



POLITECNICO
MILANO 1863

Network

9 Aprile 2019

Scaglione San Pietro

RESPONSABILI

Giovanni Meroni
Amarildo Likmeta

TUTOR

Marco Bacis
Valentina Deda

RMI

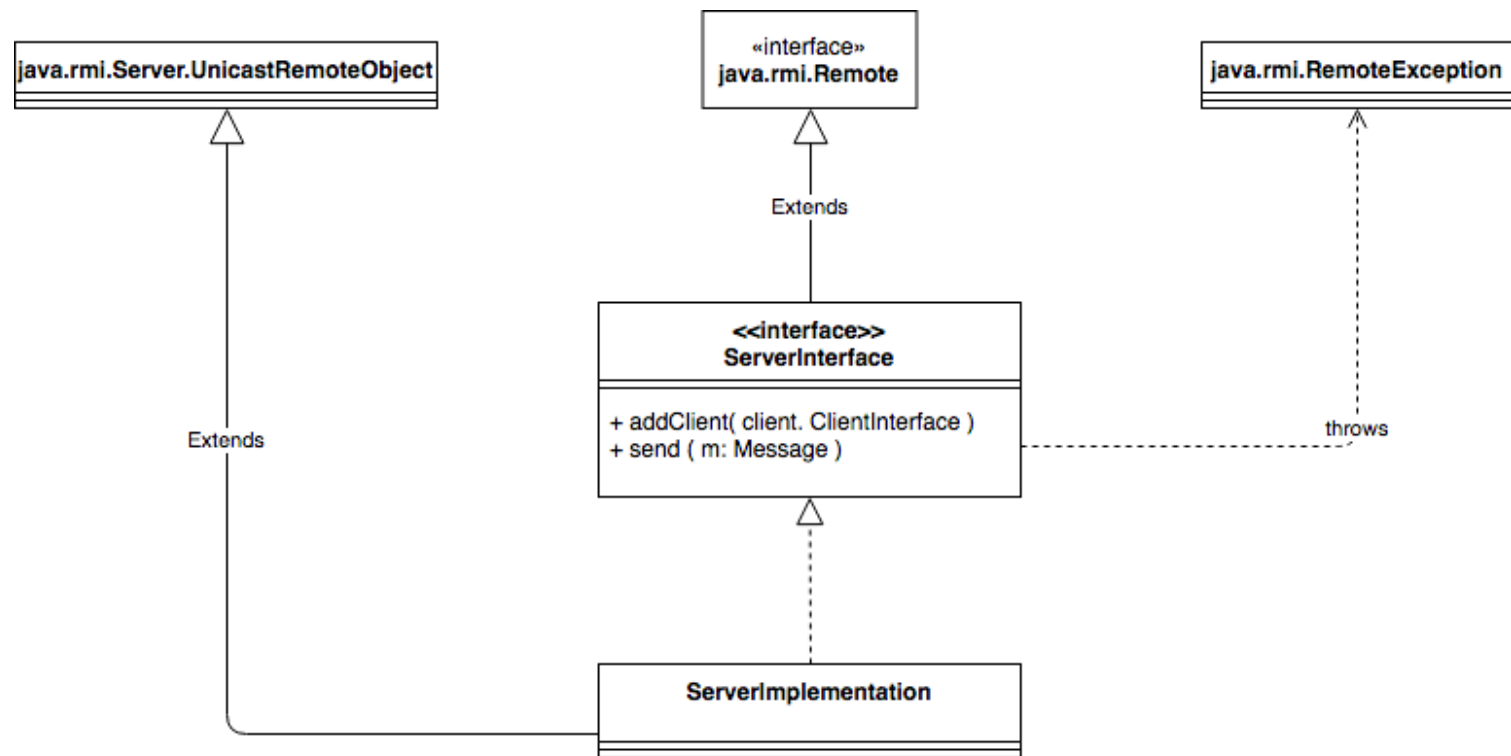
Remote Method Invocation (RMI)

- Middleware per applicare la programmazione orientata agli oggetti in un contesto distribuito
- Oggetto remoto: oggetto il cui riferimento è disponibile su una JVM diversa da quella in cui risiede l'oggetto.
- Passaggio di parametri:
 - per **valore** (una copia) per **oggetti non remoti**
 - per **indirizzo** (il riferimento remoto) per **oggetti remoti**
- Possibilità di scaricare automaticamente le classi necessarie per la valutazione remota

