Programmazione di Reti

Laboratorio #7

Andrea Piroddi

Dipartimento di Informatica, Scienza e Ingegneria

Trasferimento di file crittografati tramite socket in Python

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

CRITTOGRAFIA AES (Advanced Encryption Standard)

L'algoritmo di crittografia simmetrica più popolare e ampiamente adottato è l'Advanced Encryption Standard (AES).

Le caratteristiche di AES sono le seguenti:

- Cifrario a blocchi simmetrico a chiave simmetrica
- Dati a 128 bit, chiavi a 128/192/256 bit

CRITTOGRAFIA AES (Advanced Encryption Standard)

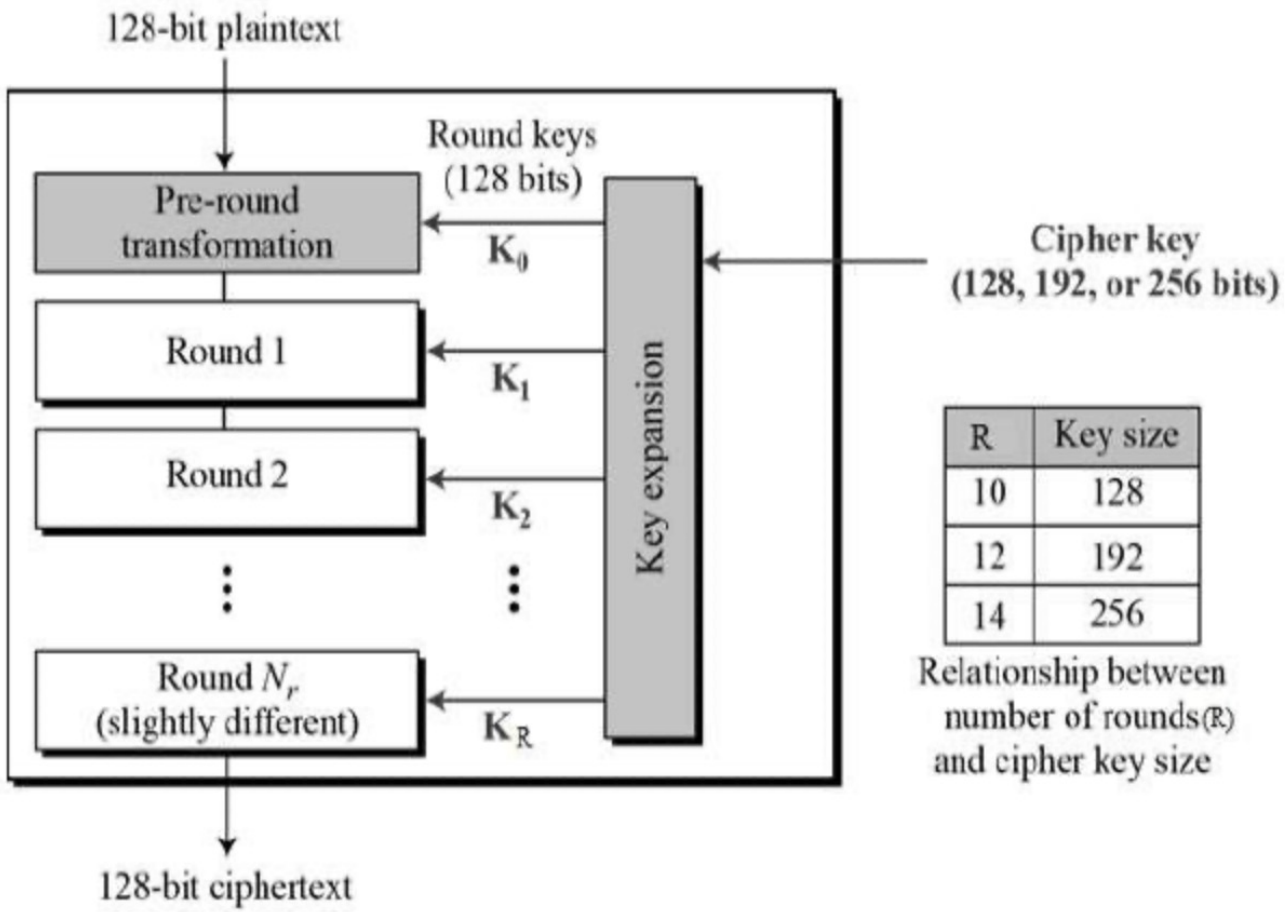
AES è un cifrario iterativo. Si basa su "sostituzione-permutazione". Comprende una serie di operazioni collegate, alcune delle quali comportano la sostituzione di input con output specifici (sostituzioni) e altre implicano il mescolamento di bit (permutazioni).

È interessante notare che AES esegue tutti i suoi calcoli su byte anziché su bit. Pertanto, AES tratta i 128 bit di un blocco di testo in chiaro come 16 byte. Questi 16 byte sono disposti in quattro colonne e quattro righe per l'elaborazione come matrice.

Il numero di cicli in AES è variabile e dipende dalla lunghezza della chiave. AES utilizza 10 round per chiavi a 128 bit, 12 round per chiavi a 192 bit e 14 round per chiavi a 256 bit. Ciascuno di questi cicli utilizza una diversa chiave a 128 bit, che viene calcolata dalla chiave AES originale.

CRITTOGRAFIA AES (Advanced Encryption Standard)

Lo schema della struttura AES è riportato nell'illustrazione seguente:

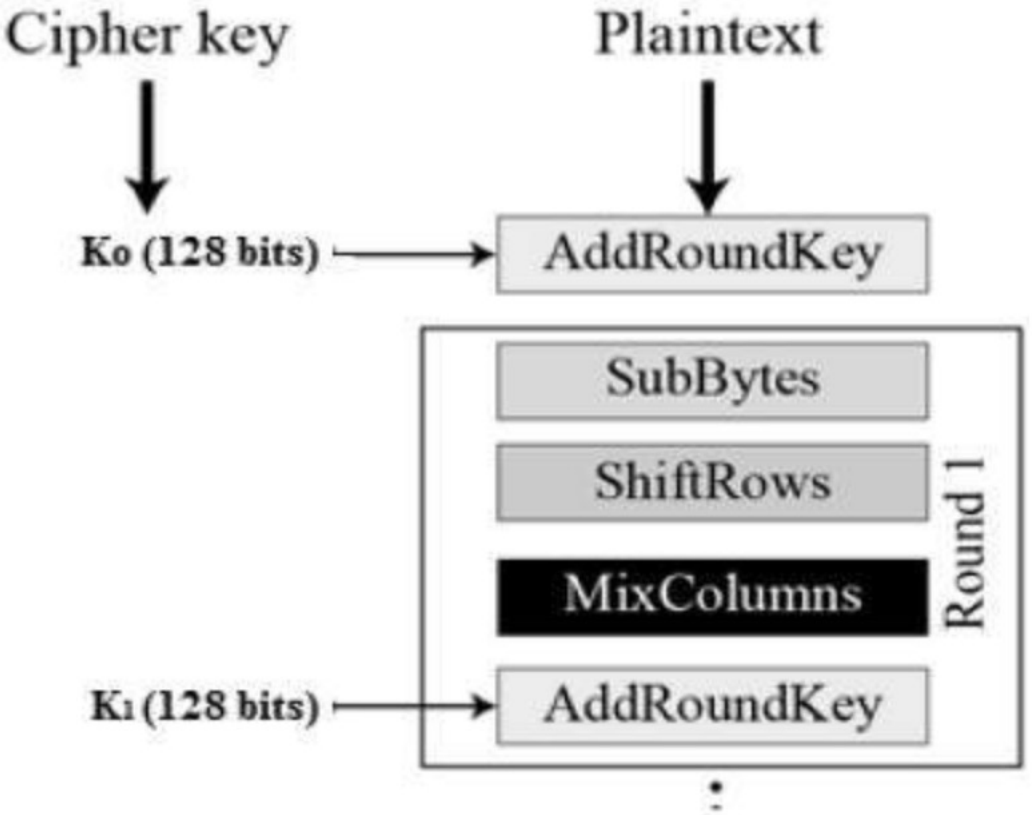


R	Key size
10	128
12	192
14	256

Relationship between number of rounds (R) and cipher key size

CRITTOGRAFIA AES (Advanced Encryption Standard)

Ogni round comprende quattro sottoprocessi. Il processo del primo round è illustrato di seguito



Trasferimento di file crittografati tramite socket in Python

Il trasferimento di file in modo sicuro è una funzionalità indispensabile in qualsiasi ambiente di sviluppo.

La crittografia end-to-end garantisce che nemmeno il server possa leggere il contenuto dei dati.

Il trasferimento di file crittografati su socket in Python è un metodo per inviare in modo sicuro file da un computer a un altro.

Implica la creazione di una connessione socket tra il client e il server e la crittografia del file sul lato client prima di inviarlo al server.

Il server quindi decrittografa il file e lo salva nella posizione desiderata.

Trasferimento di file crittografati tramite socket in Python

Il processo può essere suddiviso in diverse fasi:

1. Il client stabilisce una connessione con il server utilizzando il modulo socket.

2. Il client legge il file da trasferire e lo crittografa utilizzando una libreria come ***pycrypto*** o ***cryptography.***

3. Il file crittografato viene inviato al server tramite la connessione socket stabilita.

4. Lato server, il file crittografato ricevuto viene decrittografato utilizzando la stessa libreria di crittografia e salvato nella posizione desiderata.

Trasferimento di file crittografati tramite socket in Python

È importante notare che la chiave di crittografia deve essere scambiata in modo sicuro tra il client e il server prima del trasferimento.
Ciò può essere ottenuto tramite un protocollo di scambio di chiavi sicuro come Diffie-Hellman.

È anche importante utilizzare un algoritmo di crittografia robusto come AES per garantire che il file sia protetto durante la trasmissione.

Inoltre, è anche importante gestire correttamente gli errori, i trasferimenti di file di grandi dimensioni e la gestione delle chiavi.

