazione del RealSubject. Mentre, per il pattern Forwarder-Receiver, il Peer deve conoscere il nome del Peer destinatario, non la posizione
    - L'implementazione del RemoteProxy può soddisfare solo uno dei vincoli suddetti, sarebbe meglio far conoscere solo il nome (identità) del Peer

Forwarder-Receiver

- Ci sono diversi tipi di messaggi
  - Messaggi comandi che istruiscono il ricevente per fare alcune attività
  - Messaggi informativi che contengono dati sulla rete di risorse e gli eventi di rete
  - Messaggi di risposta che permettono di mandare una ricevuta (ack) per un messaggio ricevuto
- I **Receiver** sono responsabili di ricevere messaggi. Un receiver offre una interfaccia per uno specifico meccanismo di comunicazione IPC
  - I receiver convertono lo stream di dati ricevuti in un formato di messaggio che inviano al processo per il quale agiscono