Lato Server

- Il server vuole offrire un servizio al client. Il servizio è rappresentato da un oggetto remoto (**remote object**, *ServerImplementation* nell'esempio) che implementa una interfaccia remota (**remote interface**, *ServerInterface* nell'esempio).
- L'interfaccia remota:
 - Deve essere pubblica
 - estende ***java.rmi.Remote***
 - tutti i suoi metodi lanciano l'eccezione ***java.rmi.RemoteException***

Lato Server



Lato Server

- Il Server crea un riferimento remoto al remote object:

- O utilizzando il metodo

`UnicastRemoteObject.exportObject(obj, port)`

- O facendo estendere all'oggetto remoto

`java.rmi.server.UnicastRemoteObject` (come da esempio)

- Infine rende il servizio disponibile ai client registrandolo tramite RMI

Registry:

- `Naming.rebind("//localhost/Server", remoteObjectRef)`

Lato Client

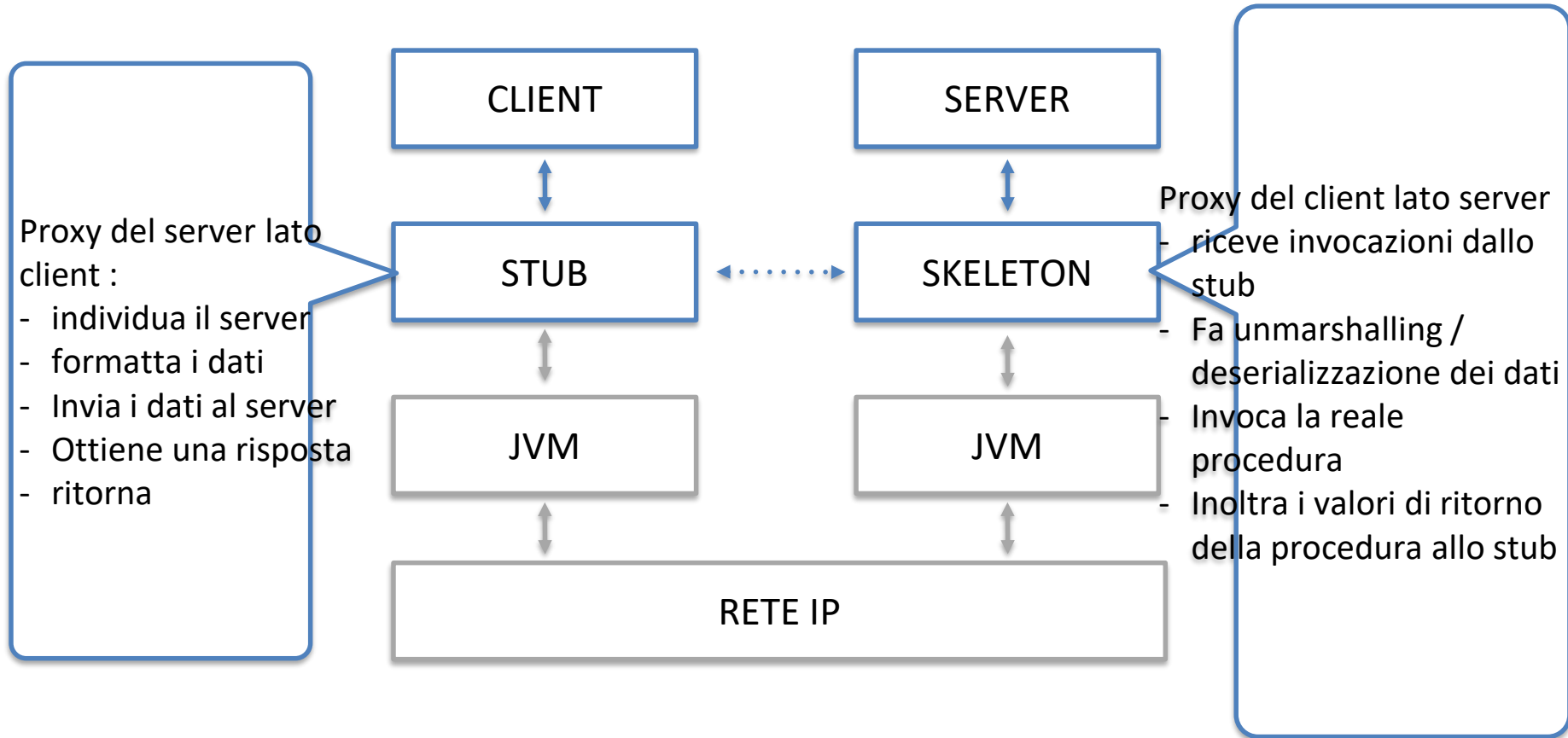
- Il client chiede a RMI Registry un riferimento all'oggetto remoto:

```
SeverInterface server =  
(ServerInterface) Naming.lookup( "localhost/Server" )
```

