Network Programming

Java Programming Language

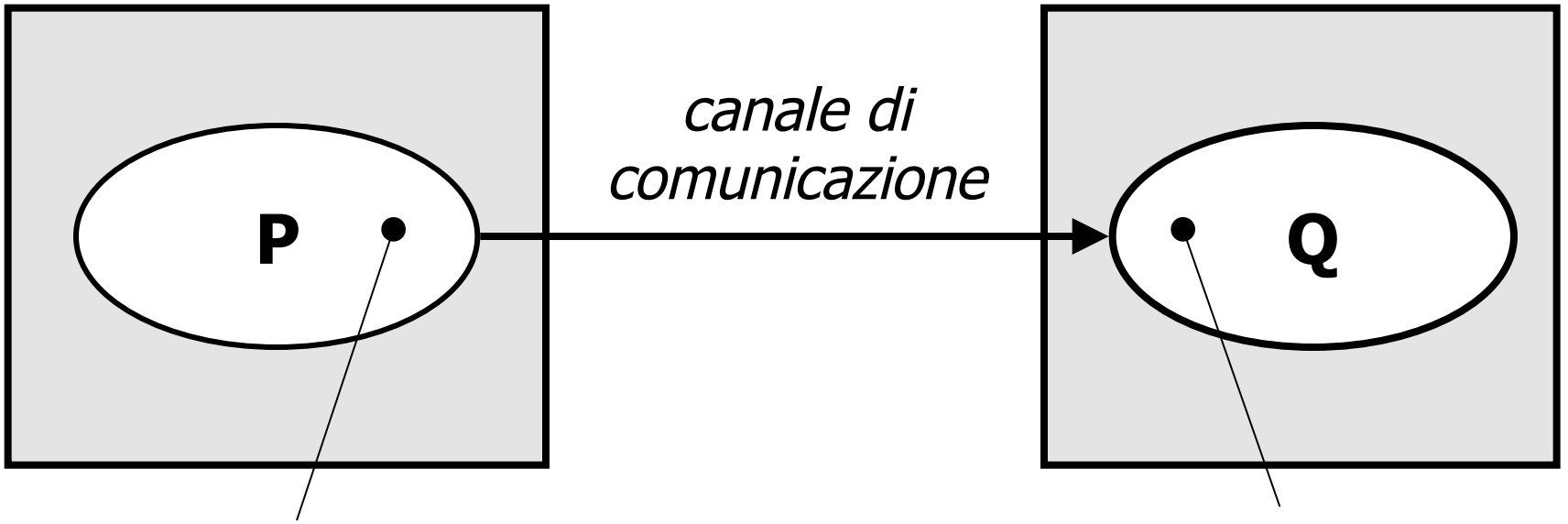
Modello a scambio di messaggi

- I processi interagiscono attraverso una rete di comunicazione
- I processi non condividono alcuna risorsa
  - **Comunicazione ma non competizione**
- Due processi comunicano attraverso un **canale di comunicazione**
  - Operazioni di **send** e **receive**