Trasferimento di file crittografati tramite socket in Python – Diffie-Helmann

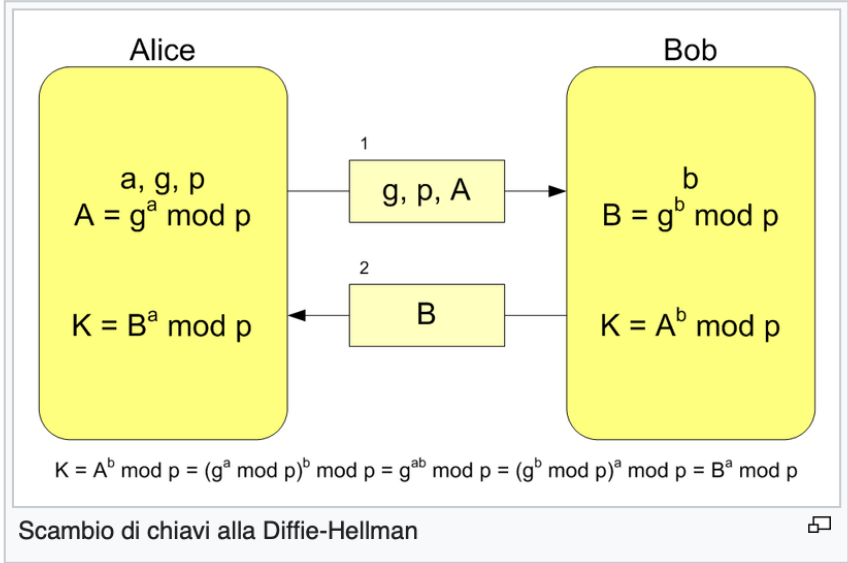
Si considera inizialmente un numero *g*, generatore del gruppo moltiplicativo degli interi modulo *p*, dove *p* è un numero primo.

Uno dei due interlocutori, ad esempio Alice, sceglie un numero casuale *"a"* e calcola il valore $A = g^a \bmod p$ (dove *mod* indica l'operazione modulo, ovvero il resto della divisione intera) e lo invia attraverso il canale pubblico a Bob (l'altro interlocutore), assieme ai valori *g* e *p*.

Bob da parte sua sceglie un numero casuale "*b*" , calcola $B = g^b \bmod p$ e lo invia ad Alice.

A questo punto Alice calcola $K_A = B^a \bmod p$, mentre Bob calcola $K_B = A^b \bmod p$.

I valori calcolati sono gli stessi, in quanto $B^a \bmod p = A^b \bmod p$.



Scambio di chiavi alla Diffie-Hellman

A questo punto i due interlocutori sono entrambi in possesso della chiave segreta e possono cominciare ad usarla per cifrare le comunicazioni successive.

Un attaccante può ascoltare tutto lo scambio, ma per calcolare i valori a e b avrebbe bisogno di risolvere l'operazione del logaritmo discreto, che è computazionalmente onerosa e richiede parecchio tempo, in quanto sub-esponenziale (sicuramente molto più del tempo di conversazione tra i 2 interlocutori).

Trasferimento di file crittografati tramite socket in Python

Per questo trasferimento di file crittografato utilizzeremo la crittografia simmetrica, ciò significa sostanzialmente che utilizzeremo la stessa chiave per la crittografia e per la decrittografia.

Questo metodo è diverso dalla crittografia RSA in cui abbiamo chiavi pubbliche e private, dove la chiave pubblica è utilizzata per la crittografia mentre la chiave privata è utilizzata per la decrittografia

Prima di andare avanti dobbiamo installare due pacchetti python:

Trasferimento di file crittografati tramite socket in Python

Pycryptodome:

Pycryptodome è un pacchetto Python che supporta sia Python 2 che 3.

Pycryptodome fornisce una raccolta di moduli crittografici che supportano vari algoritmi come cifrari simmetrici, digest di messaggi e sistemi crittografici a chiave pubblica.

Può essere utilizzato per attività come la crittografia e la decrittografia dei dati, la generazione e la gestione di chiavi crittografiche e l'esecuzione di altre operazioni crittografiche.

Per installare pycrypto utilizzare il seguente comando nel prompt dei comandi o nel terminale:

```
1  pip3 install Pycryptodome
```

Trasferimento di file crittografati tramite socket in Python

TQDN:

TQDN è una libreria Python che fornisce una barra di avanzamento rapida ed estensibile per loop e tabelle. È progettato per fornire un feedback visivo sull'avanzamento di un ciclo o di un'iterazione e può essere utilizzato per tenere traccia dell'avanzamento di attività quali trasferimenti di file, elaborazione dati e altro.

Per installare TQDN utilizzare il seguente comando nel prompt dei comandi o nel terminale:

```
1  pip3 install tqdm
```

Trasferimento di file crittografati tramite socket in Python

Ora andiamo a creare due script uno per il mittente e uno per il destinatario.

Step 1: importare la libreria richiesta

Step 2: creare la chiave

Step 3: crea un codice

Step 4: creare un socket TCP per la comunicazione Internet

Step 5: connettersi all'host locale in cui è ospitato il server ricevente

Step 6: Calcola la dimensione dei file da inviare

Step 7: caricare i dati sotto forma di byte

Step 8: crittografare i dati

Step 9: inviare al destinatario il nome del file, la dimensione del file, il file crittografato e un tag di chiusura per indicare che il file è stato ricevuto completamente.

Trasferimento di file crittografati tramite socket in Python - SERVER

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Apr  1 18:27:34 2023
LABORATORIO DI PROGRAMMAZIONE DI RETI
@author: apirodd
"""

"""
Server che riceve il file
"""
import socket
import tqdm
import os
from Crypto.Cipher import AES

#definiamo le chiavi
key = b"TheBestRandomKey"
nonce = b"TheBestRandomNce"

cipher = AES.new(key, AES.MODE_EAX, nonce)

filename1="Cartel3.csv"
# IP address del server ricevente
SERVER_HOST = "127.0.0.1"
SERVER_PORT = 5001
# riceve 4096 bytes ogni trance
BUFFER_SIZE = 1024 *4
SEPARATOR = "<SEPARATOR>"
# creiamo il socket del server
# TCP socket
s = socket.socket()
# facciamo il bind del socket all'ip address locale
s.bind((SERVER_HOST, SERVER_PORT))

# abilitiamo il nostro server ad accettare connessioni
# 5 è il numero di connessioni in attesa che il sistema
# processerà prima, le altre saranno rifiutate

s.listen(5)
print(f"[*] Listening as {SERVER_HOST}:{SERVER_PORT}")
```

Trasferimento di file crittografati tramite socket in Python - SERVER

```python
print(f"[*] Listening as {SERVER_HOST}:{SERVER_PORT}")
# accetta la connessione se presente

client_socket, address = s.accept()
# se il codice che segue viene eseguito significa che il mittente è connesso
print(f"[+] {address} is connected.")
# riceve le informazioni sul file
# riceve usandi il socket del client, non il socket del server

received = client_socket.recv(BUFFER_SIZE).decode("utf-8", "ignore")
pippo=[filename, filesize] = received.split(SEPARATOR)
# rimuove il percorso assoluto se presente

filename = os.path.basename(pippo[0])
# converte in un intero
filesize = int(pippo[1])
# comincia a ricevere il file dal socket
# e lo scrive sul file stream

progress = tqdm.tqdm(range(filesize), f"Receiving {filename}", unit="B", unit_scale=True, unit_divisor=1024)
with open(filename1, "wb") as f:
    while True:
        # legge 1024 bytes dal socket (receive)
        bytes_read = client_socket.recv(BUFFER_SIZE)
        if not bytes_read:
            # non riceve nulla
            # la trasmissione del file è completa
            break

        # scrive nel file i bytes che abbiamo ricevuto
        f.write(bytes_read)
        print(bytes_read)
        # aggiorna la barra di avanzamento
        progress.update(len(bytes_read))
        if bytes_read[-5:] == b"<END>":
            break
# chiude il client socket
client_socket.close()
# chiude il server socket
s.close()
```

Trasferimento di file crittografati tramite socket in Python - CLIENT

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Apr  1 18:26:35 2023
LABORATORIO DI PROGRAMMAZIONE DI RETI
@author: apirodd
"""

"""
Client that sends the file (uploads)
"""
import socket
import tqdm
import os
import argparse
from Crypto.Cipher import AES
key = b"TheBestRandomKey"
nonce = b"TheBestRandomNce"
# creiamo la cifratura

cipher = AES.new(key, AES.MODE_EAX, nonce)

SEPARATOR = "<SEPARATOR>"
BUFFER_SIZE = 1024 * 4 #4KB

def send_file(filename, host, port):
    # recuperiamo la dimensione del file
    filesize = os.path.getsize(filename)
    # creiamo il socket del client
    s = socket.socket()
    print(f"[+] Connecting to {host}:{port}")
    s.connect((host, port))
    print("[+] Connected.")

    # inviamo il nome del file e la dimensione del file
    s.send(f"{filename}{SEPARATOR}{filesize}".encode())
```

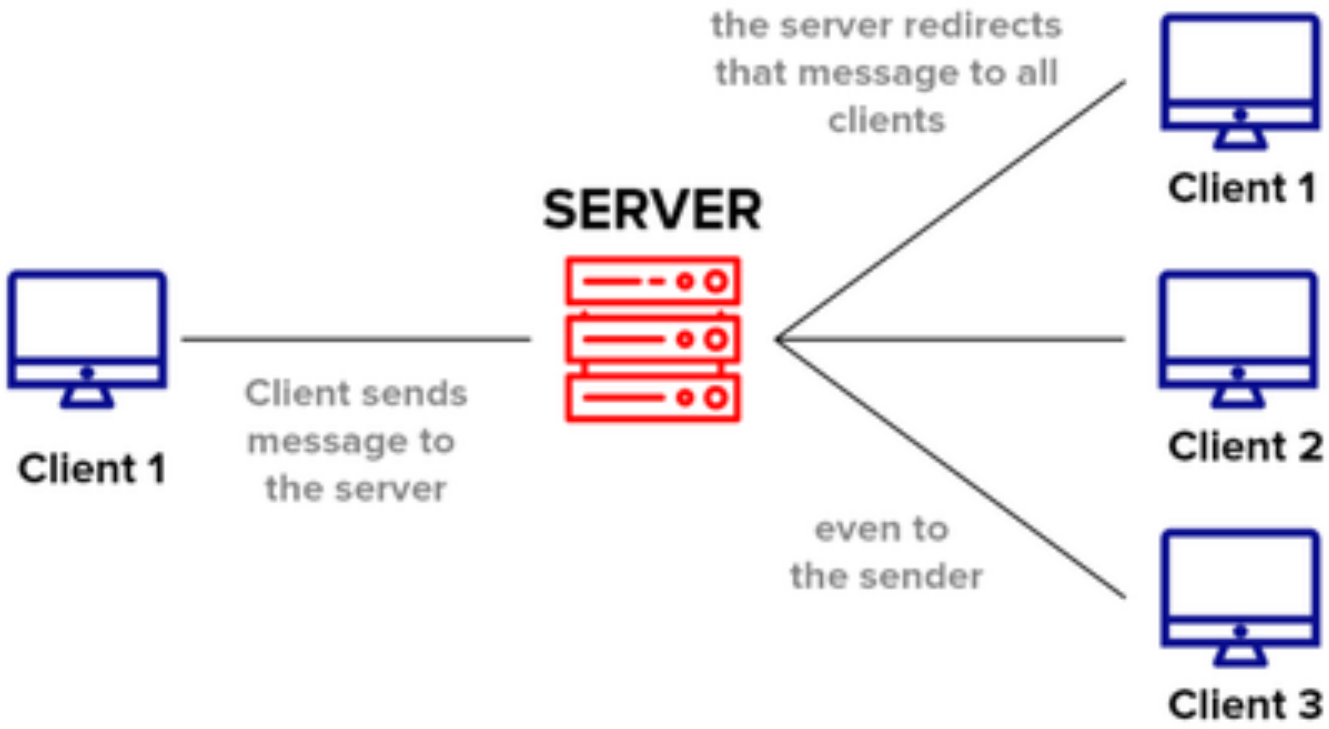
Trasferimento di file crittografati tramite socket in Python - CLIENT

```
37
38      # cominciamo ad inviare il file
39      progress = tqdm.tqdm(range(filesize), f"Sending {filename}", unit="B", unit_scale=True, unit_divisor=1024)
40      with open(filename, "rb") as f:
41          while True:
42              # leggiamo i bytes dal file
43              bytes_read = f.read(BUFFER_SIZE)
44              if not bytes_read:
45                  # la trasmissione del file è completata
46                  break
47              # utilizziamo sendall per garantire la trasmissione
48              # in reti trafficate
49              # criptiamo i dati
50              encrypted = cipher.encrypt(bytes_read)
51              s.sendall(encrypted)
52              # aggiorniamo la barra di avanzamento
53              progress.update(len(bytes_read))
54          s.send(b"<END>")
55      # chiudiamo il socket
56      s.close()
57
58  if __name__ == "__main__":
59      import argparse
60      parser = argparse.ArgumentParser(description="Simple File Sender")
61      parser.add_argument("file", help="File name to send")
62      parser.add_argument("host", help="The host/IP address of the receiver")
63      parser.add_argument("-p", "--port", help="Port to use, default is 5001", default=5001)
64      args = parser.parse_args()
65      filename = args.file
66      host = args.host
67      port = args.port
68      send_file(filename, host, port)
69
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

CHAT CLIENT SERVER

CHAT client - server



Lato SERVER

Servizio CHAT – Lato Server

