Comunicazione in distribuito

SCD

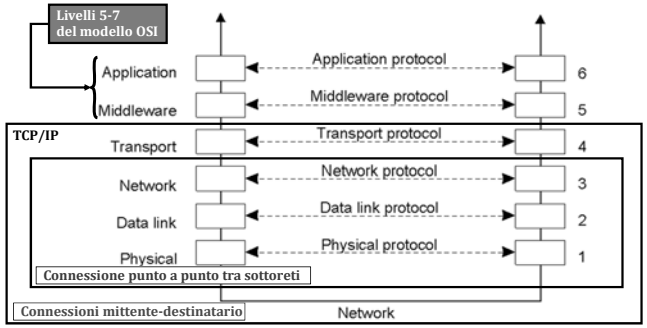
Anno accademico 2019/2020
Sistemi Concorrenti e Distribuiti

Tullio Vardanega, tullio.vardanega@unipd.it

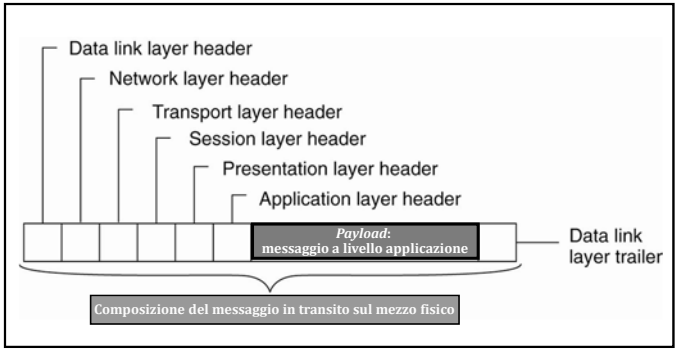
Evoluzione di modelli

- ***Remote Procedure Call*** **(RPC)**
  - Trasparente rispetto allo scambio messaggi che realizza l'interazione cliente-servente a livello applicazione
- ***Remote (Object) Method Invocation*** **(RMI)**
  - Interazione a livello applicazione come sopra ma attraverso oggetti remoti
- **Scambio messaggi a livello *middleware***
  - Con paradigmi espliciti a livello applicazione (***event-based***, Rx)
  - Interazione a livello HTTP (***pull***, REST) e superiore (***push***, WebSocket)
- ***Streaming*** **o comunicazioni a flusso continuo**
  - Flusso di dati che richiedono continuità temporale
  - Ma di queste non tratteremo

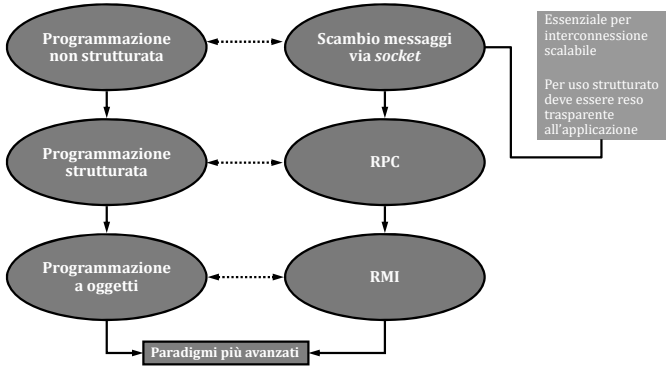
Visione a livelli – 1



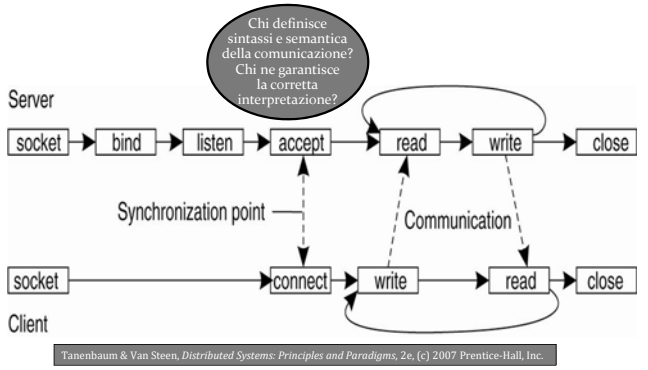
Visione a livelli – 2



Visione per analogie



La negazione dell'astrazione



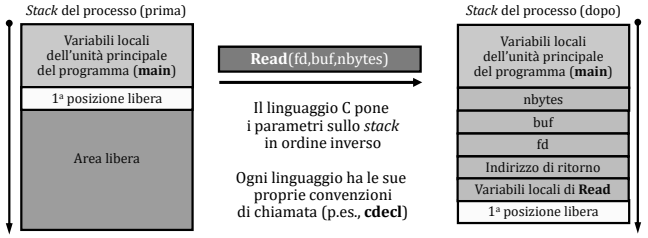
Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