Canale di comunicazione

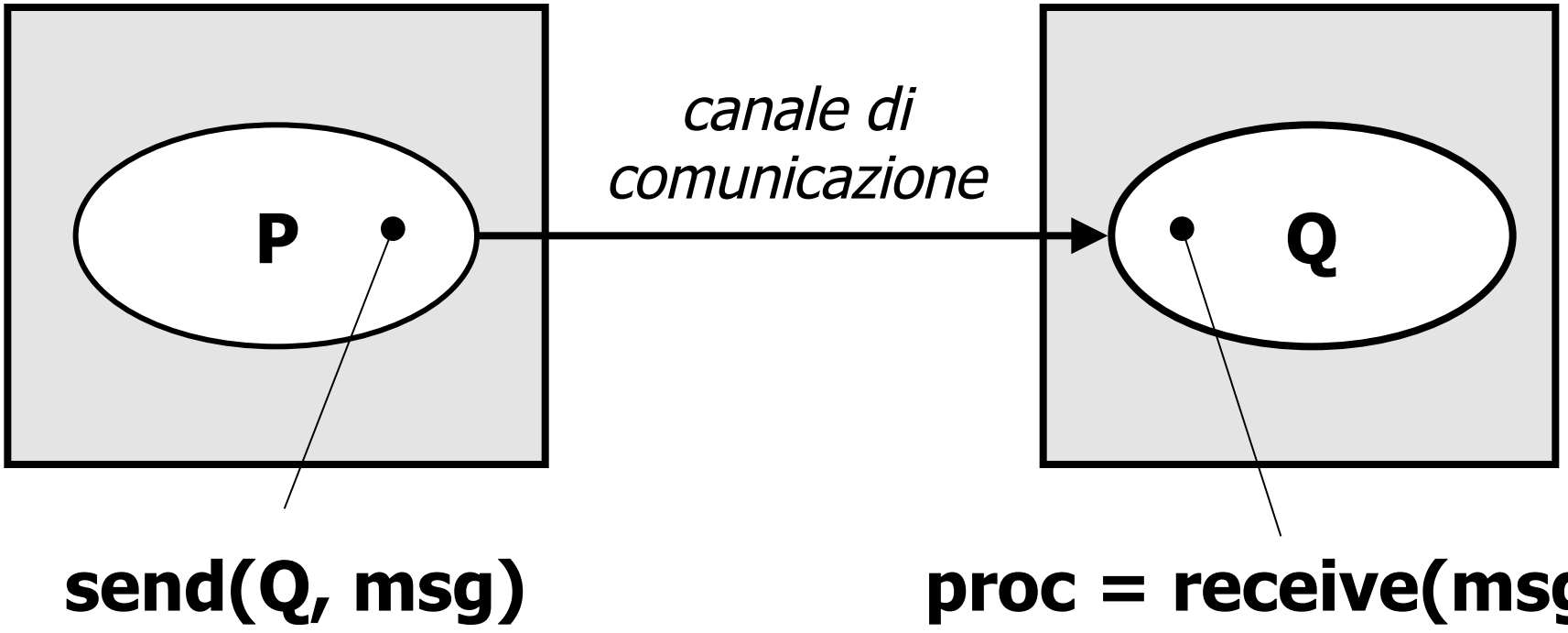
Denominazione simmetrica

Il mittente nomina esplicitamente il destinatario e viceversa



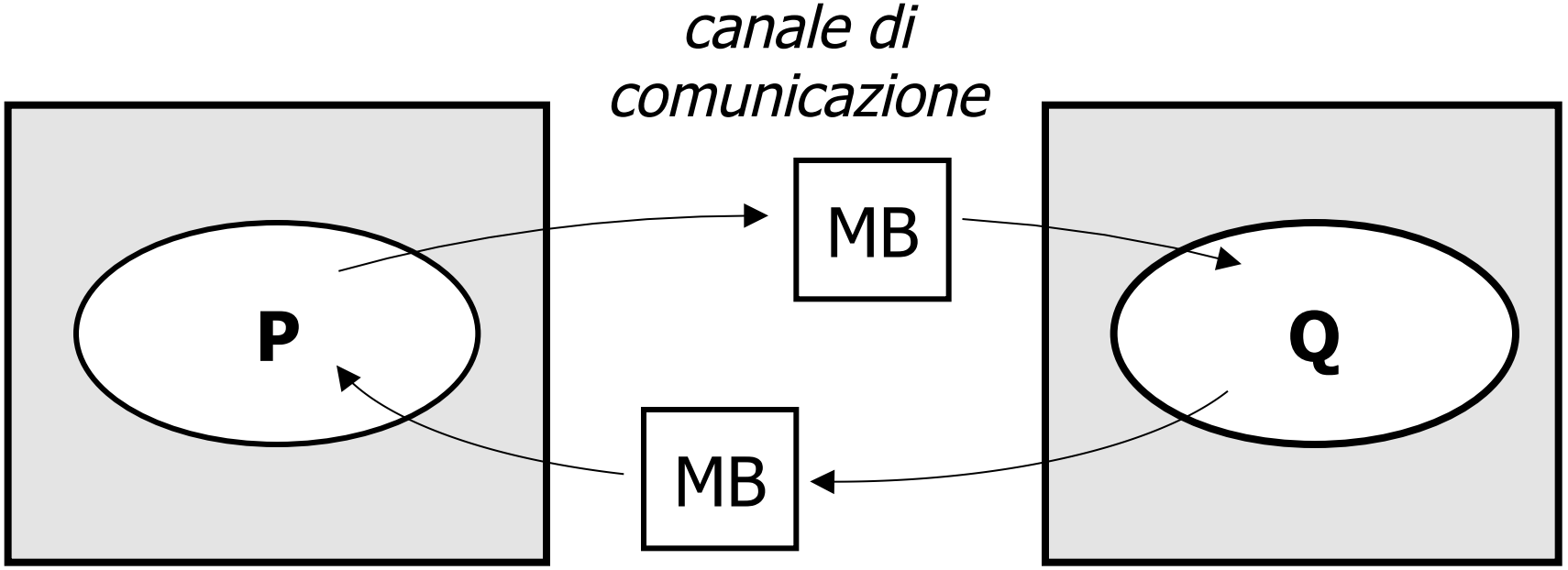
Denominazione asimmetrica

Il mittente nomina esplicitamente il destinatario ma questi non nomina i processi da cui vuole ricevere



Denominazione indiretta

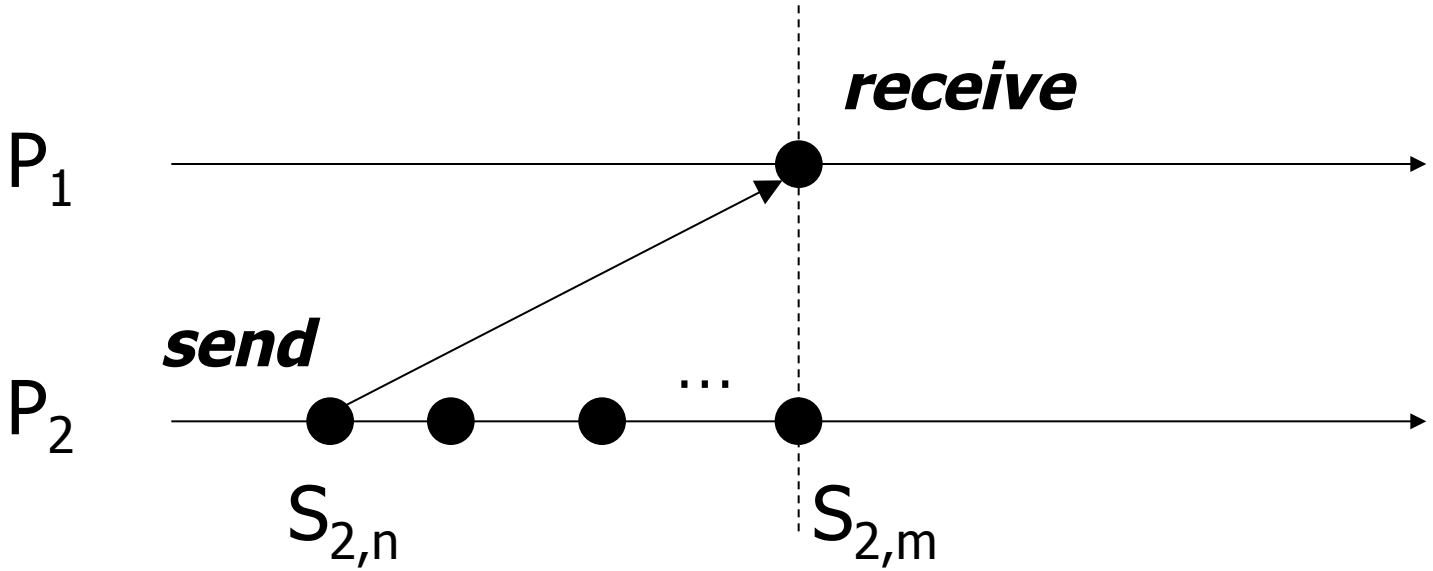
*I messaggi sono inviati a delle **mailbox** (**porte**) e da questi ricevuti*



Modalità di comunicazione

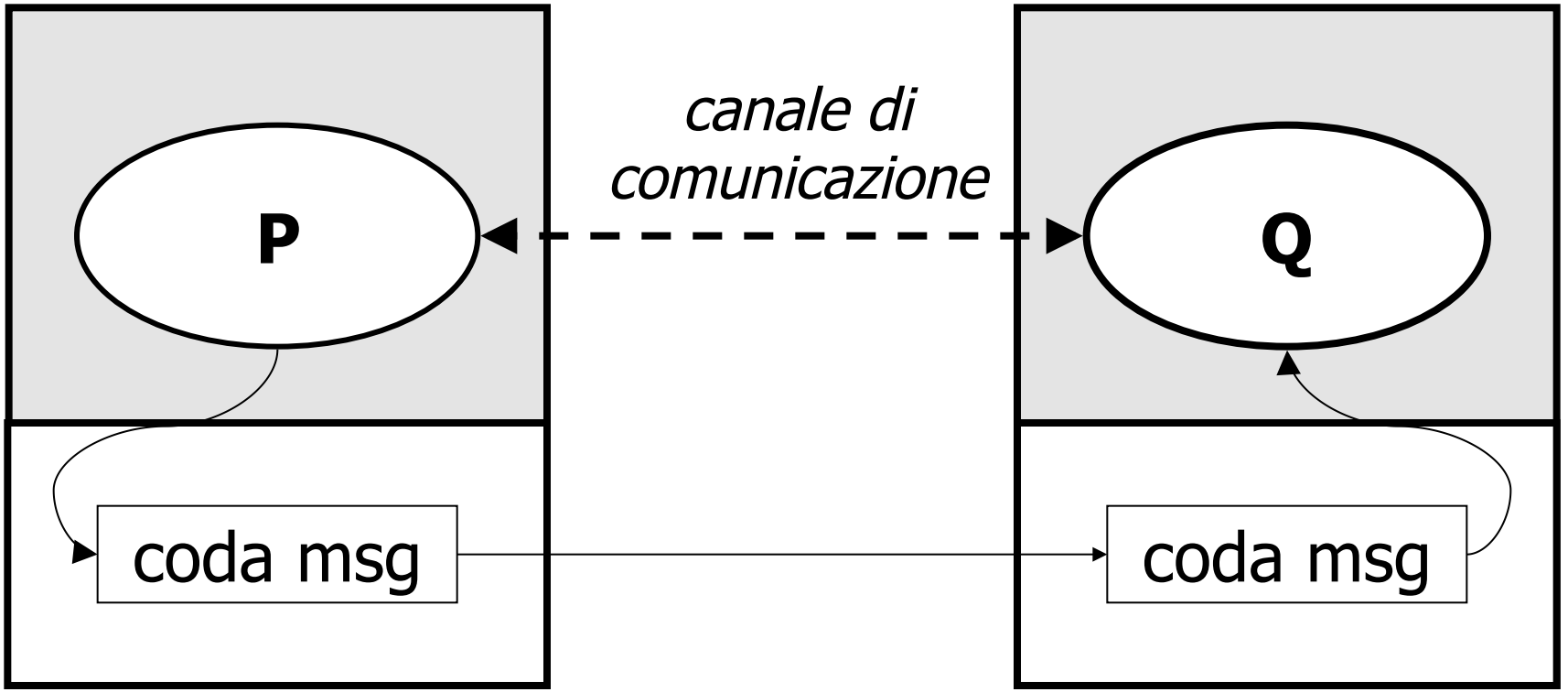
- Canale uno-a-uno
- Canale molti-a-uno
- Canale uno-a-molti