```
clients = {}
indirizzi = {}

HOST = ''
PORT = 53000
BUFSIZ = 1024
ADDR = (HOST, PORT)

SERVER = socket(AF_INET, SOCK_STREAM)
SERVER.bind(ADDR)
```

Definiamo gli elementi iniziali:

- Due dizionari per la registrazione dei client in ingresso e dei relativi nomi associati agli indirizzi
- La porta e l'interfaccia del server
- E il bind

Servizio CHAT – Lato Server

```
#!/usr/bin/env python3
"""Script Python per la realizzazione di un Server multithread
per connessioni CHAT asincrone.
Corso di Programmazione di Reti - Università di Bologna"""

from socket import AF_INET, socket, SOCK_STREAM
from threading import Thread
```

Per questo scopo useremo i socket TCP.

Quindi importiamo dal modulo socket gli elementi di cui abbiamo bisogno.

In linea di principio potremmo usare anche i socket UDP.

Chiaramente, TCP si adatta meglio al nostro intento rispetto ai socket UDP.

Servizio CHAT – Lato Server

{<socket.socket fd=752, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 53000), raddr=('127.0.0.1', 62825)>: ('127.0.0.1', 62825)}

```
""" La funzione che segue accetta le connessioni  dei client in entrata."""
def accetta_connessioni_in_entrata():
    while True:
        client, client_address = SERVER.accept()
        print("%s:%s si è collegato." % client_address)
        #al client che si connette per la prima volta fornisce alcune indicazioni di utilizzo
        client.send(bytes("Salve! Digita il tuo Nome seguito dal tasto Invio!", "utf8"))
        # ci serviamo di un dizionario per registrare i client
        indirizzi[client] = client_address
        #diamo inizio all'attività del Thread - uno per ciascun client
        Thread(target=gestice_client, args=(client,)).start()
```

- Definiamo una funzione che gestisca le connessioni in entrata.
- Quindi quando viene invocata, essa si mette in ascolto sul socket, e appena arriva una richiesta di connessione da un client, il server gli restituisce un messaggio di saluto accompagnato da alcune indicazioni sull'utilizzo della chat.
- Prende il dizionario «indirizzi» e lo aggiorna con il dato relativo al nuovo client
- Infine attiva il **Thread**() (*start*()) verso questo client andando ad invocare la funzione «*gestisce_client*»

ATTENZIONE: *start*() deve essere chiamato al massimo una volta per oggetto thread.

Questo metodo genererà un *RuntimeError* se chiamato più di una volta sullo stesso oggetto thread.

Servizio CHAT – Lato Server

{<socket.socket fd=404, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 33000), raddr=('127.0.0.1', 62495)>: 'Name1',
<socket.socket fd=708, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('127.0.0.1', 33000), raddr=('127.0.0.1', 62505)>: 'Name2'}

```
"""La funzione seguente gestisce la connessione di un singolo client."""
def gestice_client(client):  # Prende il socket del client come argomento della funzione.
    nome = client.recv(BUFSIZ).decode("utf8")
    #da il benvenuto al client e gli indica come fare per uscire dalla chat quando ha terminato
    benvenuto = 'Benvenuto %s! Se vuoi lasciare la Chat, scrivi {quit} per uscire.' % nome
    client.send(bytes(benvenuto, "utf8"))
    msg = "%s si è unito all chat!" % nome
    #messaggio in broadcast con cui vengono avvisati tutti i client connessi che l'utente x è entrato
    broadcast(bytes(msg, "utf8"))
    #aggiorna il dizionario clients creato all'inizio
    clients[client] = nome

#si mette in ascolto del thread del singolo client e ne gestisce l'invio dei messaggi o l'uscita dalla Chat
    while True:
        msg = client.recv(BUFSIZ)
        if msg != bytes("{quit}", "utf8"):
            broadcast(msg, nome+": ")
        else:
            client.send(bytes("{quit}", "utf8"))
            client.close()
            del clients[client]
            broadcast(bytes("%s ha abbandonato la Chat." % nome, "utf8"))
            break
```

Definiamo una funzione che si occuperà di gestire il **Thread**() per ciascun client che entra nella Chat. All'interno della funzione in oggetto verrà chiamata la funzione ***broadcast*** per inviare i messaggi a tutti gli utenti.

Servizio CHAT – Lato Server

```
""" La funzione, che segue, invia un messaggio in broadcast a tutti i client."""
def broadcast(msg, prefisso=""):  # il prefisso è usato per l'identificazione del nome.
    for utente in clients:
        utente.send(bytes(prefisso, "utf8")+msg)
```

Questa funzione viene chiamata quando si deve inviare messaggi a tutti gli utenti della Chat.

La funzione «cicla» all'interno del dizionario che contiene «i clients» e invia sul relativo socket il contenuto del messaggio inviato dal client mittente accompagnato dal **prefisso**.

Il **prefisso** e' fornito in ingresso al momento della chiamata della funzione e corrisponde al nome dell'utente che ha inviato il messaggio.

Servizio CHAT – Lato Server

```
if __name__ == "__main__":
```

Questa semplice riga è usata per permettere al nostro codice di capire se verrà eseguito come script, o se è invece sarà invocato come modulo da un qualche programma per usare una o più delle sua funzioni e classi.

- Se eseguiamo questo script direttamente da Shell, ad esempio tramite PyCharm o da riga di comando, allora Python assegna alla variabile __name__ la stringa "__main__". In questo caso la condizione «***if***» risulta TRUE.

- Se invece stiamo invocando lo script come modulo dall'interno di un altro programma, alla variabile __name__ verrà assegnato il nome dello script, quindi il nome del modulo.

*NOTA: Utilizziamo **if __name__ == "__main__"** in tutti quei casi in cui vogliamo poter usare il nostro codice, sia come programma che come modulo.*

Servizio CHAT – Lato Server

```
SERVER.listen(5)
print("In attesa di connessioni...")
ACCEPT_THREAD = Thread(target=accetta_connessioni_in_entrata)
ACCEPT_THREAD.start()
ACCEPT_THREAD.join()
SERVER.close()
```

Il modo più semplice di usare un thread è quello di creare un'istanza con una funzione target e invocare **start ()** per inizializzarlo.

Per attendere fino a quando un thread ha completato il suo lavoro, si utilizza il metodo **join ().** Usiamo il join() con ACCEPT_THREAD in modo che lo script principale attenda il completamento e non salti alla riga successiva, che chiude il server.

Potete trovare tutti i dettagli sui Thread() al seguente link

Lato CLIENT

Servizio CHAT – Lato Client

```
#----Connessione al Server----
HOST = input('Inserire il Server host: ')
PORT = input('Inserire la porta del server host: ')
if not PORT:
    PORT = 53000
else:
    PORT = int(PORT)

BUFSIZ = 1024
ADDR = (HOST, PORT)

client_socket = socket(AF_INET, SOCK_STREAM)
client_socket.connect(ADDR)

receive_thread = Thread(target=receive)
receive_thread.start()
# Avvia l'esecuzione della Finestra Chat.
tkt.mainloop()
```

Una volta ottenuto l'indirizzo e la porta da parte dell'utente e creato un socket per connettersi al server, diamo inizio al thread per la ricezione dei messaggi e quindi al ciclo principale (la GUI).

Servizio CHAT – Lato Client

```
#!/usr/bin/env python3
"""Script relativa alla chat del client utilizzato per lanciare la GUI Tkinter."""
from socket import AF_INET, socket, SOCK_STREAM
from threading import Thread
import tkinter as tkt
```

Per scrivere la GUI, utilizziamo **Tkinter**, uno strumento incluso in Python.

Quindi oltre ad eseguire l'import dei principali moduli (socket e threading), eseguiamo l'import anche di **tkinter**.

Per ulteriori dettagli su Tkinter potete fare riferimento al seguente link

Servizio CHAT – Lato Client

```
"""La funzione che segue ha il compito di gestire la ricezione dei messaggi."""
def receive():
    while True:
        try:
            #quando viene chiamata la funzione receive, si mette in ascolto dei messaggi che
            #arrivano sul socket
            msg = client_socket.recv(BUFSIZ).decode("utf8")
            #visualizziamo l'elenco dei messaggi sullo schermo
            #e facciamo in modo che il cursore sia visibile al termine degli stessi
            msg_list.insert(tkt.END, msg)
            # Nel caso di errore e' probabile che il client abbia abbandonato la chat.
        except OSError:
            break
```

Perché un ciclo infinito?

Vogliamo ricevere messaggi quando siamo connessi e inviarli quando vogliamo. La funzionalità all'interno del loop è piuttosto semplice;

recv () è la parte bloccante. Termina l'esecuzione quando riceve un messaggio, e quindi prosegue aggiungendo il messaggio a ***msg_list***.

msg_list è fondamentalmente una funzionalità di Tkinter per visualizzare l'elenco dei messaggi sullo schermo.

END (o "end") corrisponde alla posizione del cursore subito dopo l'ultimo carattere nel buffer.

Servizio CHAT – Lato Client