RPC – 1

- **Consentire a un processo residente su un nodo di invocare <u>localmente</u> una procedura remota, cioè residente su un altro nodo**
- **Chiamante sospeso durante la chiamata**
  - **I parametri `in` viaggiano da chiamante a chiamato**
  - **I parametri `out` viaggiano da chiamato a chiamante**
- **Chiamante (processo) e chiamato (procedura) <u>non</u> sono consapevoli dello scambio di messaggi sottostante**
  - **Un caso evidente di trasparenza**

RPC – 2

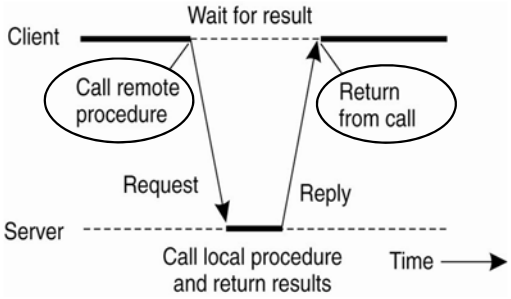
- **Chiamata di procedura locale**



RPC – 3

- I parametri di procedura locale possono essere inviati per valore (*call-by-value*) o per riferimento (*call-by-reference*)
  - Se inviati per valore sono copiati sullo *stack* del chiamato
    - Le modifiche apportate dal chiamato non hanno effetto sul chiamante
  - Se per riferimento, puntano alla memoria del chiamante
    - Ogni modifica fatta dal chiamante ha effetto sul chiamato
- La variante (`in out`), *call-by-value-result*, produce effetto sul chiamante solo al ritorno
  - Producendo un considerevole risparmio di operazioni

RPC – 4



Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

RPC – 5

- Nello spazio del chiamante le procedure remote sono «descritte» da una procedura fittizia (*client stub*) invocabile con le convenzioni locali
  - Essa svolge le azioni necessarie per effettuare la chiamata remota e riceverne il ritorno
  - Tali azioni avvengono tramite scambio messaggi in modo trasparente all'applicazione

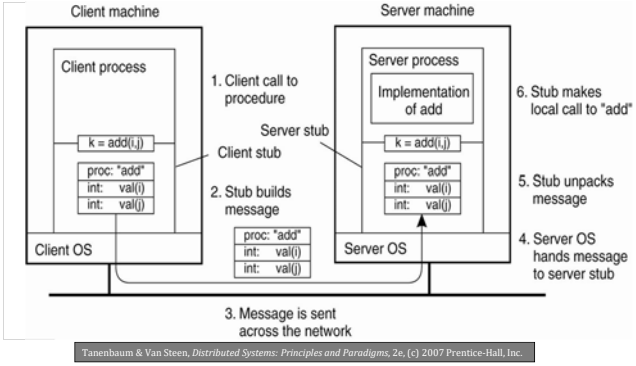
RPC – 6

- Nello spazio del chiamato l'arrivo del messaggio attiva una procedura fittizia detta *server stub*
  - Essa trasforma il messaggio in chiamata locale alla procedura invocata, ne raccoglie l'esito e lo inoltra al chiamante come messaggio
- In questo modo, chiamante e chiamato hanno reciproca trasparenza di locazione

RPC – 7

- Il *client stub* trasforma la chiamata in una sequenza di messaggi di rete
  - *Parameter marshaling*
  - Agevole con parametri passati per valore per i quali occorre solo assicurare trasparenza di accesso
    - Secondo le convenzioni locali di chiamante e chiamato
  - Arduo con parametri passati per riferimento
- Il *server stub* fa il simmetrico
  - *Parameter un-marshaling*

RPC – 8



RPC – 9

- Tre aspetti chiave caratterizzano lo specifico protocollo RPC
  - Il formato dei messaggi scambiati tra gli *stub*
  - La rappresentazione dei dati attesa da chiamante e chiamato
    - *Encoding*
  - La modalità di comunicazione su rete
    - P.es.: TCP, UDP, ...