Send asincrona



- Efficienza e semplicità implementativa
- Carenza espressiva
- Problema dei buffer limitati

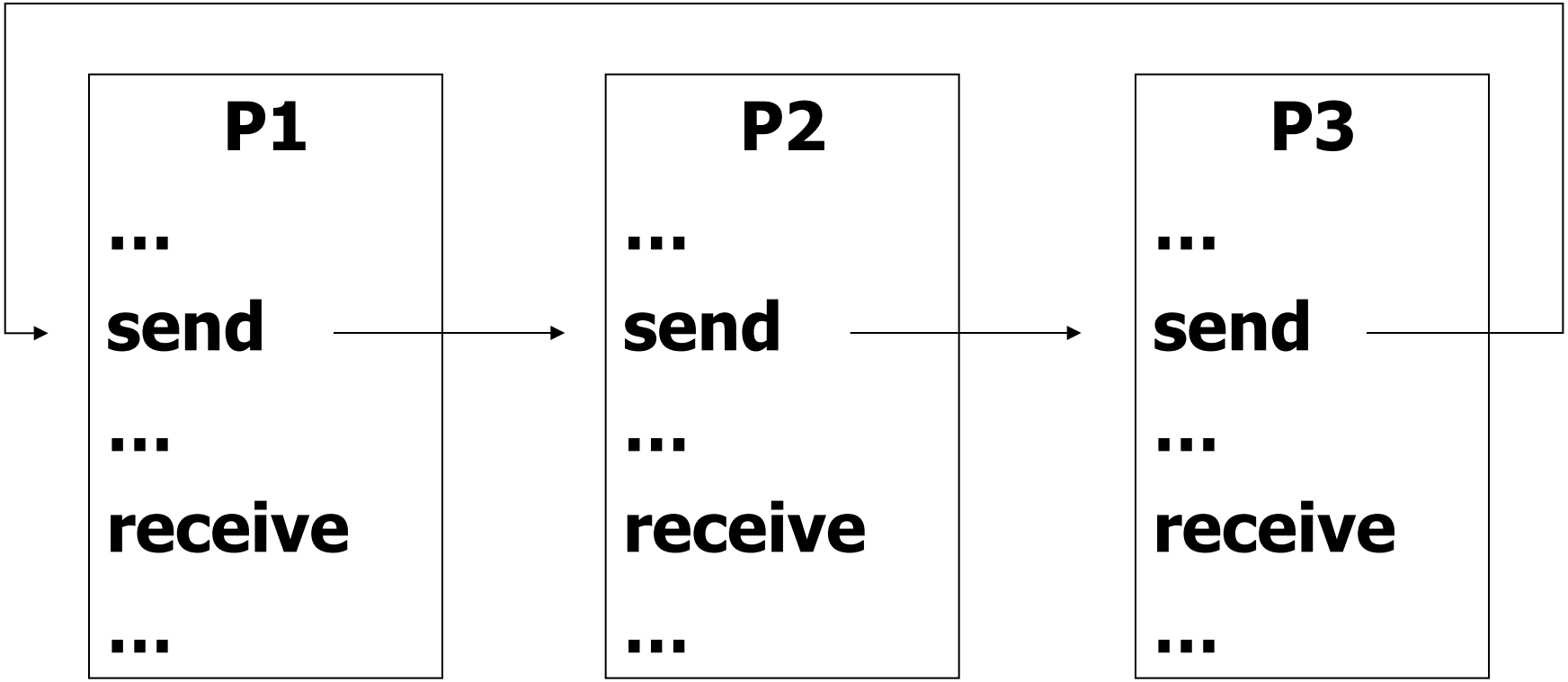
Send asincrona e buffer limitati



Gestione dei buffer limitati: soluzioni alternative:

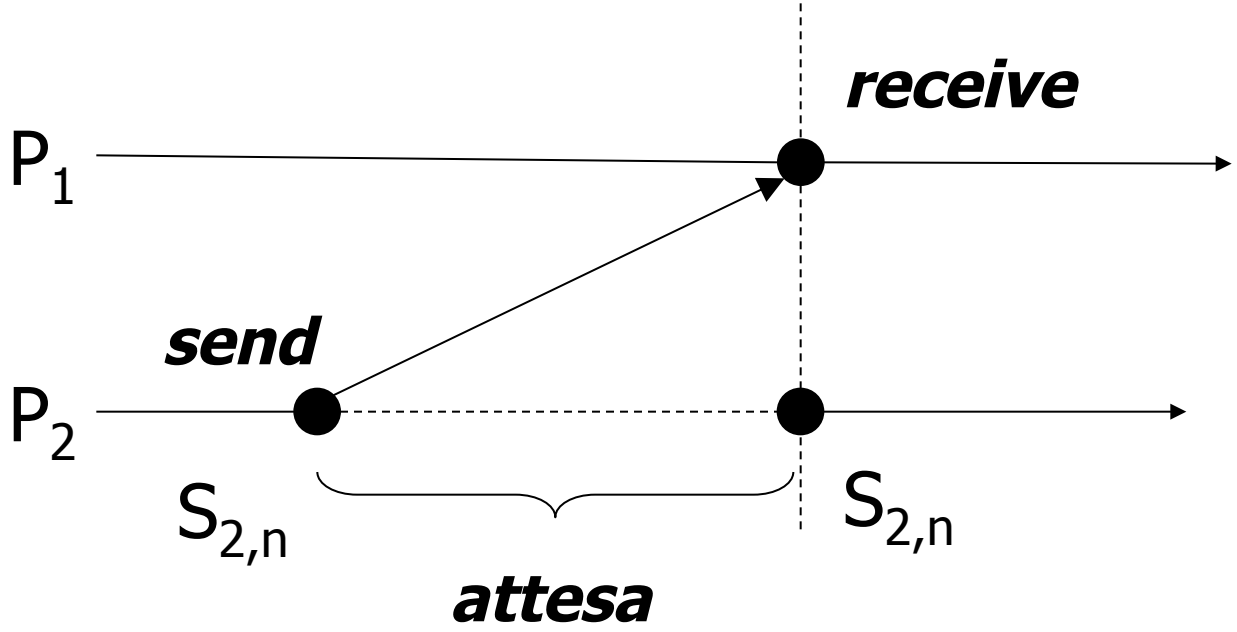
- La send può bloccarsi
- Il supporto runtime solleva una eccezione