```
"""La funzione che segue gestisce l'invio dei messaggi."""
def send(event=None):
    # gli eventi vengono passati dai binders.
    msg = my_msg.get()
    # libera la casella di input.
    my_msg.set("")
    # invia il messaggio sul socket
    client_socket.send(bytes(msg, "utf8"))
    if msg == "{quit}":
        client_socket.close()
        finestra.quit()
```

Stiamo usando ***event*** come argomento perché viene implicitamente passato da Tkinter quando viene **premuto il pulsante di invio sulla GUI**.

my_msg è il campo di input sulla GUI e quindi estraiamo il messaggio da inviare associandolo alla variabile:***msg*** = ***my_msg.get ()***.

Successivamente, cancelliamo il campo di input con ***my_msg.set*** e quindi inviamo il messaggio al server, che, come abbiamo visto in precedenza, trasmette questo messaggio a tutti i client (se non è un messaggio «quit»). Se si tratta di un messaggio «quit», chiudiamo il socket e quindi l'app GUI (tramite top.quit ())

Servizio CHAT – Lato Client

```
"""La funzione che segue viene invocata quando viene chiusa la finestra della chat."""
def on_closing(event=None):
    my_msg.set("{quit}")
    send()
```

Definiamo un'altra funzione, che verrà chiamata quando scegliamo di chiudere la finestra della GUI.

È una sorta di funzione di pulizia prima della chiusura e deve chiudere la connessione socket prima della chiusura della GUI.

NOTA:

Se si vuole semplicemente uscire dal programma, è possibile usare ***destroy***().

Se invece si vuole eseguire un ciclo infinito senza distruggere la finestra Tkinter allora si usa ***quit***()

Servizio CHAT – Lato Client

```
finestra = tkt.Tk()
finestra.title("Chat_Laboratorio")
```

Ora iniziamo a creare la GUI, nel Main Space (cioè all'esterno di qualsiasi funzione).

Iniziamo definendo il widget di livello superiore e impostando il suo titolo:

«**Chat_Laboratorio**»

Servizio CHAT – Lato Client

```
#creiamo il Frame per contenere i messaggi
messages_frame = tkt.Frame(finestra)
#creiamo una variabile di tipo stringa per i messaggi da inviare.
my_msg = tkt.StringVar()
#indichiamo all'utente dove deve scrivere i suoi messaggi
my_msg.set("Scrivi qui i tuoi messaggi.")
#creiamo una scrollbar per navigare tra i messaggi precedenti.
scrollbar = tkt.Scrollbar(messages_frame)
```

Quindi creiamo un frame per contenere l'elenco dei messaggi.

Successivamente, creiamo una variabile stringa, principalmente per memorizzare il valore che otteniamo dal campo di input.

Abbiamo impostato il valore iniziale di quella variabile a «**Scrivi qui i tuoi messaggi**» per richiedere all'utente di scrivere il proprio messaggio in quello spazio.

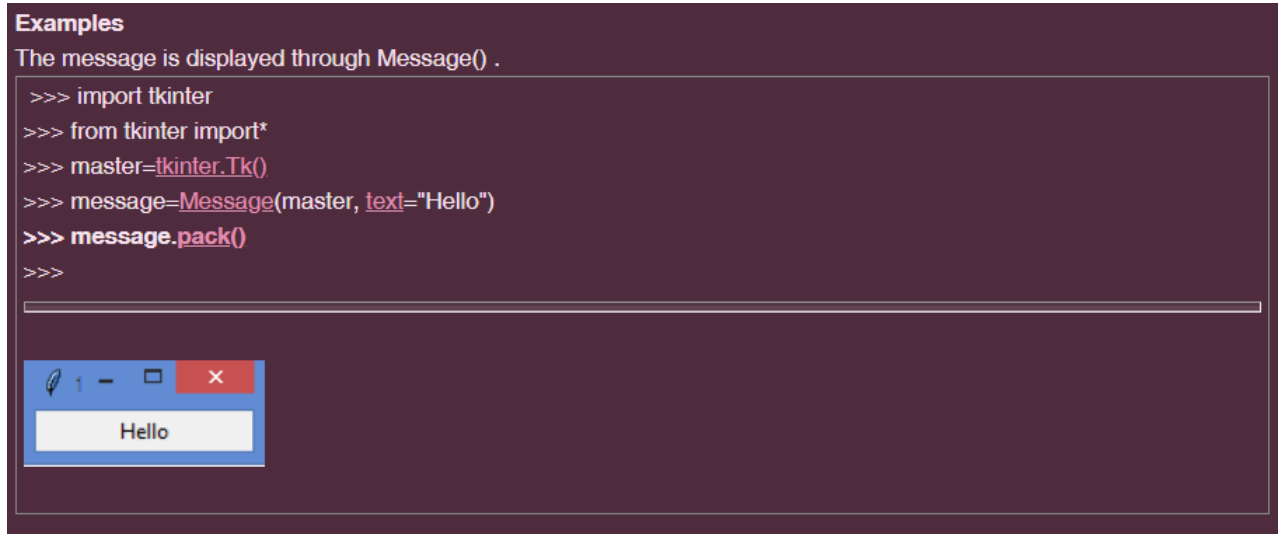
Successivamente, creiamo una barra di scorrimento per scorrere questo frame di messaggi.

Servizio CHAT – Lato Client

```
# La parte seguente contiene i messaggi.
msg_list = tkt.Listbox(messages_frame, height=15, width=50, yscrollcommand=scrollbar.set)
scrollbar.pack(side=tkt.RIGHT, fill=tkt.Y)
msg_list.pack(side=tkt.LEFT, fill=tkt.BOTH)
msg_list.pack()
messages_frame.pack()
```

Ora definiamo l'elenco dei messaggi che verrà archiviato in ***message_frame*** e quindi impacchettato.

Potete trovare indicazioni sulla costruzione e gestione del Frame al seguente link.



Servizio CHAT – Lato Client

```
#Creiamo il campo di input e lo associamo alla variabile stringa
entry_field = tkt.Entry(finestra, textvariable=my_msg)
# leghiamo la funzione send al tasto Return
entry_field.bind("<Return>", send)

entry_field.pack()
#creiamo il tasto invio e lo associamo alla funzione send
send_button = tkt.Button(finestra, text="Invio", command=send)
#integriamo il tasto nel pacchetto
send_button.pack()