A questo punto il client può invocare i metodi dell'oggetto remoto come se fosse un normale oggetto locale, ma:

- Gli oggetti remoti che passo si comportano come normali oggetti
- Gli oggetti non remoti che passo devono implementare ***java.io.Serializable*** e quello che passo è in realtà una copia (quindi le modifiche che faccio lato server hanno effetto solo sulla copia)

Architettura RMI



A decorative horizontal band at the top of the slide, consisting of a solid teal bar above a series of thin, vertical white lines of varying heights.

Demo

Socket

Socket

- Permettono di trattare la comunicazione di rete allo stesso modo con cui è possibile trattare la lettura/scrittura di un file
- La classe *java.net.Socket* rappresenta la connessione stabilita
- Il Socket è identificato dalle coppie IP:porta e supporta due protocolli di comunicazione: UDP e TCP (quello che useremo)

Lato Server

- Il server crea un oggetto *java.net.ServerSocket* passando la porta
- Tramite il metodo **accept()** si mette in ascolto su quella porta e attende la connessione da parte di un di un client. Il metodo ritornerà un oggetto *Socket* relativo al client che si connette.
- Alla fine si chiude il *socket* con il metodo *close*

```
1 import java.net.ServerSocket;
2 import java.net.Socket;
3
4 //...
5 ServerSocket serverSocket = new ServerSocket(port);
6 Socket connection = serverSocket.accept();
7 //...
8 connection.close();
9 serverSocket.close();
```

Lato Client

- Basta creare un oggetto Socket passando l'hostname e la porta
- Si utilizza il socket come un normale I/O
- I metodi *getInputStream* e *getOutputStream* ritornano gli stream di ingresso/uscita associati al socket
- Alla fine si chiude il socket col metodo close

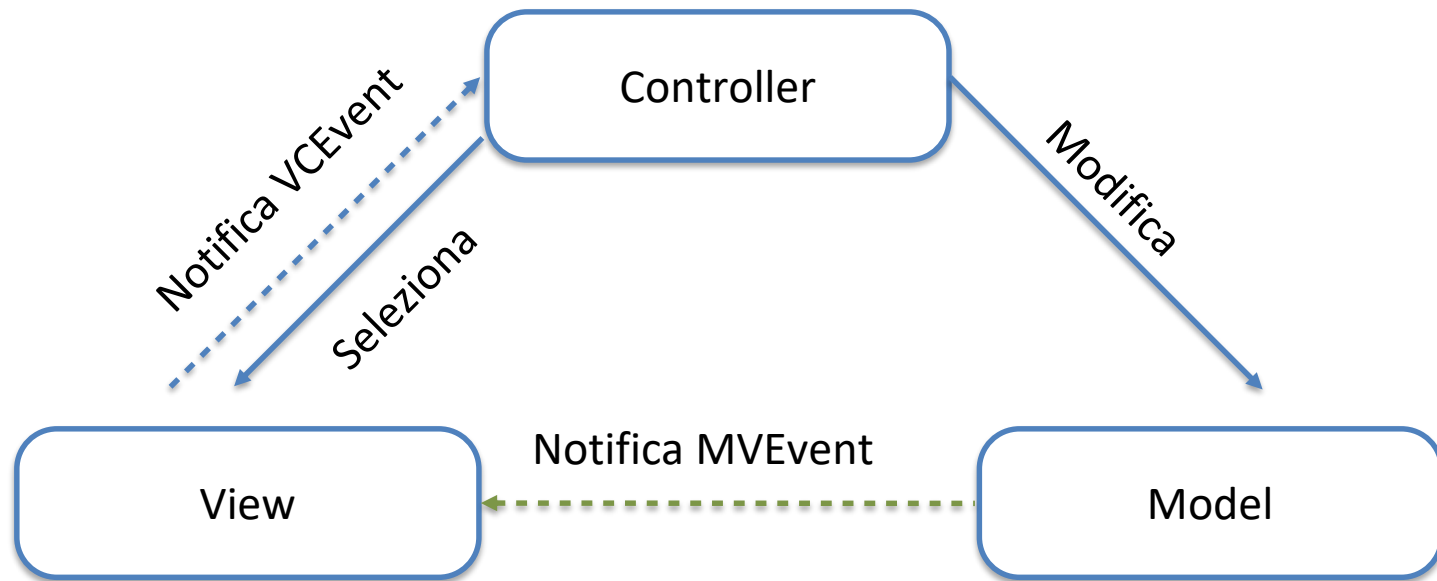
```
1 import java.net.Socket;
2
3 //...
4 Socket socket;
5 socket = new Socket(host, port);
6 OutputStream os = socket.getOutputStream();
7 InputStream reader = socket.getInputStream();
8 //...
9 socket.close();
```