Send asincrona: blocco critico nascosto



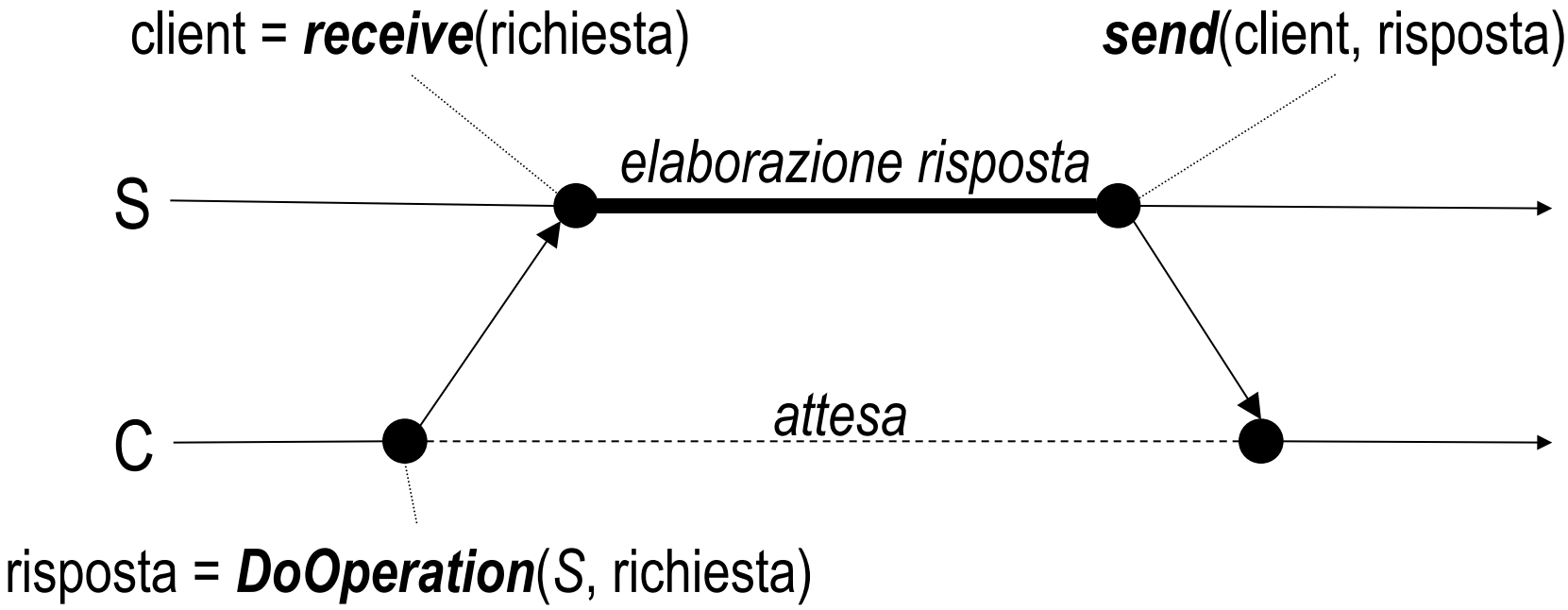
Il programma produce un blocco critico se tutti i buffer sono pieni

Send sincrona



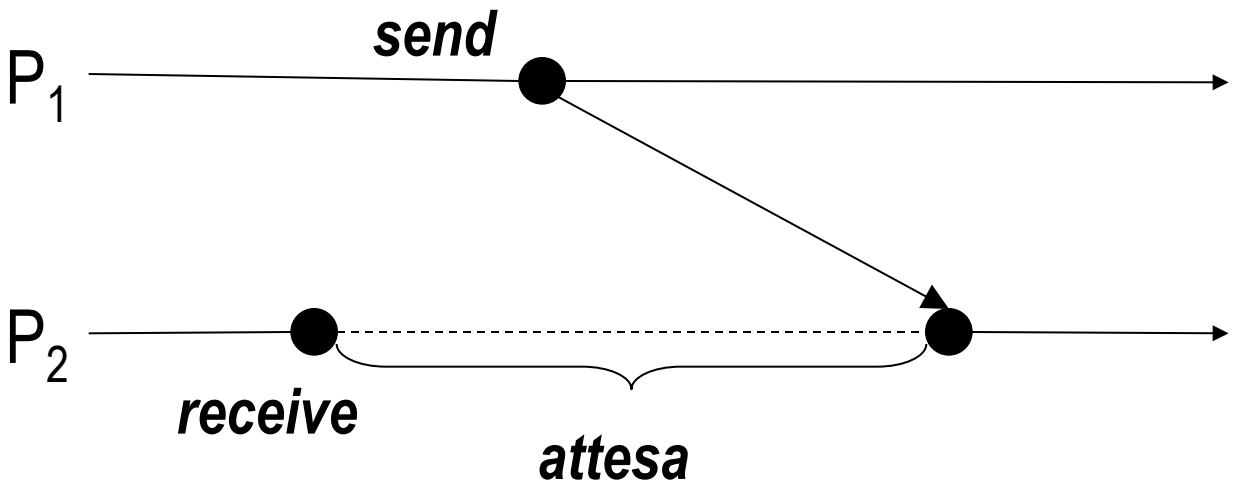
- Maggiore livello di astrazione
  - È un punto di sincronizzazione
- Buffer limitati (non ha bisogno di meccanismi nascosti)
- Può essere realizzata con le primitive asincrone
  - Ridotto parallelismo

Send tipo chiamata di procedura remota



- Livello di astrazione ancora più elevato
- Parallelismo ancora più ridotto

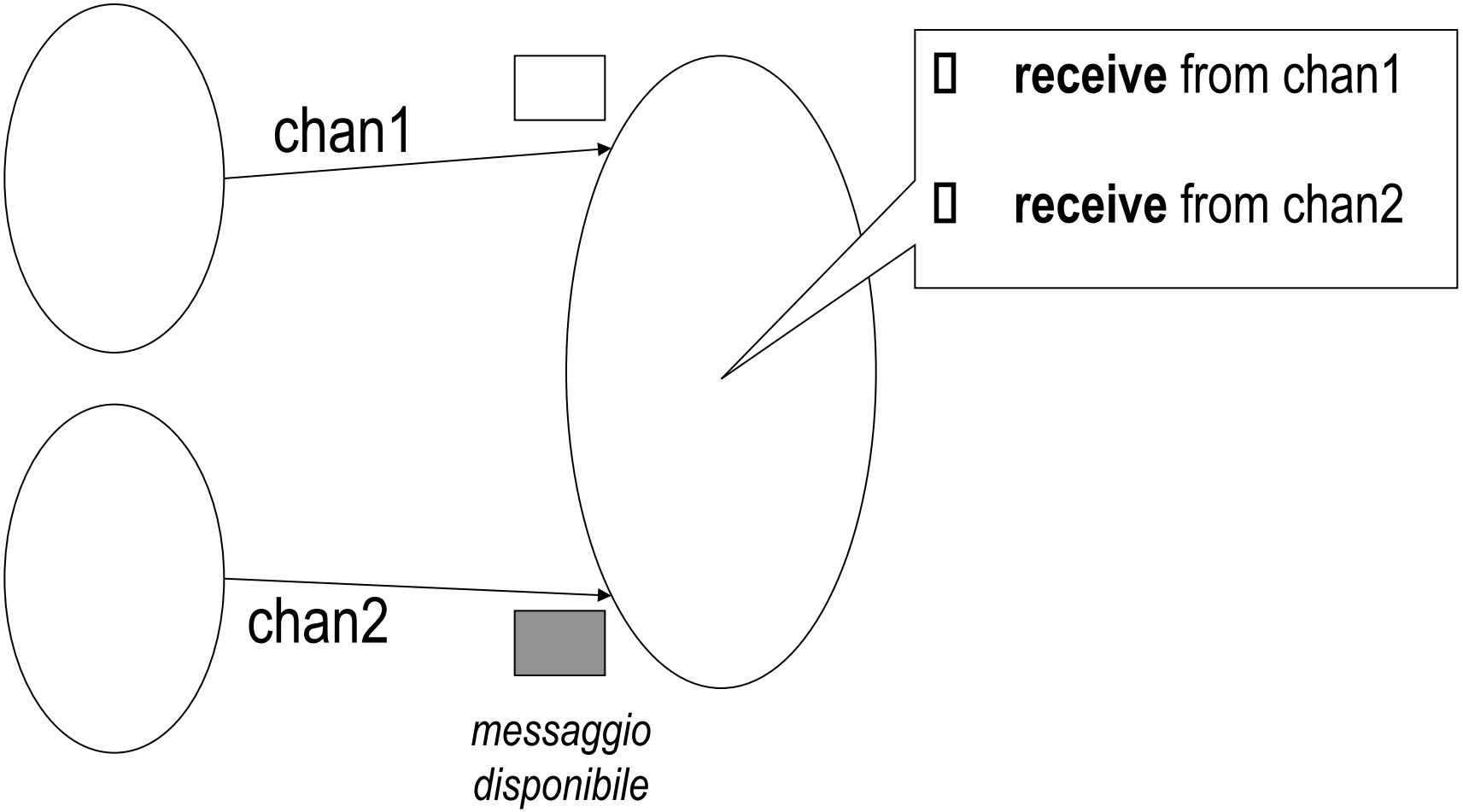
Receive bloccante



- Punto di sincronizzazione
- Sono necessari meccanismi aggiuntivi per ricevere da canali alternativi