finestra.protocol("WM_DELETE_WINDOW", on_closing)
```

- Creiamo il campo di input affinché l'utente possa inserire il proprio messaggio e lo associamo alla variabile stringa definita prima.
- Associamo il tasto RETURN alla funzione **send** () in modo che ogni volta che l'utente preme return, il messaggio viene inviato al server.
- Creiamo il pulsante di «**Invio**» nel caso in cui l'utente desideri inviare i propri messaggi facendo clic su di esso. Associamo questo pulsante alla funzione **send** ().
- Infine utilizziamo la funzione di pulizia **on_closing** () che dovrebbe essere chiamata quando l'utente desidera chiudere la finestra della GUI.

Servizio CHAT – Lato Client

Se infine intendete creare un file .exe eseguibile potete utilizzare, dopo averlo installato il pacchetto

PYINSTALLER

(da shell di DOS: *python -m pip install pyinstaller)*

In questo modo se fornite a Pyinstaller in ingresso il file relativo al client CHAT, potete creare un eseguibile da condividere con altre persone.

Esercizi

Esercizi – Livello di Applicazione e Livello di Trasporto

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Esercizi - Livello di Applicazione

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

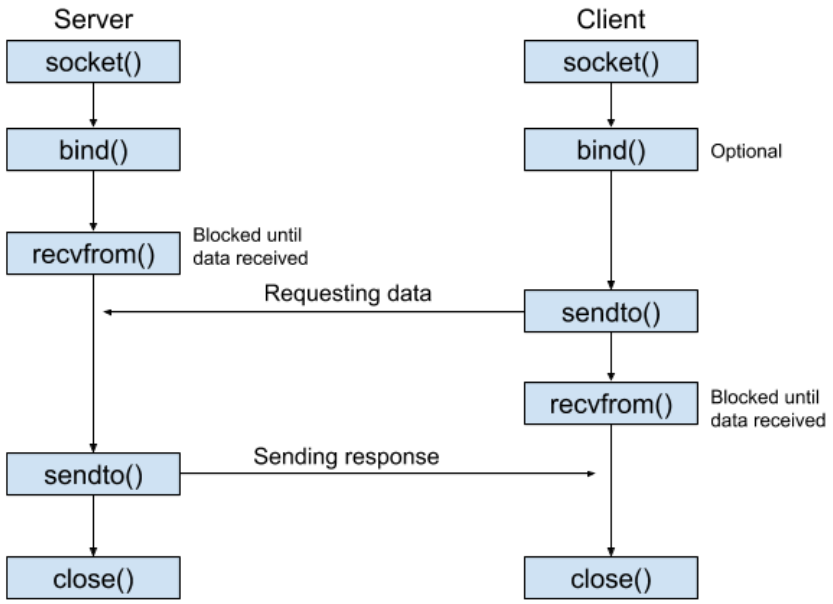
Esercizi - Livello di Applicazione

R26: Il server UDP, visto a lezione, richiedeva una sola socket, mentre il server TCP ne richiedeva due. Perché? Se il server TCP dovesse supportare n connessioni contemporanee, ciascuna proveniente da un diverso client, di quante socket avrebbe bisogno?

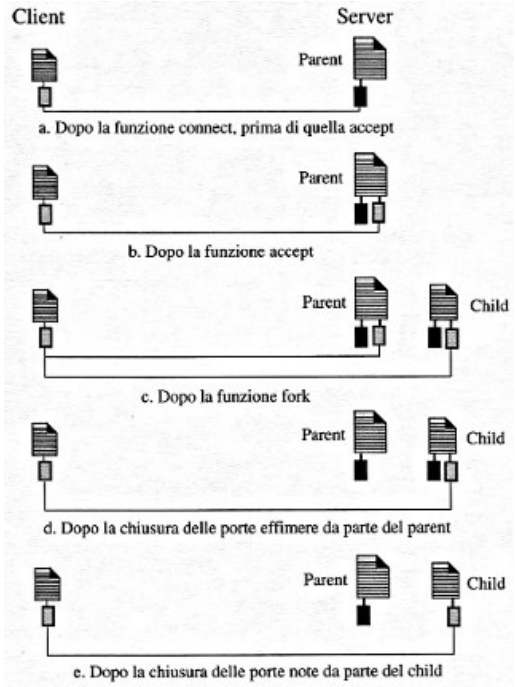
Soluzione Esercizio – R26

Con il server UDP, non esiste un socket di benvenuto e tutti i dati provenienti da client diversi entrano nel server attraverso questo socket.

Con il server TCP, esiste un socket di benvenuto e ogni volta che un client avvia una connessione al server, viene creato un nuovo socket. Quindi, per supportare ***n*** connessioni simultanee, il server avrebbe bisogno di ***n+1*** socket.



Socket UDP



Socket TCP

Esercizi - Livello di Applicazione

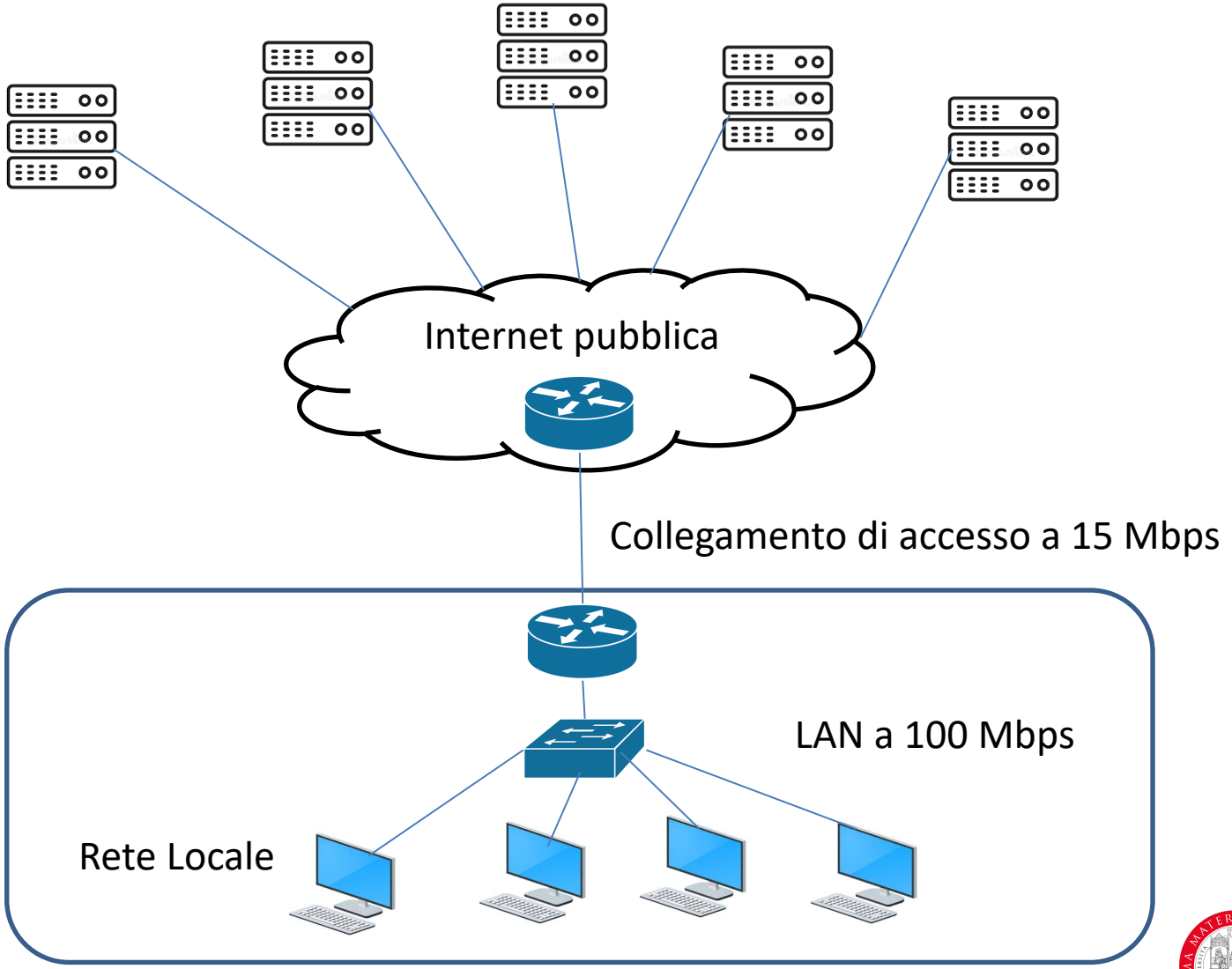
P9: Si consideri la figura riportata qui di seguito, in cui si vede la rete di un'istituzione connessa a Internet. Supponete che la dimensione media degli oggetti sia di 850,000 bit e che la frequenza media di richieste dai browser dell'istituzione verso i server di origine sia di 16 richieste al secondo. Ipotizzate, inoltre, che la quantità di tempo che intercorre da quando il router sul lato Internet del collegamento di accesso inoltra una richiesta HTTP a quando riceve la risposta sia mediamente di **tre** secondi.

Fate un modello del tempo medio totale di risposta come la somma del ritardo medio di accesso (ossia, il ritardo dal router Internet al router dell'istituzione) e del ritardo Internet medio. Per il ritardo medio di accesso, usate la formula $\frac{\Delta}{(1-\Delta\beta)}$, dove Δ è il tempo medio richiesto per inviare un oggetto sul collegamento di accesso e β è la frequenza di arrivo di oggetti al collegamento di accesso.

a. Trovate il tempo medio totale di risposta
b. Supponete ora che nella LAN dell'istituzione sia installato un proxy e che la frequenza con cui gli oggetti non sono in cache (*miss rate*) sia 0.4. Trovate il tempo totale di risposta.

Esercizi - Livello di Applicazione

- dimensione media degli oggetti = 850,000 bit
- frequenza media di richieste dai browser dell'istituzione verso i server di origine sia di 16 richieste al secondo
- quantità di tempo che intercorre da quando il router sul lato Internet del collegamento di accesso inoltra una richiesta HTTP a quando riceve la risposta sia mediamente di tre secondi.
- Collegamento di accesso a 15 Mbps



Soluzione Esercizio P9.a

a. Il tempo per trasmettere un oggetto di dimensione L su un collegamento di velocità R è $\frac{L}{R}$. Il tempo medio è la dimensione media dell'oggetto divisa per R ($= 15Mbps$ si veda il disegno):

$$\Delta = \frac{850{,}000\ bits}{15{,}000{,}000\ bits/sec} = 0.0567\ sec$$

L'intensità del traffico sul collegamento è data da

$$\beta\Delta = (16\ richieste/sec) \cdot (0.0567\ sec/richiesta) = 0.907$$

Pertanto, il ritardo medio di accesso è $\frac{\Delta}{(1-\Delta\beta)} = \frac{0.0567\ sec}{(1-0.907)} = 0.6\ sec$

Il tempo di risposta medio totale è quindi

$$T_{medio\ totale\ di\ risposta} = 0.6\ sec + 3\ sec = 3.6\ sec$$

Soluzione Esercizio P9.b

b. L'intensità di traffico sul collegamento di accesso è ridotta del 60% in quanto il 60% delle richieste viene soddisfatta all'interno della rete istituzionale.

Pertanto il ritardo di accesso medio è

$$\frac{\Delta}{(1 - 0.4 \cdot \Delta\beta)} = \frac{0.0567\ sec}{(1 - 0.4\ \cdot 0.907)} = 0.089\ sec$$

Il tempo di risposta è approssimativamente zero se la richiesta è soddisfatta dalla cache (cosa che avviene con probabilità 0.6);

il tempo di risposta medio è di $0.089 + 3 = 3.089\ sec$ per cache miss (che si verifica il 40% delle volte).

Quindi il tempo medio di risposta è $(0.6) \cdot (0\ sec) + (0.4) \cdot (3.089\ sec) = 1.24\ sec.$

Così il tempo medio di risposta è ridotto da 3,6 sec a 1,24 sec.

Esercizi - Livello di Applicazione

P10: Considerate un breve collegamento di 10 metri, sul quale il mittente può trasmettere a una frequenza di 150 bps in entrambe le direzioni. Supponete che i pacchetti dati siano lunghi 100,000 bit e quelli contenenti solo controllo (per esempio ACK o handshake) siano lunghi 200 bit.