RPC – 10

- Il servente registra il suo indirizzo (nodo:porta) presso una anagrafe pubblica
- Lo *stub* del cliente innesca una connessione di rete (TCP) verso quell'indirizzo, per inoltrare la chiamata e acquisirne l'esito
- L'azione di lato cliente è detta *binding*
  - Come fa il *client stub* a conoscere la destinazione?
- Per risparmiare porte, in ascolto sul nodo del servente può trovarsi un *daemon* che agisce per conto di più serventi

RPC – 11

- RPC è sincrona
  - Può essere asincrona solo se priva di parametri `out`
- L'eventualità di errori trasmissivi produce una tassonomia di semantiche determinata dal protocollo *request-reply* in uso tra gli *stub*
  - *At-least-once*
  - *At-most-once*
  - *Exactly-once*

Semantica della comunicazione – 1

- Il protocollo *request-reply* di RPC utilizza 3 meccanismi basati sull'attesa di conferma
  - Lato cliente: ***Request Retry – RR1***
    - Il cliente prova fino a ottenere risposta o certezza di irraggiungibilità
  - Lato servente: ***Duplicate Filter – DF***
    - Il servente scarta eventuali duplicati provenienti dallo stesso cliente
  - Lato servente: ***Result Retransmit – RR2***
    - Il servente conserva le risposte, per ritrasmettere senza ricalcolo
    - Fondamentale per calcolo non idempotente

Semantica della comunicazione – 2

- Semantica *best effort*
  - Nessun meccanismo aggiuntivo
  - Il cliente non può sapere quante volte la sua richiesta sia stata eseguita
- Semantica *at least once*
  - Il lato cliente usa RR1
  - Il lato servente niente
  - Se la risposta arriva, il cliente non sa quante volte sia stata calcolata dal servente

Semantica della comunicazione – 3

- Semantica *at most once*
  - Il lato cliente usa RR1
  - Il lato servente usa DF e RR2
  - Se la risposta arriva, è stata calcolata una sola volta
  - La risposta non arriva solo a servente guasto
- Semantica *exactly once*
  - Ha bisogno di meccanismi addizionali per tollerare guasti nel servente
    - P.es. replicazione trasparente

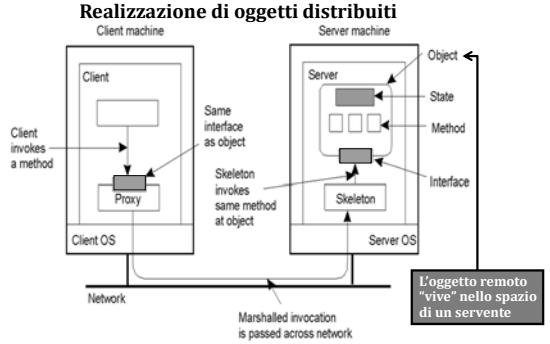
RMI – 1

- **Il paradigma RPC può essere facilmente esteso al modello a oggetti distribuiti**
  - **CORBA: *Common Object Request Broker Architecture***
    - OMG – con enfasi sulla interoperabilità
  - **DCOM: *Distributed Component Object Model*, poi diventato .NET**
    - Microsoft – con enfasi sull'omogeneità
  - **J2EE: Java Platform, Enterprise Edition**
    - Sun Microsystems, poi acquisita da Oracle – con enfasi sull'omogeneità

RMI – 2

- **La separazione logica tra interfaccia e oggetto facilita la distribuzione**
  - **L'interfaccia di oggetto può essere pubblicato remotamente senza coinvolgere l'istanza corrispondente**
  - **Al *binding* di un cliente con un oggetto remoto, una copia dell'interfaccia del "servente" (*proxy*) viene caricata nello spazio del cliente**
    - Il ruolo del *proxy* è analogo a quello del *client stub* in ambiente RPC
  - **La richiesta in arrivo all'oggetto remoto viene trattata da un "agente" (*skeleton*) del cliente localmente al servente**
    - Il ruolo dello *skeleton* è analogo a quello del *server stub* in ambiente RPC