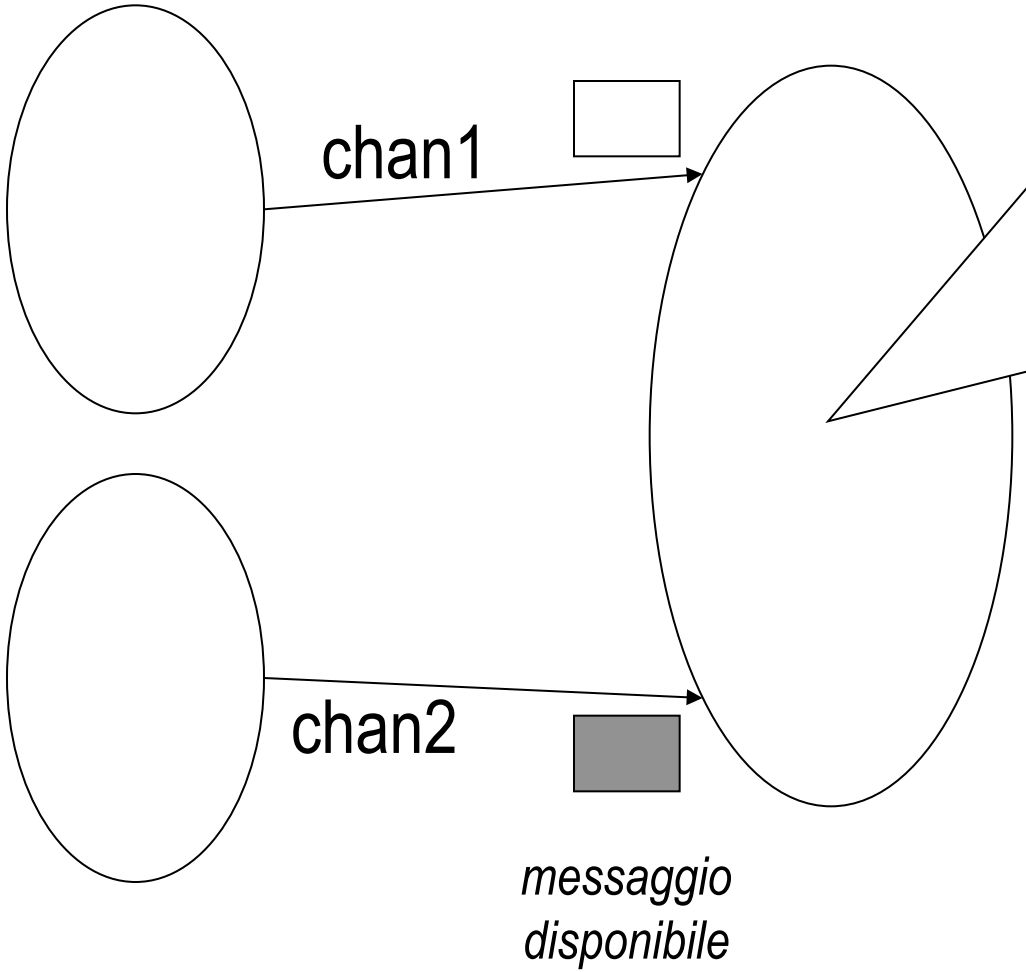
Receive bloccante

Si vuole ricevere da chan1 o da chan2



Receive non-bloccante

Si vuole ricevere da chan1 o da chan2




```
…
ricevuto = false;
while (!ricevuto) {
   msg = receive(chan1);
   if (msg != null)
      ricevuto = true;
   else {
      msg = receive(chan2);
      if (msg != null)
         ricevuto = true;
   }
}
…
```

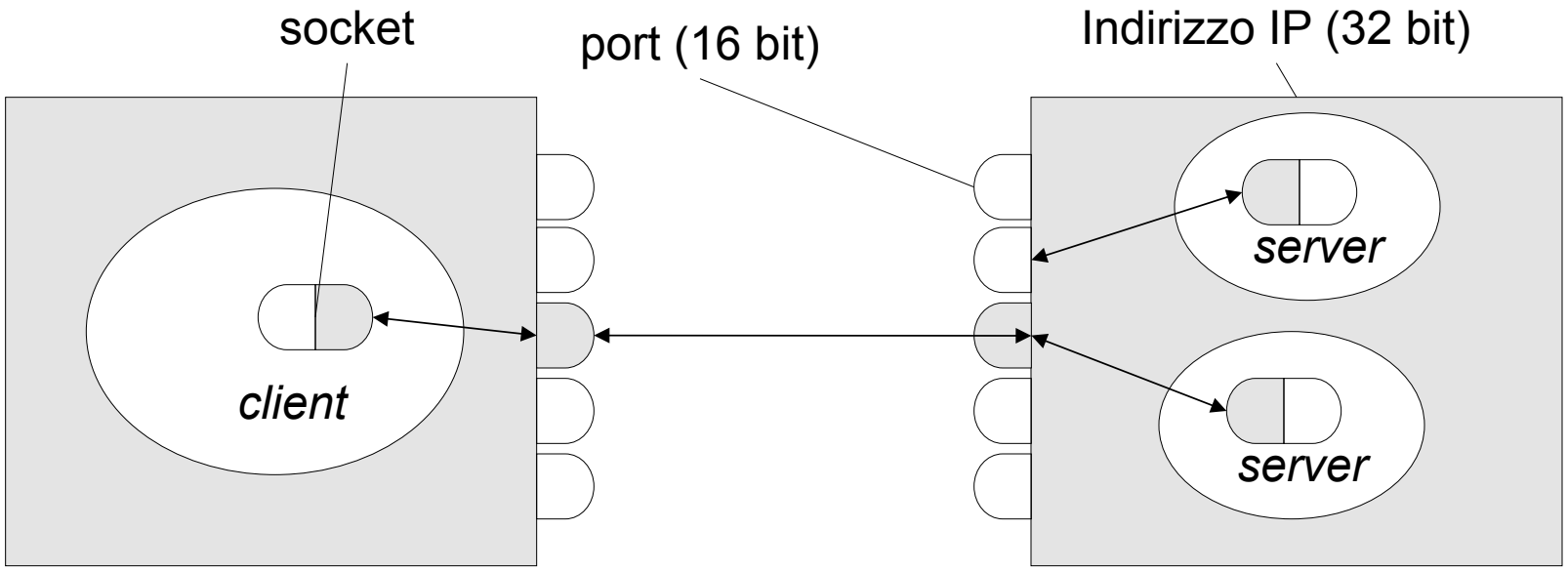
cicli di attesa attiva

Protocolli Internet

- Un processo è definito da un programma Java eseguibile
- Java fornisce API per l'utilizzo dei protocolli
  - **UDP** (comunicazione inaffidabile)
  - **TCP** (comunicazione affidabile)

  attraverso l'astrazione di **socket**
- La modalità di naming è **asimmetrica, indiretta**
- La modalità di comunicazione è **molti-a-uno**
- La **send è asincrona** e la **receive è bloccante**