Assumete che vi siano N connessioni parallele e ciascuna occupi $\frac{1}{N}$ della banda del collegamento.

Considerate ora il protocollo HTTP e supponete che ciascun oggetto scaricato sia lungo 100 kbit e che l'oggetto iniziale scaricato contenga 10 oggetti referenziati dallo stesso mittente.

Ha senso lo scaricamento parallelo tramite istanze parallele non persistenti di HTTP?

Considerate ora HTTP persistente. Vi aspettate un guadagno significativo rispetto al caso non persistente? Giustificate e spiegate la vostra risposta.

Soluzione Esercizio P10

Si noti che ogni oggetto scaricato (100Kbit) può essere completamente inserito in un pacchetto di dati (100,000 bit).

Sia T_p il ritardo di propagazione unidirezionale tra il client e il server.

Considera innanzitutto **i download paralleli che utilizzano connessioni non persistenti**. I download paralleli consentirebbero a 10 connessioni di condividere la larghezza di banda di 150 bit/sec, dando a ciascuna solo 15 bit/sec. Pertanto, il tempo totale necessario per ricevere tutti gli oggetti è dato da:

$$\left(\frac{200}{150}+T_p+\frac{200}{150}+T_p+\frac{200}{150}+T_p+\frac{100{,}000}{150}+T_p\right)+\left(\frac{200}{150/10}+T_p+\frac{200}{150/10}+T_p+\frac{200}{150/10}+T_p+\frac{100{,}000}{150/10}+T_p\right)=7377+8\cdot T_p\ sec$$

syn — ack — Syn ack — Pagina html — 10 syn In parallelo — 10 ack In parallelo — 10 syn ack In parallelo — 10 oggetti In parallelo

Connessioni TCP NON Persistenti

$$1.33+1.33+1.33+666.67+13.3+13.3+13.3+6666{,}67+8\cdot T_p=7377.27+8\cdot T_p$$

Soluzione Esercizio P10

Consideriamo ora **una connessione HTTP persistente**. Il tempo totale necessario è dato da:

$$\left(\frac{200}{150}+T_p+\frac{200}{150}+T_p+\frac{200}{150}+T_p+\frac{100{,}000}{150}+T_p\right)+10\cdot\left(\frac{200}{150}+T_p+\frac{100{,}000}{150}+T_p\right)=7351+24\cdot T_p\ sec$$

syn — ack — Syn ack — Pagina html — Richiesta oggetto — Invio oggetto

Connessione TCP Persistente

$$1.33+1.33+1.33+666.67+13.3+6666.67+24\cdot T_p=7350.67+24\cdot T_p$$

Supponendo che la velocità della luce sia $300\cdot 10^6 m/s$, allora $T_p=\frac{10\ m}{300\cdot 10^6 m/s}=0.03\cdot 10^{-6} sec=0.03\mu s$.

T_p è quindi trascurabile rispetto al ritardo di trasmissione.

Pertanto, vediamo che l'HTTP persistente non è significativamente più veloce (meno dell'1%) rispetto al caso non persistente con download parallelo.

Esercizi - Livello di Applicazione

P22: Considerate la distribuzione di un file di dimensione $F = 15\ Gbit$ a N peer.

Il server ha una velocità di upload di $u_s = 30\ Mbps$ e ciascun peer ha una banda di download di $d_i = 2\ Mbps$ e di upload u.

Per $N = 10, 100, 1000$ e $u = 300\ kbps, 700\ kbps\ e\ 2Mbps$, preparate un grafico o una tabella che rappresenti per entrambi i metodi di distribuzione, client-server e P2P, il tempo di distribuzione minimo per ciascuna combinazione di $N\ e\ u$.

DEFINIZIONE: Il tempo di distribuzione è il tempo necessario perché tutti gli N peer ottengano una copia del file.

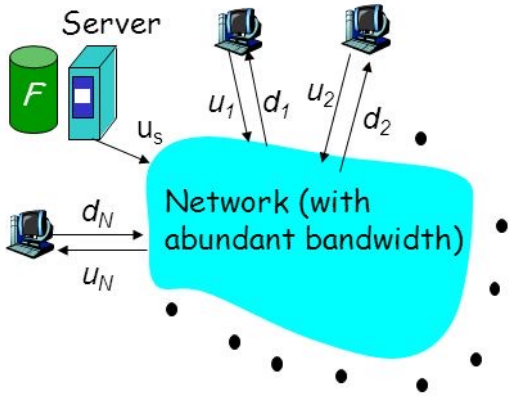
Soluzione Esercizio P22

Per calcolare il tempo minimo di distribuzione per la distribuzione client-server, utilizziamo la seguente formula:

$$D_{cs} = \max\left\{{}^{NF}/_{u_s}, {}^{F}/_{d_{min}}\right\}$$

File distribution time: server-client

- server sequentially sends N copies:
  - NF/u_s time
- client i takes F/d_i time to download



Time to distribute F to N clients using client/server approach = $d_{cs} = \max\{NF/u_s, F/\min_i(d_i)\}$

increases linearly w.r.t. N (for large N)

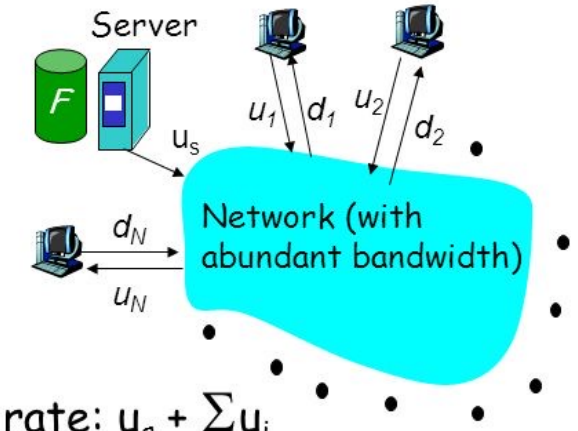
Soluzione Esercizio P22

Allo stesso modo, per calcolare il tempo minimo di distribuzione per la distribuzione P2P, utilizziamo la seguente formula:

$$D_{P2P} = \max\left\{ {}^{F}/_{u_s}, {}^{F}/_{d_{min}}, {}^{NF}\Big/_{\left(u_s + \sum_{i=1}^{N} u_i\right)} \right\}$$

File distribution time: P2P

- server must send one copy: F/u_s time
- client i takes F/d_i time to download
- NF bits must be downloaded (aggregate)
  - fastest possible upload rate: $u_s + \Sigma u_i$



$$d_{P2P} = \max\{ F/u_s, F/\min_i(d_i), NF/(u_s + \Sigma u_i)\}$$

Nell'architettura P2P, notiamo che:

- all'inizio della distribuzione, solo il server contiene il file. Per inviarlo alla comunità dei peer, deve effettuare l'upload almeno una volta attraverso il proprio link di accesso, quindi il minimo tempo di distribuzione lato server è almeno **F/u_s.** A differenza dell'architettura client-server, infatti, può non essere necessario più di un invio da parte del server, dato che i peer possono ridistribuire il file tra loro autonomamente
- come nell'architettura client-server, il peer con la connessione più lenta non può ricevere in un tempo minore di **F/d_{min}**
- la capacità totale di upload del sistema nel suo insieme è pari alla velocità di upload del server, più le velocità di upload di ogni singolo peer, ossia **$u_{tot} = u_s + u_1 + u_2 + ... + u_n$**. Poiché il sistema deve trasmettere globalmente NF bit, il tempo minimo di distribuzione non può essere inferiore a **$NF/(u_s + u_1 + u_2 + ... + u_n)$**

Soluzione Esercizio P22

Dove:

$F = 15Gbits = 15 * 1024\ Mbits$

$u_s = 30Mbps$

$d_{min} = d_i = 2Mbps$

Si noti inoltre che

$\mathbf{300\ Kbps = \frac{300}{1024} Mbps}$

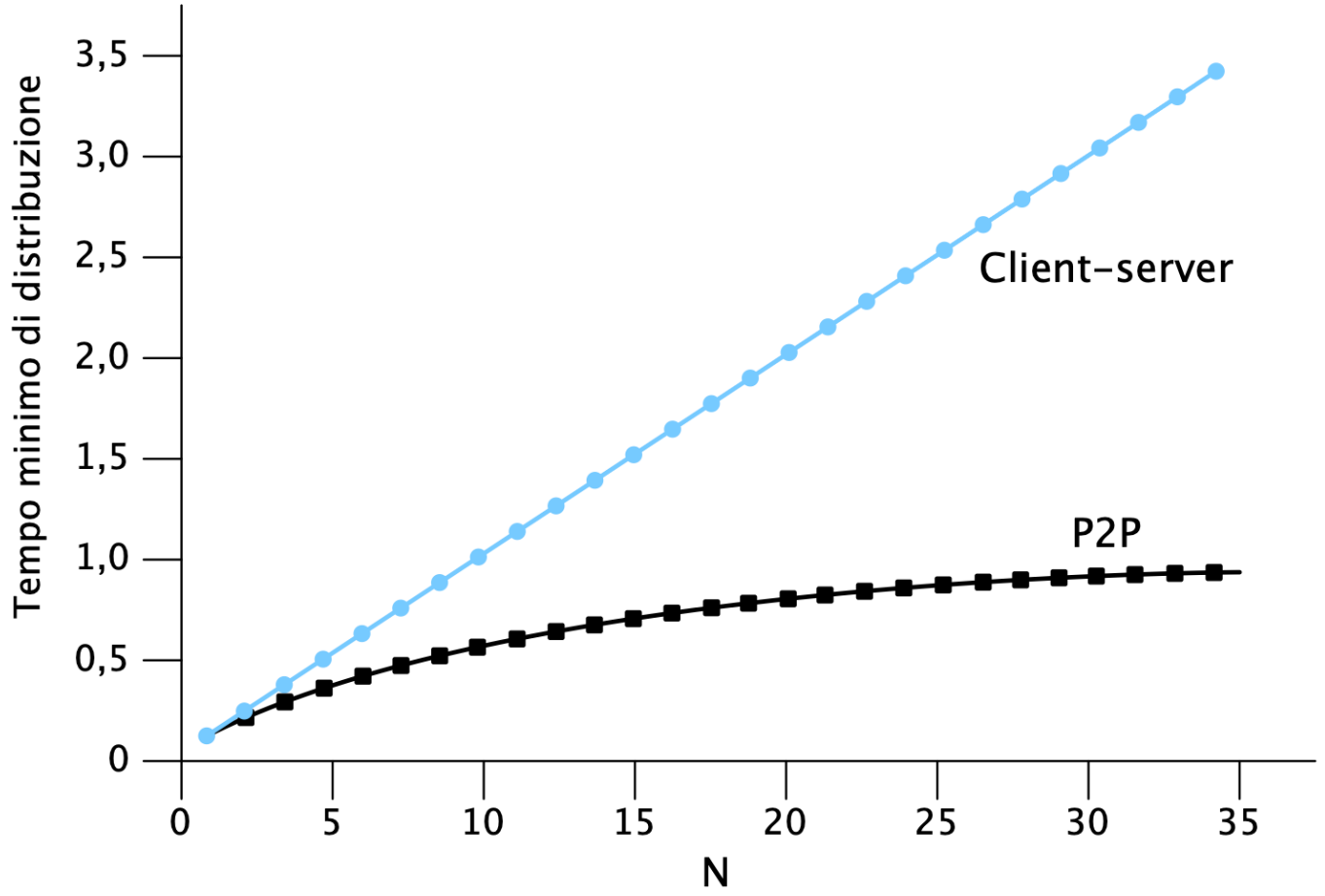