RMI – 3



RMI – 4

- **Vi sono oggetti di tipo *compile-time***
  - **Con realizzazione completamente determinata dal linguaggio di programmazione**
    - Ambiente e protocollo d'uso noti e uniformi ma non *interoperable*
- **E oggetti di tipo *run-time***
  - **Quando ciò che non è oggetto appare come tale**
    - L'entità concreta (la sua interfaccia) viene incapsulata in un ***object wrapper*** che appare all'esterno come un normale oggetto remoto
    - In questo modo si ottiene ***interoperability***

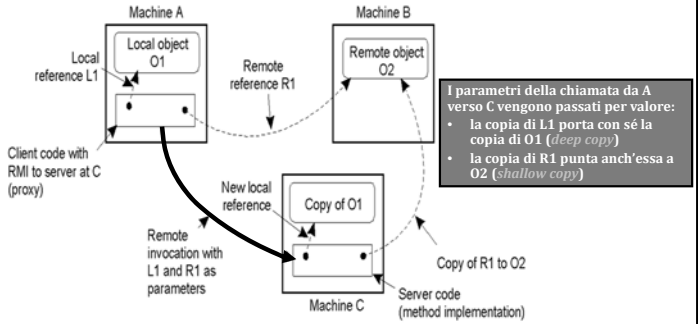
RMI – 5

- **Vi sono oggetti persistenti**
  - **Che continuano a esistere anche al di fuori dello spazio di indirizzamento del servente**
    - Lo stato persistente dell'oggetto distribuito viene salvato in memoria secondaria e da lì ripristinato dai processi servente delegati a farlo
- **E oggetti transitori**
  - **Che cessano di esistere con la terminazione del servente che li contiene**
- **Modelli RMI diversi fanno scelte diverse**

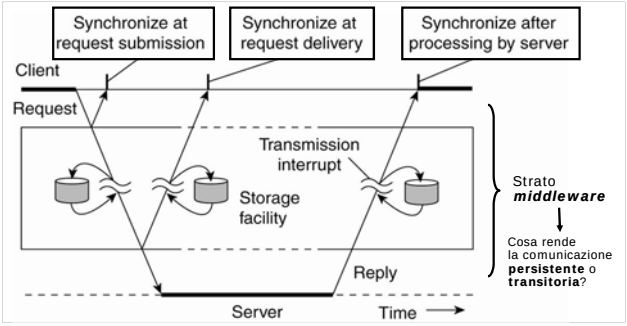
RMI – 6

- **RMI offre maggiore trasparenza di RPC**
  - **I riferimenti a oggetti distribuiti hanno *scope* globale e possono essere liberamente scambiati a livello sistema**
  - **Un riferimento poco scalabile usa un analogo del *daemon* RPC per interconnettere cliente e servente dell'oggetto**
    - <indirizzo di rete del *daemon*; identificatore del servente>
- **Modalità di riferimento**
  - ***Explicit binding***
    - Il cliente si rivolge a una anagrafe che restituisce un puntatore localmente utilizzabile al ***proxy*** dell'oggetto servente (**Java RMI**)
  - ***Implicit binding***
    - Il linguaggio risolve direttamente il riferimento in modo invisibile al cliente

RMI – 8

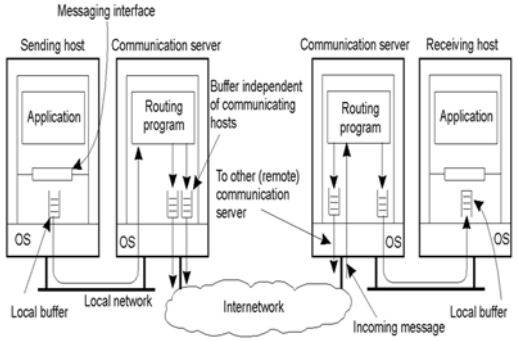


Scambio messaggi: sincronizzazione



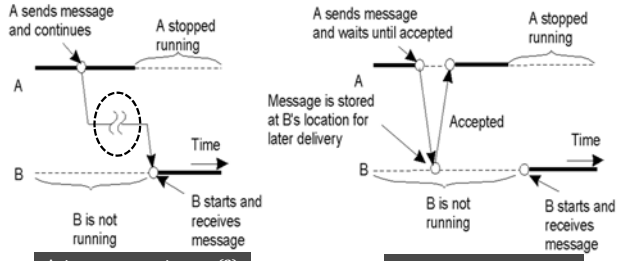
Tratto da: Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