Socket



- Il mittente specifica il destinatario per mezzo della coppia ($IP_{ricevente}$, $Porta_{ricevente}$)
- Un canale di comunicazione è bidirezionale ed è specificato da ($Ip_{mittente}$, $Porta_{mittente}$, $IP_{ricevente}$, $Porta_{ricevente}$)
- Un processo utilizza un socket per inviare e ricevere messaggi
- Un socket deve essere “legato” ad una porta

Indirizzi e nomi

La classe InetAddress

Java utilizza la classe **InetAddress** per rappresentare un indirizzo di rete

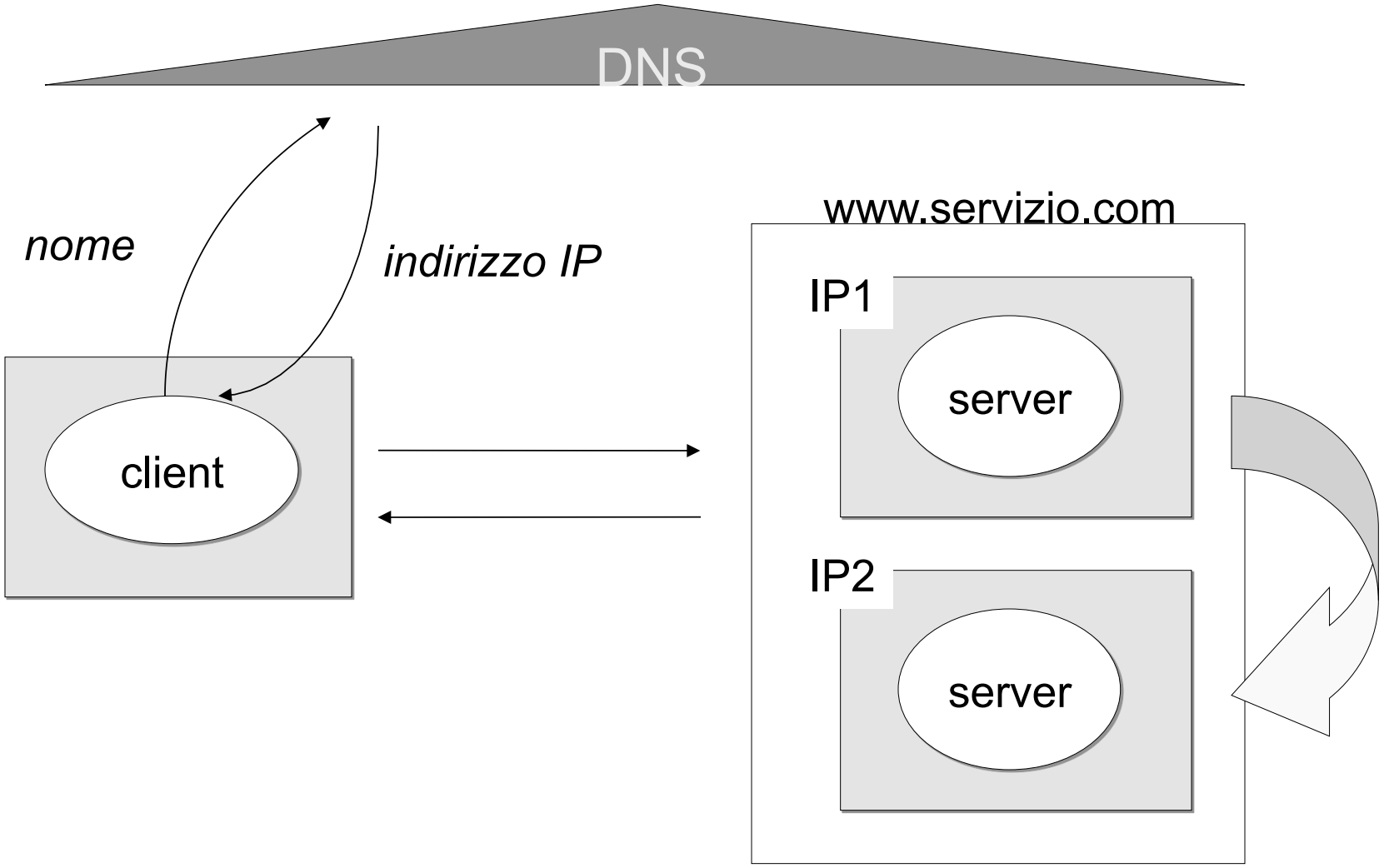
package java.net

public static InetAddress getByName()
throws UnknownHostException

ESEMPI

InetAddress unHost = InetAddress.getByName(“131.114.9.137”)
InetAddress unHost = InetAddress.getByName(“cirano.iet.unipi.it”)
InetAddress unHost = InetAddress.getByName(localhost)
InetAddress unHost = InetAddress.getByName(null)

Rilocazione di un servizio



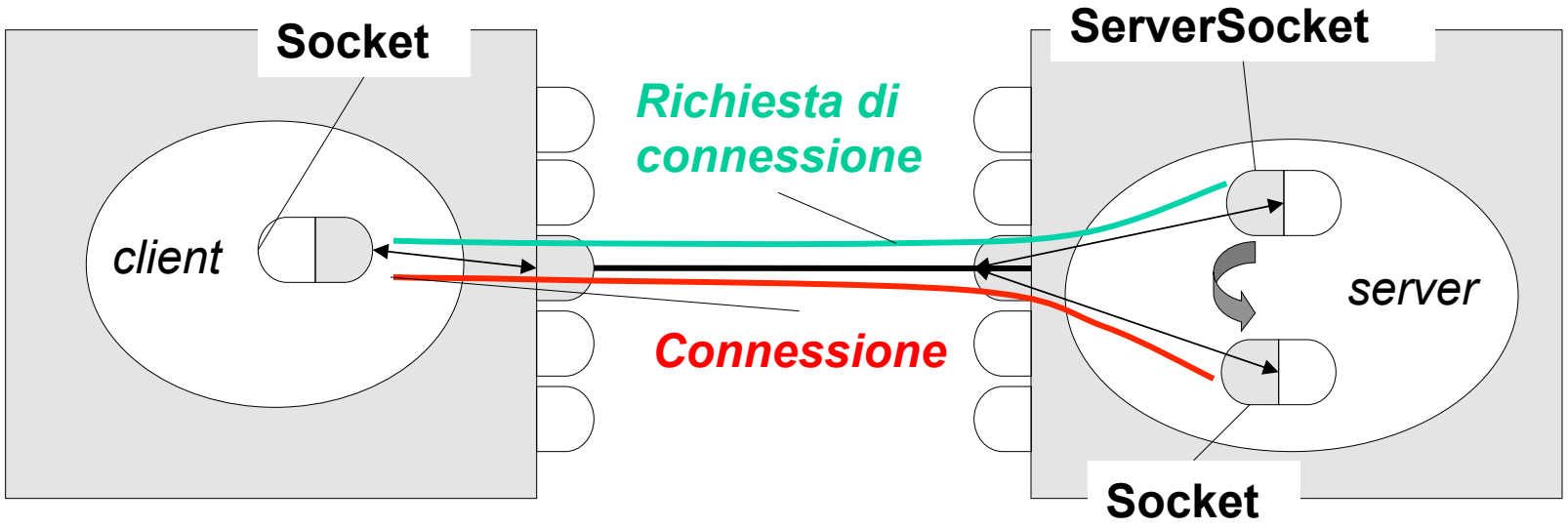
La programmazione di rete con Java

Le API per TCP

TCP

- TCP fornisce un servizio di *comunicazione punto-punto, affidabile*, orientato alla connessione che garantisce che
  - i messaggi sono consegnati al destinatario nonostante un numero “ragionevole” di pacchetti scartati o persi;
  - i messaggi consegnati al destinatario non sono nè alterati, né duplicati né fuori sequenza