Client - Server

		N		
		10	100	1000
u	300 Kbps	7680	51200	512000
	700Kbps	7680	51200	512000
	2 Mbps	7680	51200	512000

Peer - Peer

		N		
		10	100	1000
u	300 Kbps	7680	25904	47559
	700Kbps	7680	15616	21525
	2 Mbps	7680	7680	7680

Soluzione Esercizio P22



Esercizi - Livello di Trasporto

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

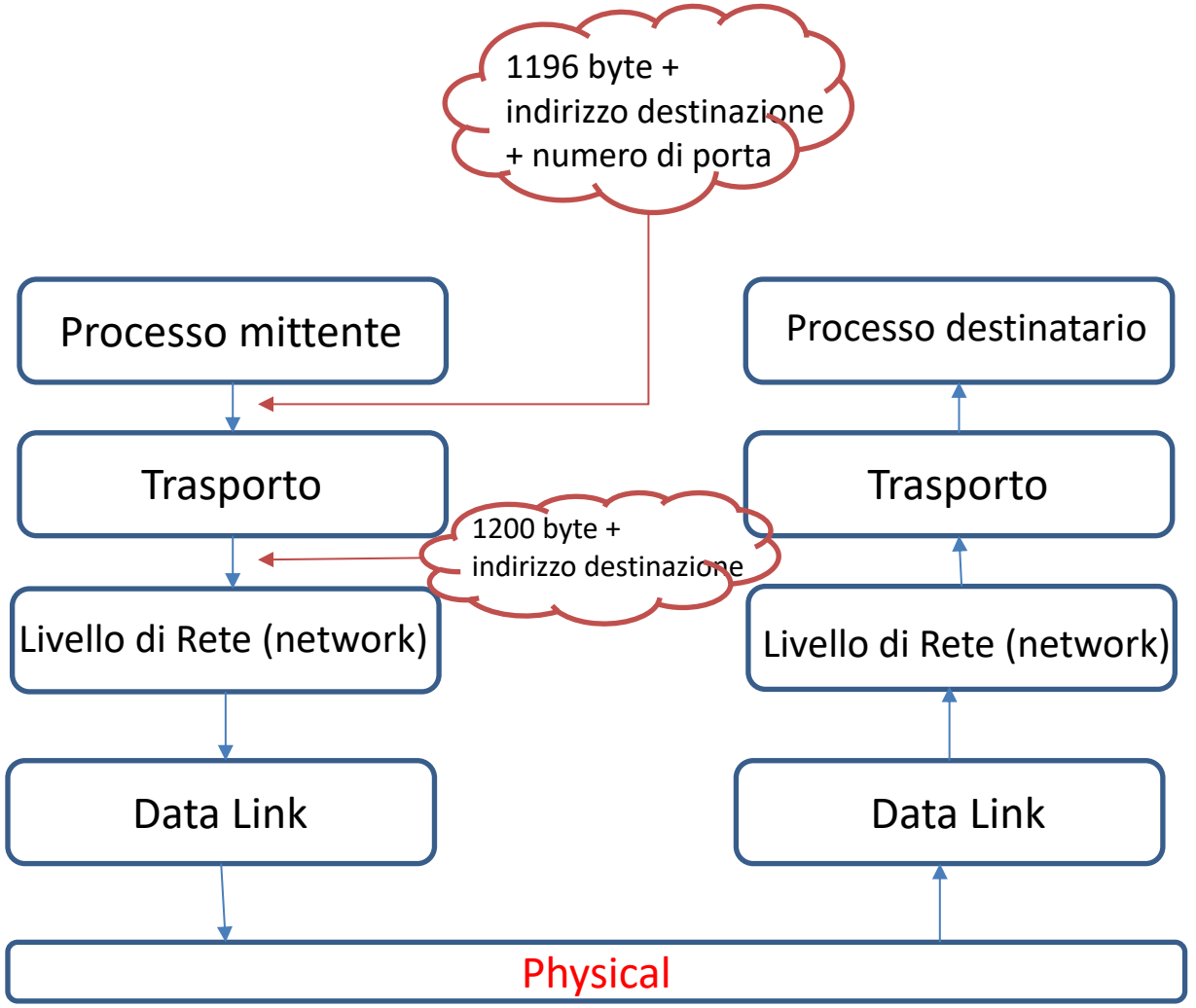
Esercizi - Livello di Trasporto

R1: Supponete che il livello di rete fornisca il seguente servizio. Il livello di rete nell'host sorgente accetta un segmento di dimensione massima di 1200 byte e un indirizzo dell'host di destinazione dal livello di trasporto. Il livello di rete poi garantisce di consegnare il segmento al livello di trasporto nell'host di destinazione. Supponete che molti processi applicativi di rete possano essere in esecuzione sull'host di destinazione.

a. Progettate un protocollo a livello di trasporto più semplice possibile che porti i dati applicativi al processo desiderato dell'host di destinazione. Assumete che il sistema operativo nell'host di destinazione abbia assegnato un numero di porta di 4 byte a ciascun processo applicativo in esecuzione.
b. Modificate questo protocollo in modo che fornisca un «indirizzo di ritorno» al processo di destinazione.
c. Nei vostri protocolli il livello di trasporto deve fare qualcosa nel nucleo della rete di calcolatori?

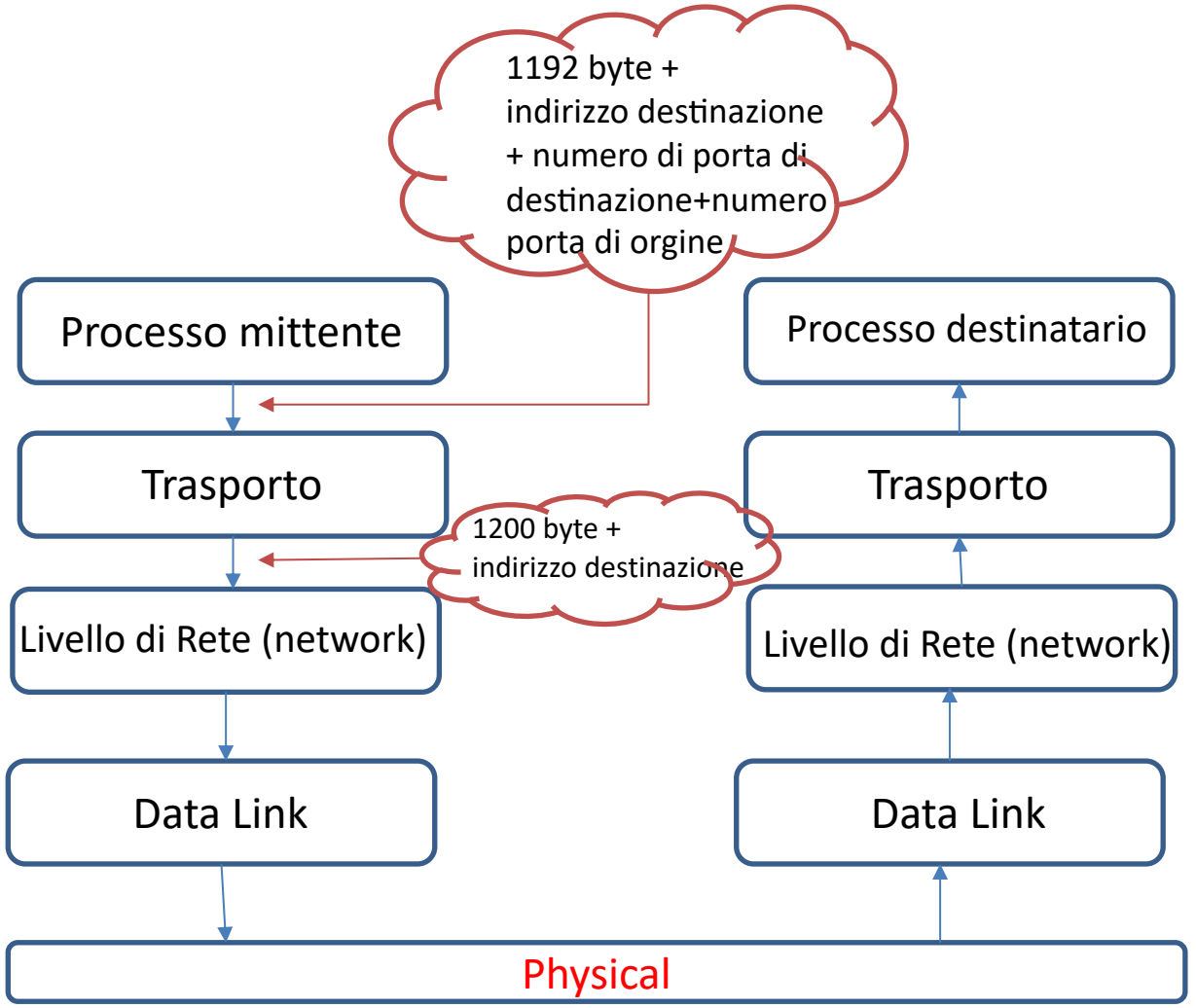
Soluzione Esercizio R1

a. Chiamiamo questo protocollo Simple Transport Protocol (STP). Dal lato del mittente, STP accetta dal processo mittente un blocco di dati non superiore a 1196 byte, un indirizzo host di destinazione e un numero di porta di destinazione. STP aggiunge un'intestazione di 4 byte a ciascun blocco e inserisce il numero di porta del processo di destinazione in questa intestazione. STP fornisce quindi l'indirizzo host di destinazione e il segmento risultante al livello di rete. Il livello di rete consegna il segmento a STP nell'host di destinazione. STP quindi esamina il numero di porta nel segmento, estrae i dati dal segmento e passa i dati al processo identificato dal numero di porta.



Soluzione Esercizio R1

b. Il segmento ha ora due campi di intestazione: un campo della porta di origine e un campo della porta di destinazione. Sul lato mittente, STP accetta un blocco di dati non superiore a 1192 byte, un indirizzo host di destinazione, un numero di porta di origine e un numero di porta di destinazione. STP crea un segmento che contiene i dati dell'applicazione, il numero della porta di origine e il numero della porta di destinazione. Fornisce quindi il segmento e l'indirizzo host di destinazione al livello di rete. Dopo aver ricevuto il segmento, STP presso l'host ricevente fornisce al processo dell'applicazione i dati dell'applicazione e il numero della porta di origine.



Soluzione Esercizio R1

c. No, il livello di trasporto non deve fare nulla nel core; lo strato di trasporto “vive” negli end systems.

Esercizi - Livello di Trasporto

R8: Supponete che un web server sia in esecuzione sull'host C sulla porta 80.

Supponete che questo web server usi le connessioni persistenti e stia al momento ricevendo richieste da due diversi host, A e B.

Tutte le richieste vengono inviate attraverso la stessa socket sull'host C?

Se vengono fatte passare attraverso socket diverse, entrambe hanno la porta 80? Discutete e spiegate questa situazione.

Soluzione Esercizio R8

Per ogni connessione persistente, il server Web crea un "socket di connessione" separato.

Ogni socket di connessione è identificato da una quadrupla: (indirizzo IP di origine, numero di porta di origine, indirizzo IP di destinazione, numero di porta di destinazione).

Quando l'host C riceve un datagramma IP, esamina questi quattro campi nel datagramma/segmento per determinare a quale socket deve passare il payload del segmento TCP.

Pertanto, le richieste di A e B passano attraverso socket diversi.

L'identificatore per entrambi questi socket ha 80 per la porta di destinazione; tuttavia, gli identificatori per questi socket hanno valori diversi per gli indirizzi IP di origine.

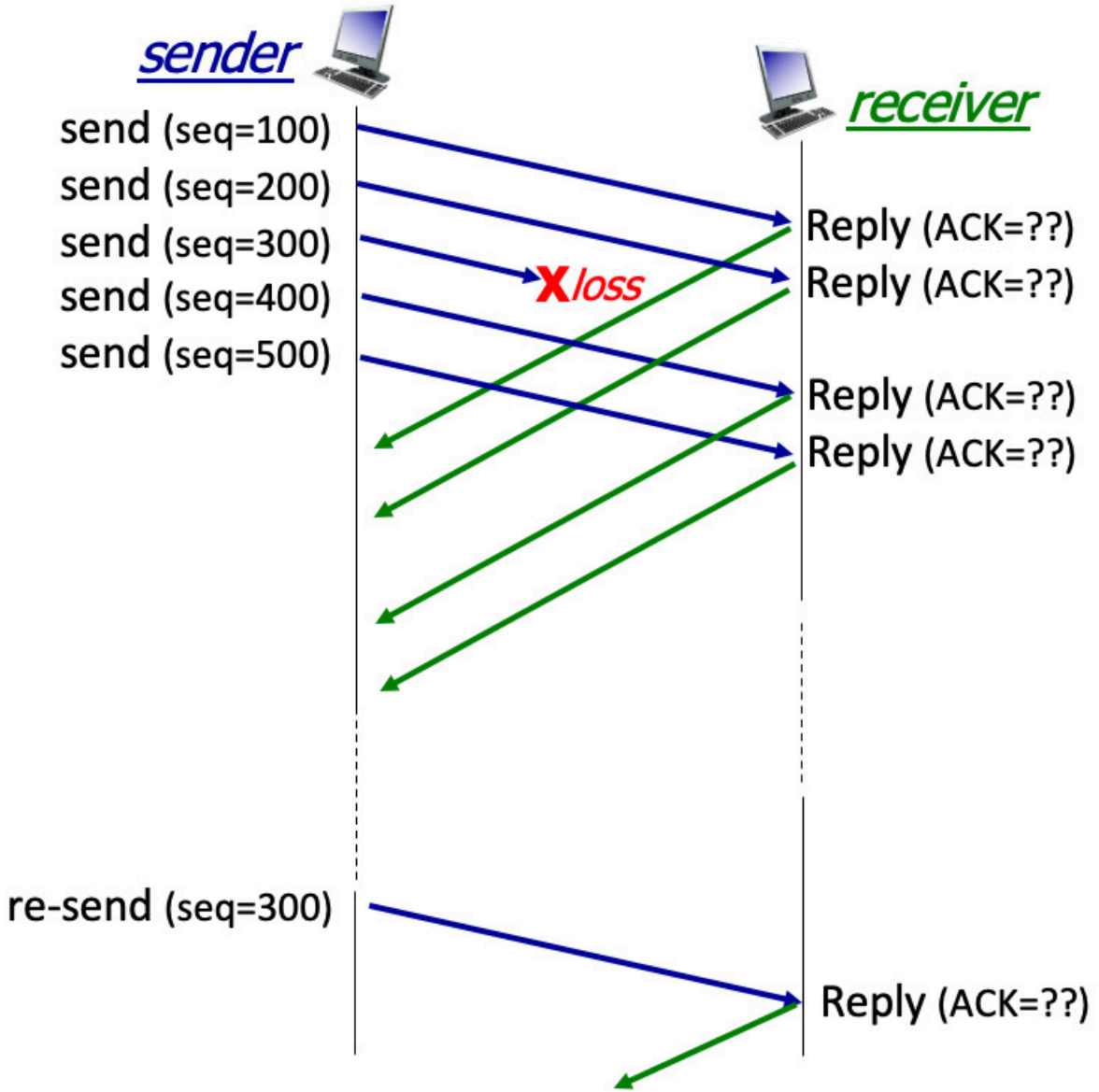
A differenza di UDP, quando il livello di trasporto passa il carico utile di un segmento TCP al processo dell'applicazione, non specifica l'indirizzo IP di origine, poiché questo è implicitamente specificato dall'identificatore del socket.

Esercizi - Livello di Trasporto

R9: Nei nostri protocolli *rdt*, perché abbiamo bisogno di introdurre i numeri di sequenza?

Soluzione Esercizio R9

I numeri di sequenza sono necessari a un destinatario per scoprire se un pacchetto in arrivo contiene nuovi dati o è una ritrasmissione.



Esercizi - Livello di Trasporto

P3: UDP e TCP utilizzano il complemento a 1 per calcolare il checksum. Supponiamo di avere i seguenti tre byte: 01010011, 01100110 e 01110100.

Qual è il complemento a 1 della loro somma?

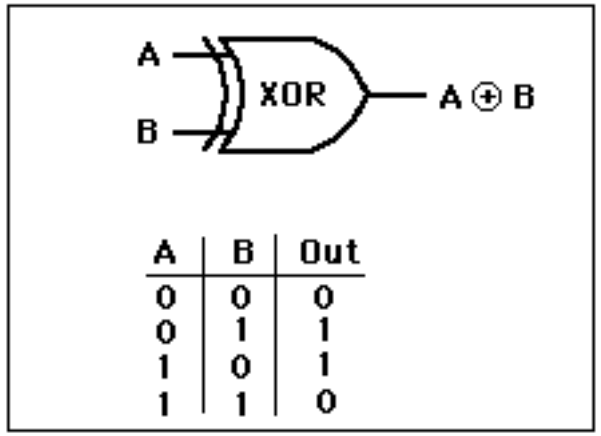
Notiamo che, sebbene UDP e TCP usino checksum da 16 bit, in questo problema consideriamo addendi a 8 bit.

Con lo schema del complemento a 1, come vengono rilevati gli errori dal destinatario?

E' possibile che un errore su 1 bit non venga rilevato?

E che cosa avviene per gli errori su 2 bit?

Soluzione Esercizio P3



A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

Avvolgere in caso di overflow