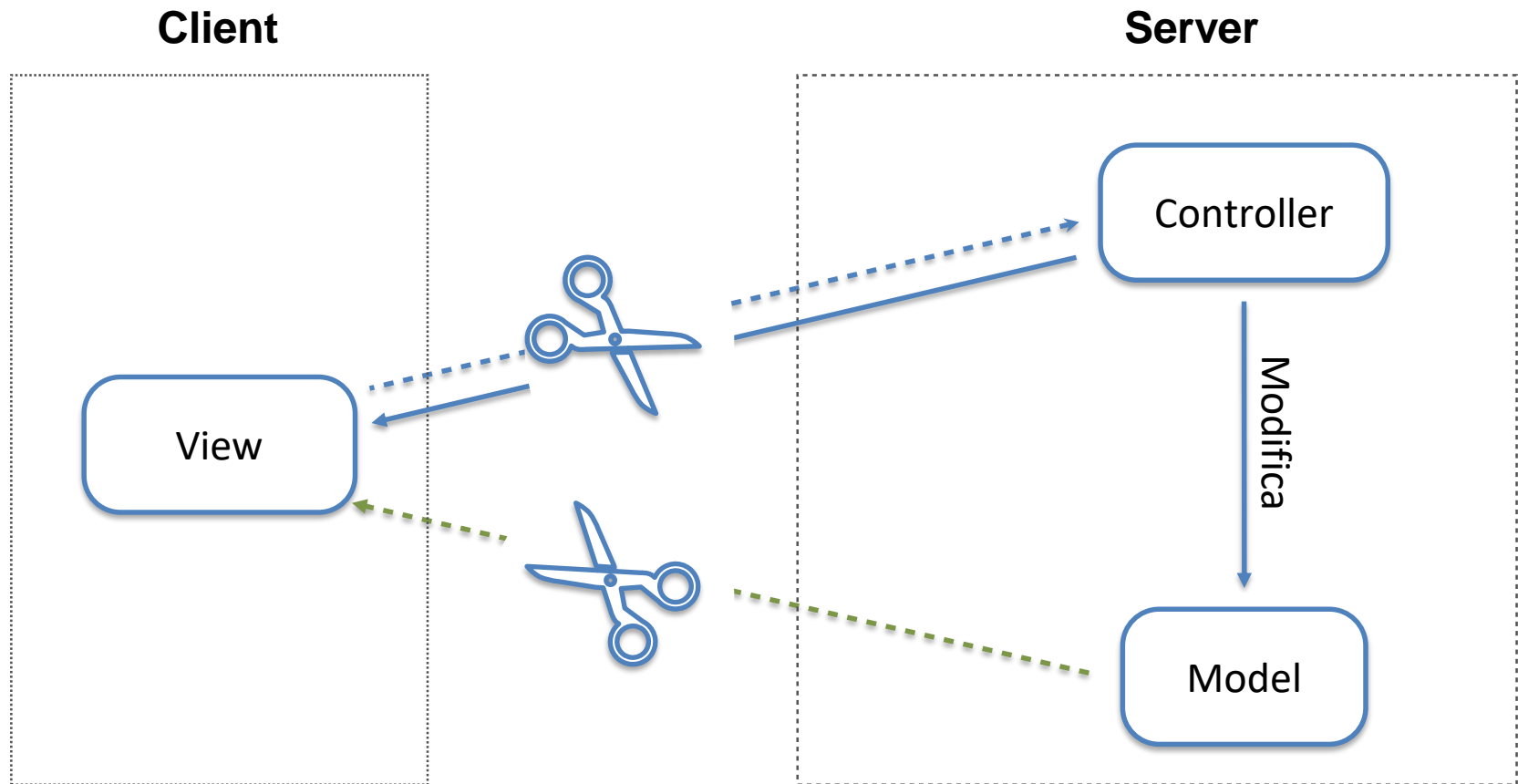
Demo

MVC + Rete

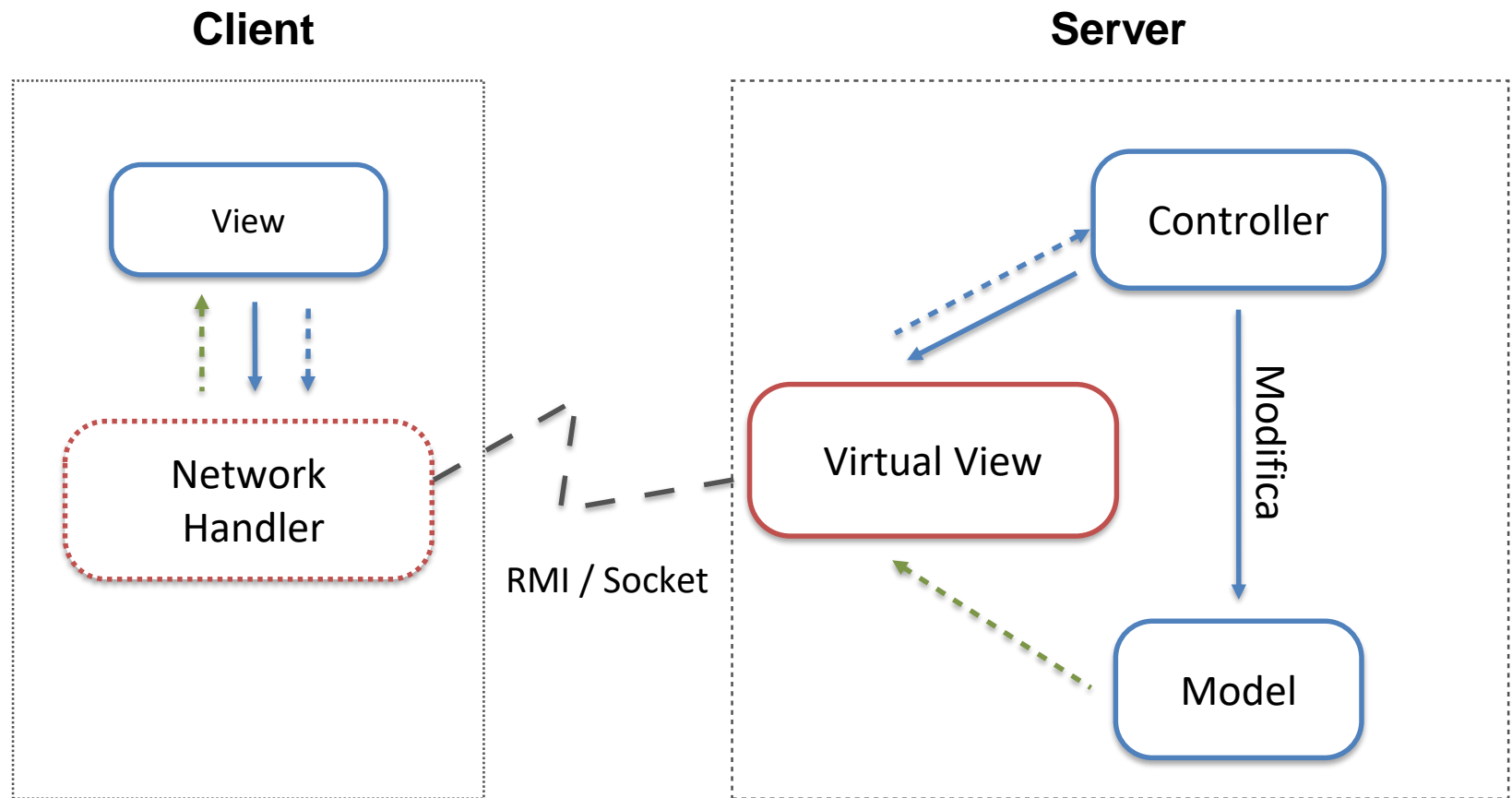
Model-View-Controller