Le classi Socket e ServerSocket



Il cliente si *connette* ad una porta del server tramite un **Socket**

Il server *accetta* connessioni su di un **ServerSocket**

Una connessione è terminata da due **Socket**

Quando si instaura la connessione, ad ogni **Socket** è associato un flusso di ingresso ed uno di uscita

Gli oggetti trasmessi sulla connessione sono serializzati

ServerSocket **e** Socket

- La classe **ServerSocket** è utilizzata dal server per creare un socket su cui accettare richieste di connessione
  - L'operazione **accept** ascolta richieste di connessione.
  - L'operazione ritorna un **Socket** se un cliente ha fatto una richiesta di connessione oppure blocca l'esecuzione del server in attesa di tale richiesta
- La classe **Socket** descrive i punti terminali (socket) di una connessione tra due macchine
  - Ogni socket è associato a due stream
  - I metodi **getInputStream** ed **getOutputStream** ritornano i flussi di tipo **InputStream** e **OutputStream**, rispettivamente, associati con il socket

Esempio

- SimpleServer
  - attende che si stabilisca una connessione con **SimpleClient**
  - ritorna a **SimpleClient** tutto ciò che riceve sulla connessione
  - chiude la connessione e termina non appena riceve la stringa “END”
- SimpleClient
  - si connette a **SimpleServer**
  - gli invia stringhe attraverso la connessione
  - riceve l’eco delle stringhe attraverso la connessione
  - invia la stringa “END” e termina

Multi-threaded server

```
1. import java.io.*;
2. import java.net.*;
3. public class Server {
4.  static final int PORT = 8080;
5.  public static void main(String[] args) throws IOException {
6.   ServerSocket s = new ServerSocket(PORT);
7.   try {
8.     while ( true ) {
9.       Socket socket = s.accept();
10.                 try {
11.                 new ServerThread(socket); // conn <-> thread
12.                 } catch(IOException e) {
13.                 socket.close();
14.           }
15.           }
16.       } finally {
17.         s.close();
18.       }
19. }
20.}
```

Multi-threaded server

```
1. class ServerThread extends Thread {
2.
3.   private Socket socket;
4.   private BufferedReader in;
5.   private PrintWriter out;
6.
7.
8.   public ServerThread(Socket s) throws IOException {
9.     socket = s;
10.    in = new BufferedReader(new InputStreamReader(
11.          socket.getInputStream()));
12.    out =new PrintWriter(new BufferedWriter(
13.          new OutputStreamWriter(socket.getOutputStream())),
14.                                 true);
15.    start();
16.  } // continua
```

Multi-threaded server

```
1.   public void run() {
2.     try {
3.      while (true) {
4.        String str=in.readLine();
5.           if(str.equals("END")) break;
6.        System.out.println("Echo:" + str);
7.        out.println(str);
8.     }
9.     }catch(IOException e){
10.    }finally{
11.     try{
12.       socket.close();
13.     }catch(IOException e){}
14.    }
15.  } // closes run
16. } // closes class
```

Metodi bloccanti

- Il costruttore **Socket()** è bloccante in attesa che sia stabilita la connessione al server
- Il metodo **connect()** è bloccante in attesa che sia stabilita una connessione
- Il metodo **read** sull'**InputStream** associato ad un socket è bloccante in attesa che ci siano dati disponibili

Impostazione dei timeout

- **Timeout**
  - Una variabile **int** maggiore o uguale a zero
  - Un valore maggiore di zero specifica un intervallo di attesa in millisecondi
  - Il valore zero specifica un intervallo di attesa indefinito

Timeout sulla connessione

- **Timeout sulla connessione**

  connessione senza timeout

  Socket s = new Socket(host, port); // bloccante

  connessione con timeout

  Socket s = new Socket(); // socket non connesso

  s.connect(host, port, timeout);

- Se il timeout scatta prima della connessione viene lanciato un **SocketTimeoutException**

Timeout sulla lettura

```
Socket s = new Socket(host, port);
int timeout = 10;
…
s.setSoTimeout(timeout);
…
 BufferedReader in = new BufferedReader(
                                 new InputStreamReader(
                                       s.getInputStream()));
try {
      String line;
      while ( (line = in.readLine()) != null ) {
            <elaborazione linea>;
      }
} catch(SocketTimeoutException e) {
      <gestione dell'eccezione>;
} catch (IOException e) {
      <gestione dell'eccezione>;
}
…
```

Half-close

- Quando un cliente invia un messaggio al servitore, questo deve capire quando il messaggio è finito. Per questo motivo,
  - molti protocolli Internet sono orientati alla linea (ad esempio SMTP)
  - altri protocolli utilizzano messaggi con il formato (***header***, ***payload***) dove ***header*** specifica anche la dimensione dei dati
- Chiudere il socket come si farebbe con un file non è consigliabile perchè in tal modo si abbatte la connessione
- Si può utilizzare la tecnica **half-close**
  - si chiude il flusso di uscita (**OutputStream**) associato al socket per indicare al server la fine dei dati, ma
  - si lascia aperto il flusso **InputStream** per leggere la risposta del server
- Questa tecnica è adeguata per servizi "one shot" o "botta e risposta" (ad esempio HTTP)

La programmazione di rete con Java

Le API per UDP

UDP: concetti generali

- UDP fornisce un servizio di comunicazione *inaffidabile* senza connessione, cioè un datagram
  - può non arrivare (omission failure)
  - arrivare duplicato oppure fuori sequenza
- Un datagram ha una dimensione massima di 64Kb e tipica di 8Kb
- Le primitive di comunicazione sono
  - **send** asincrona
  - **receive** bloccante
- Il meccanismo di denominazione è **uno-a-molti**, **asimmetrico indiretto (receive from any)**
- UDP si utilizza in servizi che possono tollerare degli omission failures occasionali e non vogliono sopportare l'overhead imposto da TCP

UDP: concetti generali

- Un processo che vuole inviare o ricevere un datagram deve prima creare un socket e "*legarlo*" (bind) ad una porta
- Il server "*lega*" (bind) il socket ad una porta specifica, il cliente lega il socket ad una qualunque porta libera
- La primitiva **receive** ritorna l'indirizzo IP e la porta del mittente
- I datagrammi indirizzati a porte non "*legate*" (bound) ad alcun processo sono scartati

La classe DatagramPacket

- La classe **DatagramPacket** descrive un datagramma UDP
  - **DatagramPacket(byte[] buf, int length)** costruisce un **DatagramPacket** per ricevere pacchetti di lunghezza **length**
  - **DatagramPacket(byte[] buf, int length, InetAddress address, int port)** costruisce un **DatagramPacket** per inviare pacchetti di lunghezza **length** alla porta **port** dell'host **address**

struttura di un datagram

Metodi della classe DatagramPacket

InetAddress getAddress() ritorna l'indirizzo IP dell'elaboratore a cui questo messaggio è stato inviato o da cui è stato ricevuto

byte[] getData() ritorna i dati contenuti nel buffer di questo messaggio

int getLength() ritorna la lunghezza (in byte) dei dati ricevuti o da spedire contenuti in questo messaggio

int getPort() ritorna il numero di porta sul nodo remoto al quale questo messaggio sarà inviato o dal quale è stato ricevuto

La classe DatagramSocket

- La classe **DatagramSocket** descrive i socket per inviare e ricevere pacchetti UDP
  - **DatagramSocket()** costruisce un **DatagramSocket** e lo lega ad una porta disponibile sull'elaboratore locale
  - **DatagramSocket(int port)** costruisce un DatagramSocket e lo lega alla porta specificata sull'elaboratore locale
  - **DatagramSocket(int port, InetAddress laddr)** costruisce un **DatagramSocket** e lo lega alla porta specificata ed all'indirizzo locale specificato

I metodi di DatagramSocket

- **public void send(DatagramPacket p) throws IOException** invia un **DatagramPacket** da questo socket. Il messaggio include i dati da trasmettere, la loro lunghezza, **l'indirizzo IP del mittente ed il numero di porta su questo elaboratore**

- **public void receive (DatagramPacket p) throws IOException** riceve un DatagramPacket da questo socket.