Scambio messaggi: persistenza



Lo scambio messaggi distribuito comporta problemi di persistenza della comunicazione e di sincronizzazione tra mittente e destinatario

Varianti scambio messaggi – 1

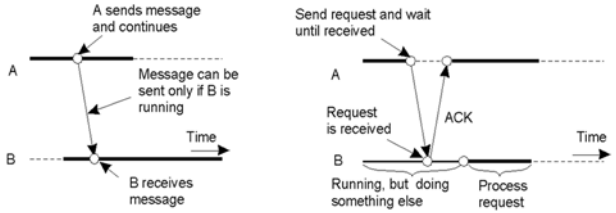


Asincrona, persistente(?)

Persistente, sincrona(?)

Tratto da: Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

Varianti scambio messaggi – 2

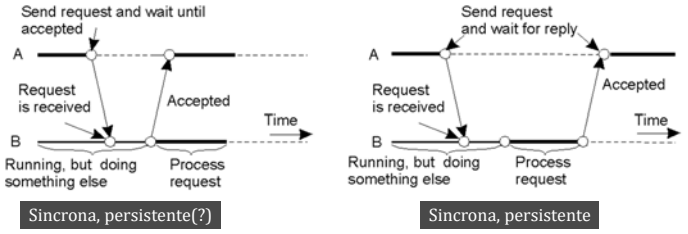


Asincrona, persistente(?)

Persistente, sincrona(?)

Tratto da: Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

Varianti scambio messaggi – 3



Sincrona, persistente(?)

Sincrona, persistente

Tratto da: Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

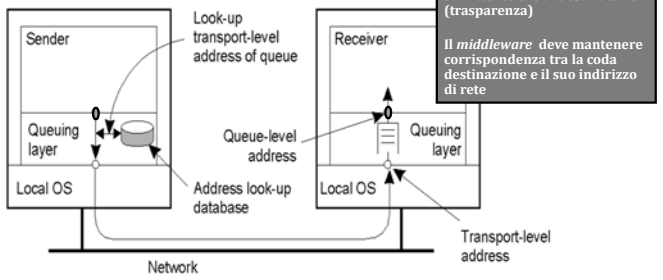
Scambio messaggi: realizzazione – 1

- ***Middleware* orientato a messaggi**
  - **Applicazioni distribuite comunicano tramite inserzione di messaggi in specifiche code**
    - Modello a code di messaggi
    - Eccellente supporto a comunicazioni persistenti e asincrone
    - Ma senza garanzia che il destinatario prelevi il messaggio dalla sua coda
  - **Di immediata realizzazione tramite**
    - **`Put`** non bloccante (asincrona → come trattare il caso di coda piena?)
    - **`Get`** bloccante (sincrona rispetto alla presenza di messaggi in coda)
      - Un meccanismo di ***callback*** separa la coda dall'attivazione del destinatario
    - Un PO realizza la coda con metodo **`Put`** di tipo **`P`** e metodo **`Get`** di tipo **`E`**
    - Coda *proxy* @ mittente e coda *skeleton* @ destinatario

Scambio messaggi: realizzazione – 2

La coda appare locale sia al mittente che al destinatario (trasparenza)

Il *middleware* deve mantenere corrispondenza tra la coda destinazione e il suo indirizzo di rete