MVC + Rete

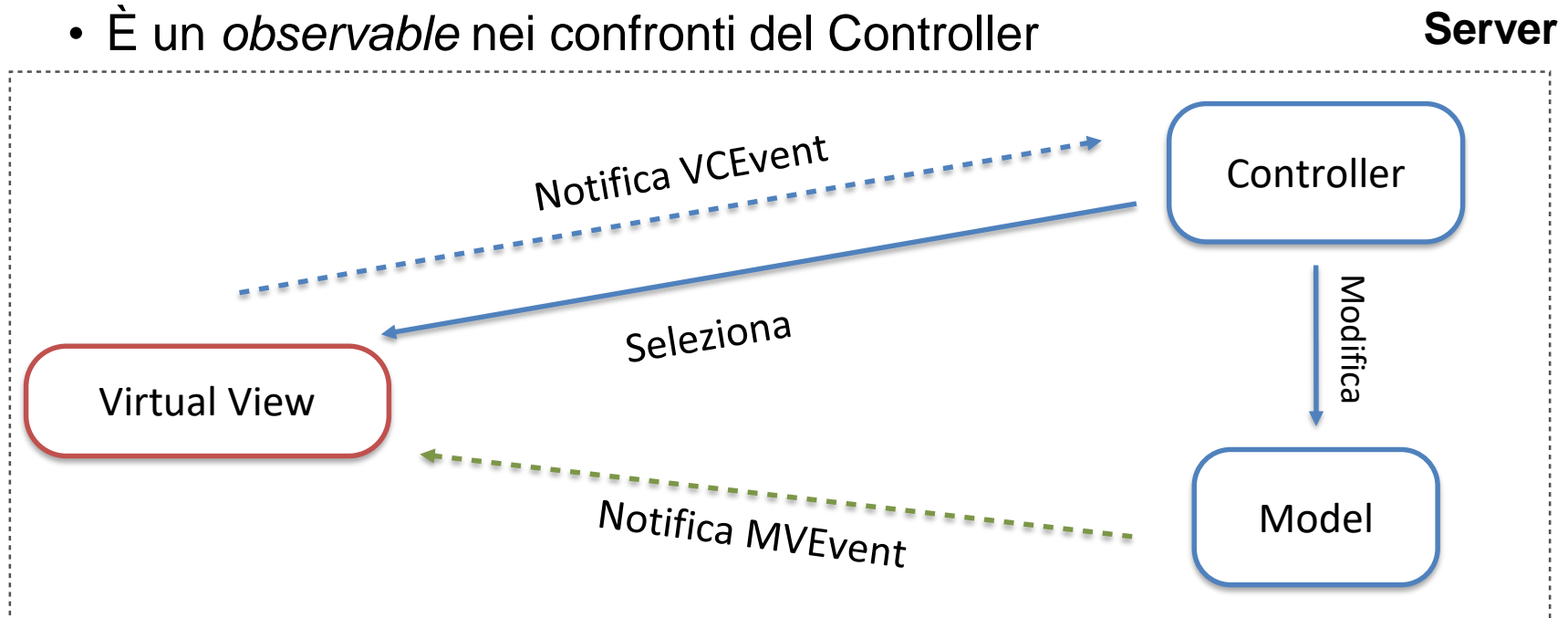


MVC + Rete




Virtual View

- Si comporta come una vera View nei confronti di Model e Controller:
 - È un *observer* nei confronti del Model
 - È un *observable* nei confronti del Controller