  Quando il metodo ritorna, il buffer specificato dal **DatagramPacket** è riempito con i dati ricevuti. Il **DatagramPacket** contiene anche l'indirizzo IP dell'elaboratore mittente ed il numero di porta su tale elaboratore.

  Il metodo è bloccante.

  Il campo lunghezza del **DatagramPacket** specifica la lunghezza del messaggio ricevuto. Se il messaggio è più lungo di quanto specificato, il messaggio viene troncato.

Esempio

- DgramServer
  - attende di ricevere un DatagramPacket
  - invia l'eco al mittente
  - ritorna in attesa
- DgramClient per 5 volte esegue le seguenti azioni:
  - invia una stringa a **DgramServer** in un **DatagramPacket**
  - attende l'eco dal server

Produttori e Consumatori

- Numero finito di produttori (P)
- Numero finito di consumatori (C)
- Ogni produttore può inviare un messaggio ad uno qualunque dei consumatori.
- **Problema**.
  Si vogliono evitare situazioni in cui esistono consumatori in attesa di messaggi ed altri con messaggi in coda
- **Soluzione**: **utilizzo di un meccanismo di bufferizzazione esplicito**
  - ***un buffer memorizza i messaggi inviati dai produttori e li convoglia in ordine FIFO ai consumatori***

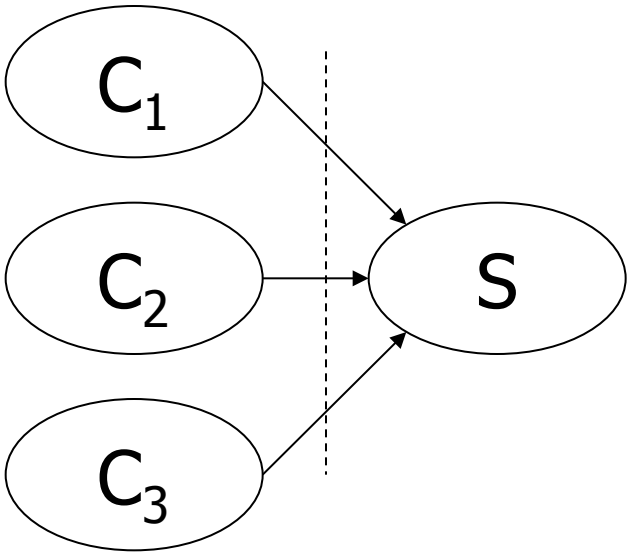
Produttori e Consumatori

- Ogni produttore trasmette un dato per mezzo di un ***messaggio dati***
- Ogni consumatore invia un ***messaggio di controllo*** (***pronto***) per notificare la sua disponibilità a ricevere un messaggio e quindi si mette in attesa di ricevere tale messaggio
- Si assume che il buffer dei messaggi e dei consumatori in attesa abbiano lunghezza illimitata
- Vai alla soluzione

Molti-a-uno / Uno-a-molti

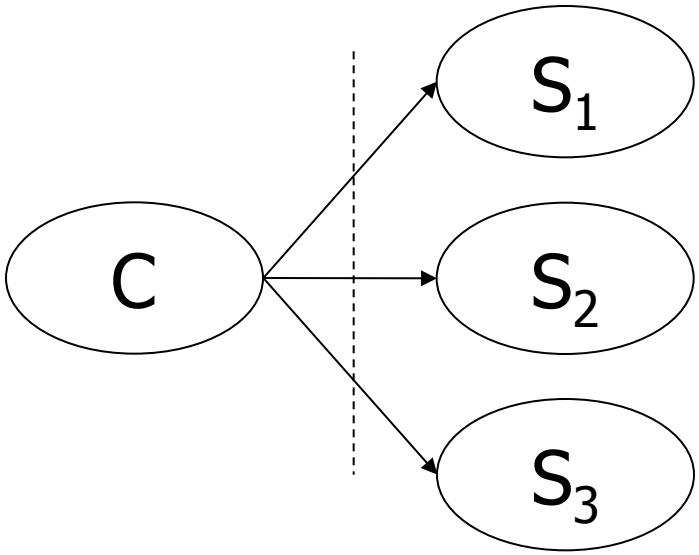
schema molti-a-uno

unicast



schema uno-a-molti

multicast



Multicast

- **Applications**
  - Bulk data transfer (software upgrade)
  - Streaming continuous media (audio, video)
  - Shared data applications (whitheboard)
  - Data feed (stock quotes)
- **Possible implementations**
  - One-to-all unicast (non richiede un supporto esplicito dal livello rete)
  - Application level multicast (l'applicazione deve mantenere unoverlay di distribuzione)
  - Explicit multicast (richiede supporto dal livello rete)

Internet multicast

- **Address indirection**: un singolo identificatore viene utilizzato per un gruppo di ricevitori
  - Indirizzo IP in **classe D** nel range [224.0.0.0, 239.255.255.255]
  - L'indirizzo 224.0.0.0 è riservato
- **Internet Group Management Protocol** (**IGMP**) (RFC 2236)
  - Permette ad un host di informare il proprio router che un'applicazione vuole unirsi (***join***) ad uno specifico multicast group.
  - Mantiene la ***local membership***
- **Network-layer multicast routing algorithms**
  - Permettono di coordinare i router in modo che i i multicast datagram siano consegnati alle destinazioni finali
  - PIM, DVMRP, MOSPF

Multicast Socket

- **Time To Live** (**TTL**) permette di controllare quanto distante un multicast datagram si può spingere
- TTL è usato anche come soglia
  - un multicast datagram può attraversare un router se il suo TTL è maggiore di quello impostato nel router (comunque è decrementato di uno)

TTL	Scope
0	Same host
1	Same subnet
<32	Same site, organization, dept
<64	Same region
<128	Same continent
<255	Unrestricted, global

Multicast Socket

```
import java.io.*;
import java.net.*;
import java.util.*;

public class MulticastServer {
   public static void main(String[] args) {
      byte[] buf = new byte[256];
      String dString = new Date().toString();
      buf = dString.getBytes();
      MulticastSocket socket = null;
      try {
         socket = new MulticastSocket();
         socket.setTimeToLive(0);

         InetAddress group = InetAddress.getByName("230.0.0.1");
         DatagramPacket packet;
         packet = new DatagramPacket(buf, buf.length, group, 4446);
         socket.send(packet);
      } catch (IOException e) {
         e.printStackTrace();
      } finally {
         socket.close();
      }
   }
}
```

public class MulticastSock extends DatagramSocket

Il **MulticastSocket** non è bound ad alcuna porta perché non deve ricevere risposta

setTimeToLive permette di controllare la diffusione del multicast

230.0.0.1: uno dei possibili gruppi multicast

Multicast Socket

```
import java.net.*;
import java.io.*;

public class MulticastClient {
   public static void main(String[] args) throws
                                              IOException {
      MulticastSocket socket  = null;
      byte[] buf = new byte[256];
      socket  = new MulticastSocket(4446);
      InetAddress group =
                           InetAddress.getByName("230.0.0.1");
        socket.joinGroup(group);

        DatagramPacket packet;
        for (int i = 0; i < 5; i++) {
           packet = new DatagramPacket(buf, buf.length);
           socket.receive(packet);
           String received = new String(packet.getData());
           System.out.println("Received  " + received);
        }
        socket.leaveGroup(group);
        socket.close();
   }
}
```

Algoritmo

- Join a multicast group
- Receive multicast packets
- Leave a multicast group

UNIVERSITÀ DI PISA

Livelli di middleware