```
  0 1 0 1 0 0 1 1
+ 0 1 1 0 0 1 1 0
  1 0 1 1 1 0 0 1
```

```
  1 0 1 1 1 0 0 1
+ 0 1 1 1 0 1 0 0
  0 0 1 0 1 1 0 1
```

Riporto di 1

```
+ 0 0 1 0 1 1 0 1
  0 0 1 0 1 1 1 0
```

Il Complemento a uno = 1 1 0 1 0 0 0 1

```
  0 1 0 1 0 0 1 0
+ 0 1 1 0 0 1 1 1
  1 0 1 1 1 0 0 1
```

Per rilevare gli errori, il ricevitore aggiunge le quattro parole (le tre parole originali e il checksum). Se la somma contiene uno zero, il ricevente sa che c'è stato un errore. Verranno rilevati tutti gli errori a un bit, ma gli errori a due bit possono non essere rilevati (ad esempio, se l'ultima cifra della prima parola viene convertita in uno 0 e l'ultima cifra della seconda parola viene convertita in un 1).

Esercizi - Livello di Trasporto

P26: Considerate il trasferimento di un file enorme di L byte dall'Host A all'Host B. Ipotizzate un MSS pari a 536 byte.

a. Quale è il valore massimo di L tale per cui i numeri di sequenza TCP non vengono esauriti? Ricordiamo che il campo numero di sequenza in TCP è composto da quattro byte.

b. Per il valore di L ottenuto in (a) determinate quanto impiegherebbe la trasmissione del file. Ipotizzate l'aggiunta di un numero totale di 66 byte per le intestazioni di trasporto, di rete e di collegamento a ciascun segmento prima che il pacchetto risultante venga immesso su un collegamento da 155 Mbps. Ignorate il controllo di flusso e di congestione di modo che A possa immettere i segmenti end-to-end e con continuità.

Soluzione Esercizio P26.a

a. Quale è il valore massimo di L tale per cui i numeri di sequenza TCP non vengono esauriti? Ricordiamo che il campo numero di sequenza in TCP è composto da quattro byte (32 bit).

Ci sono $2^{32} = 4.294.967.296$ possibili numeri di sequenza.

Il numero di sequenza non aumenta di uno con ciascun segmento.
Piuttosto, esso incrementa del numero di byte di dati inviati.

Se inviamo 10 byte il numero di sequenza successivo sarà quello precedente +10.

Quindi la dimensione dell'MSS è irrilevante:

la dimensione massima del file che può essere inviata da A a B è semplicemente il numero di byte rappresentabile da $2^{32} \approx 4.19\ Gbyte$.

Soluzione Esercizio P26.b

b. Per il valore di L ottenuto in (a) determinate quanto impiegherebbe la trasmissione del file. Ipotizzate l'aggiunta di un numero totale di 66 byte per le intestazioni di trasporto, di rete e di collegamento a ciascun segmento prima che il pacchetto risultante venga immesso su un collegamento da 155 Mbps. Ignorate il controllo di flusso e di congestione di modo che A possa immettere i segmenti end-to-end e con continuità.

Il numero di segmenti è $\left\lceil \frac{2^{32}}{536} \right\rceil = 8{,}012{,}999.$

66 byte di intestazione vengono aggiunti a ciascun segmento per un totale di

$66 \cdot 8{,}012{,}999 = 528{,}857{,}934$ byte di intestazione.

Il numero totale di byte trasmessi è

$$2^{32} + 528{,}857{,}934 = 4{,}823{,}825{,}230 \cong 4.824 \cdot 10^9 bytes$$

Pertanto occorrerebbero $\frac{4.824 \cdot 10^9 \cdot 8\ bits}{155 \cdot 10^6 bits/sec} \approx 249\ sec$

per trasmettere il file su un collegamento a 155~Mbps.

Esercizi - Livello di Trasporto

P37: Paragonate Go-Back-N, Selective Repeat, e TCP (senza il delayed ACK). Assumete che i valori di timeout per tutti questi tre protocolli siano sufficientemente grandi per cui 5 segmenti di dati consecutivi e i corrispondenti ACK possano essere ricevuti (se non persi nel canale) dall'host ricevente (Host B) e dall'host mittente (Host A) rispettivamente. Supponete che l'Host A invii cinque segmenti di dati all'HOST B e che il secondo segmento (inviato da A) venga perso. Alla fine, tutti e 5 i segmenti di dati sono stati ricevuti correttamente dall'Host B.

a. Quanti segmenti ha inviato l'Host A in tutto e quanti ACK ha mandato l'Host B in tutto? Quali sono i loro numeri di sequenza? Rispondete a queste domande per tutti e tre i protocolli.
b. Se i valori di timeout per tutti e tre i protocolli sono molto più grandi di 5 RTT, quale protocollo consegnerà con successo tutti e cinque i segmenti di dati nell'intervallo di tempo più breve?

Soluzione Esercizio P37.a

a. Quanti segmenti ha inviato l'Host A in tutto e quanti ACK ha mandato l'Host B in tutto? Quali sono i loro numeri di sequenza? Rispondete a queste domande per tutti e tre i protocolli.

Go-Back-N:
A invia 9 segmenti in totale. Vengono inizialmente inviati i segmenti 1, 2, 3, 4, 5 e successivamente i segmenti 2, 3, 4 e 5.
B invia 8 ACK. Sono 4 ACKS con numero di sequenza 1 ,e 4 ACKS con numeri di sequenza 2, 3, 4 e 5.

Selective Repeat:
A invia 6 segmenti in totale. Inizialmente vengono inviati i segmenti 1, 2, 3, 4, 5 e successivamente il segmento 2.
B invia 5 ACK. Sono 4 ACK con numero di sequenza 1, 3, 4, 5. E c'è un ACK con numero di sequenza 2.

TCP:
A invia 6 segmenti in totale. Inizialmente vengono inviati i segmenti 1, 2, 3, 4, 5 e successivamente il segmento 2.
B invia 5 ACK. Sono 4 ACK con numero di sequenza 2. C'è un ACK con numero di sequenza 6. Si noti che TCP invia sempre un ACK con numero di sequenza previsto.

Soluzione Esercizio P37.b

b. Se i valori di timeout per tutti e tre i protocolli sono molto più grandi di 5 RTT, quale protocollo consegnerà con successo tutti e cinque i segmenti di dati nell'intervallo di tempo più breve?

TCP.
Questo perché TCP utilizza la ritrasmissione rapida senza attendere il timeout.