Network Handler

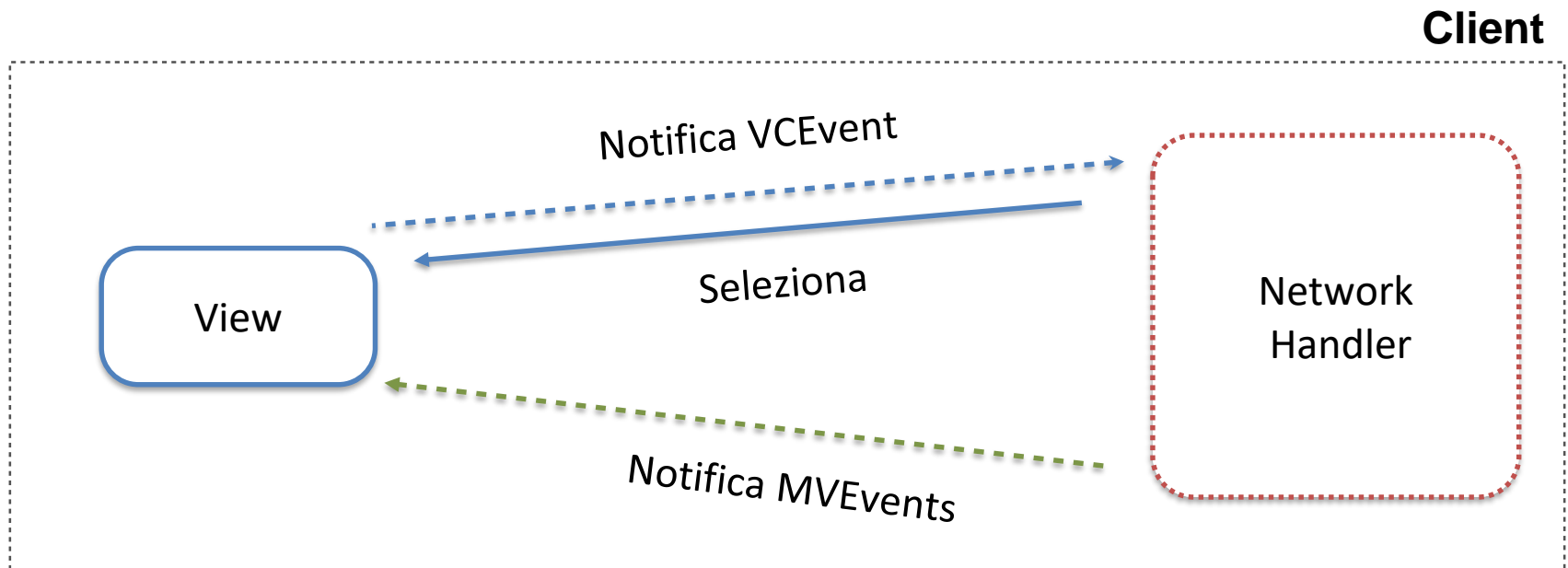
- Potrebbe corrispondere a più di un componente. La decisione su *quanti* e *quali* componenti implementano il *Network Handler* dipende dalle vostre scelte di design.



Network
Handler
?

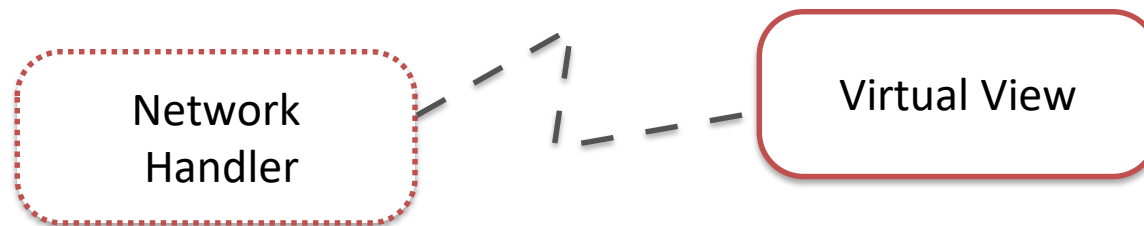
Network Handler

- Complessivamente è *observer* di eventi di tipo *VCEvent* e *observable* per eventi di tipo *MVEvent*