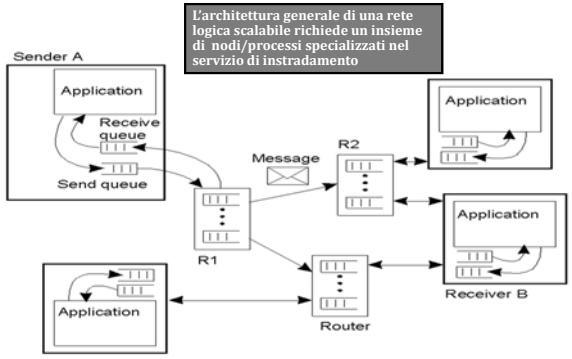


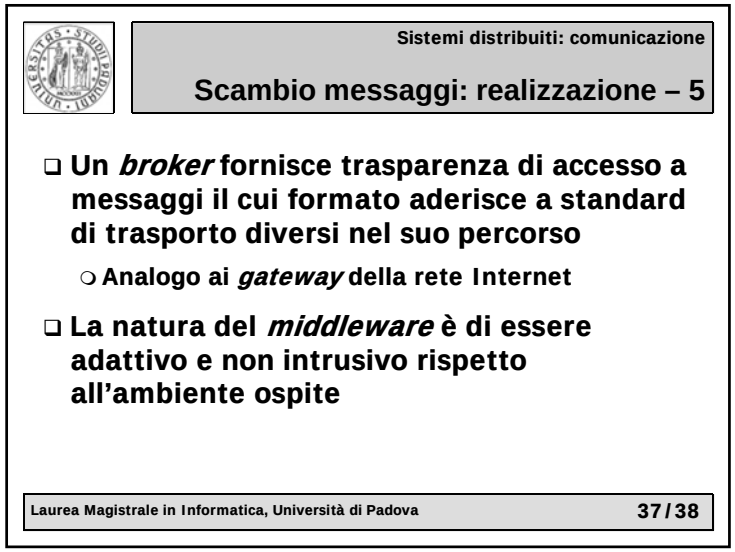
Scambio messaggi: realizzazione – 3

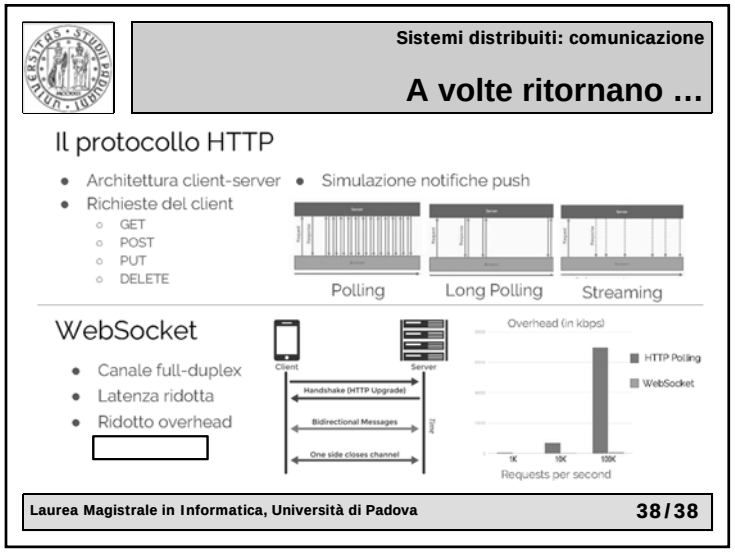
- **Il *middleware* realizza una rete logica sovrapposta a quella fisica con topologia propria e distinta**
  - ***Overlay network* con proprio servizio di *routing***
    - Quei *router* conoscono la topologia della rete logica e inoltrano i messaggi del mittente alla coda del destinatario
  - **Topologie complesse e variabili richiedono gestione dinamica delle corrispondenze <coda destinatario : indirizzo di rete>**
    - In totale analogia con quanto avviene nel modello IP

Scambio messaggi: realizzazione – 4

L'architettura generale di una rete logica scalabile richiede un insieme di nodi/processi specializzati nel servizio di instradamento




Sistemi distribuiti: comunicazione
Scambio messaggi: realizzazione – 5
Un *broker* fornisce trasparenza di accesso a messaggi il cui formato aderisce a standard di trasporto diversi nel suo percorso
Analogo ai *gateway* della rete Internet
La natura del *middleware* è di essere adattivo e non intrusivo rispetto all'ambiente ospite
Laurea Magistrale in Informatica, Università di Padova
37 / 38


Sistemi distribuiti: comunicazione
A volte ritornano …
Il protocollo HTTP
Architettura client-server
Richieste del client
GET
POST
PUT
DELETE
Simulazione notifiche push
Polling
Long Polling
Streaming
WebSocket
Canale full-duplex
Latenza ridotta
Ridotto overhead
Client
Server
Handshake (HTTP Upgrade)
Bidirectional Messages
One side closes channel
Time
Overhead (in kbps)
HTTP Polling
WebSocket
1K
10K
100K
Requests per second
Laurea Magistrale in Informatica, Università di Padova
38 / 38