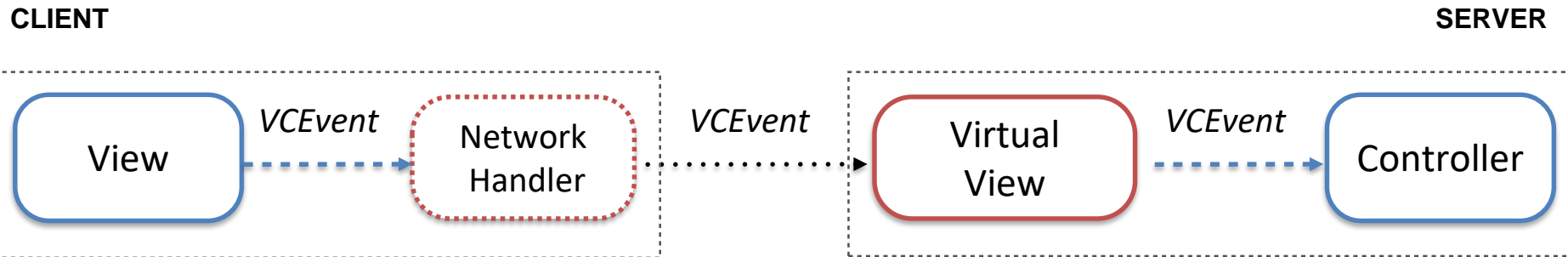
Virtual View + Network Handler

Nascondono agli altri componenti la gestione della rete



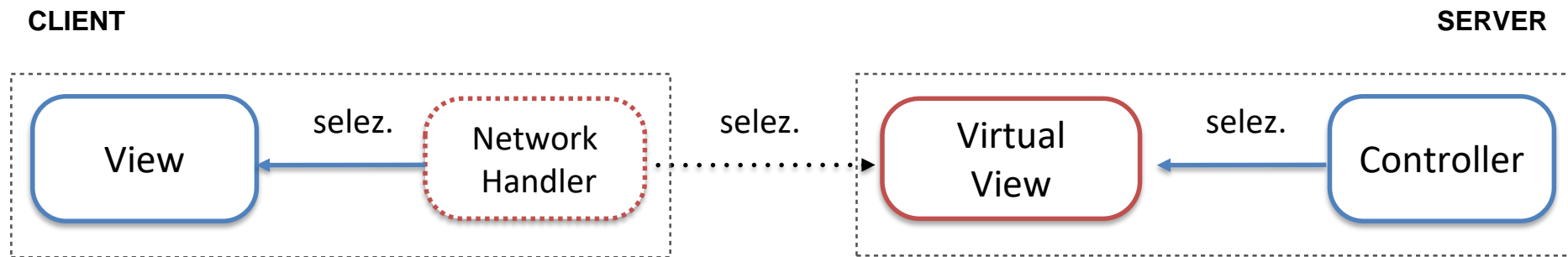
Scenario 1: VCEvents

- Il giocatore esegue una azione sulla View
- La View notifica il *Network Handler* (VCEvent)
- Il *Network Handler* inoltra l'evento alla *Virtual View* attraverso la rete
- La *Virtual View* notifica il Controller



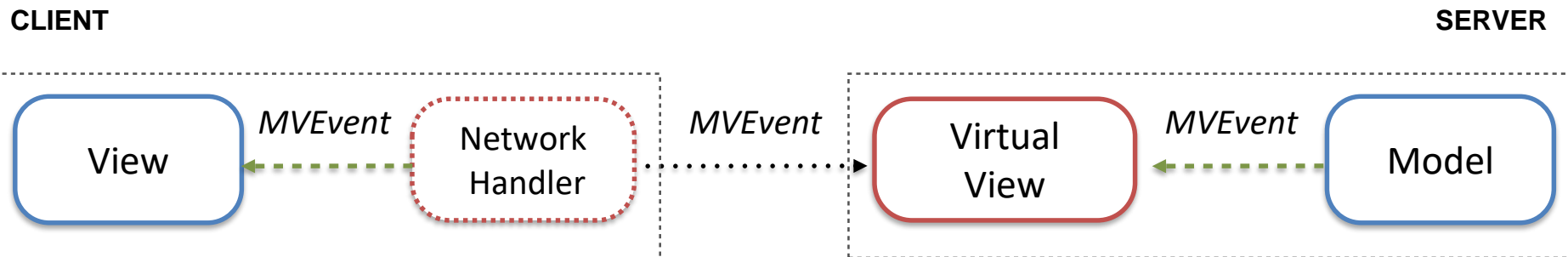
Scenario 2: Selezione

- Il Controller decide di selezionare una vista (o richiedere una azione ad un giocatore) e chiama il metodo necessario sulla Virtual View (che implementa la stessa interfaccia di View)
- La Virtual View *invoca**, attraverso la rete, un metodo sul Network Handler
- Il Network Handler chiama sulla View lo stesso metodo chiamato dal Controller sulla Virtual View



Scenario 3: MVEvent

- Il *Model* viene modificato dal *Controller* e notifica il cambiamento di stato inviando un evento (*MVEvent*) alla *VirtualView*.
- La *VirtualView* inoltra l'evento al *Network Handler* attraverso la rete
- Il *Network Handler* notifica la *View* sul Client
- Nota: questo scenario ha diverse varianti, dipendenti dalle vostre scelte di design (es: se istanziate un *Model* anche lato Client)



Disclaimer

Versione originale delle slide © 2018 Andrea Gulino