Mensajes

Programación Concurrente
UNQ - 2016

Recap

- Un programa secuencial es una lista de instrucciones

- La semántica de un programa secuencial es una traza de ejecución

- Dado un estado inicial, el programa secuencial produce un único estado final

    - El programa secuencial es **determinístico**

Recap

- Los programas concurrentes pueden hacer uso eficiente de los recursos computacionales

- Un programa concurrente es un conjunto de programas secuenciales

- La semántica de la ejecución de un programa concurrente es un conjunto de trazas de ejecución

- Dado un estado inicial y un programa concurrente, tenemos un conjunto posible de estados finales

    - El programa concurrente es **no-determinístico**.

Recap

- Algunas de las trazas de ejecución de un programa concurrente pueden no ser deseadas
- Estas trazas no deseadas violentan propiedades deseables de nuestra ejecución
  - Liveness: Algo bueno eventualmente ocurre
  - Safety: Algo malo nunca ocurre
- Sincronización: evitar violar safety o liveness

Recap

- **Exclusión mutua**: impedir un scheduling donde dos o más procesos/threads ejecuten una misma porción del programa.
- Asegurar **exclusión mutua** suponiendo:
  - Únicamente Lectura/Escritura como acciones atómicas: Dekker, Peterson y Bakery
  - Con soporte de acciones atómicas: TestAndSet(), Exchange(), FetchAndAdd()

Recap

- Mecanismos avanzados de sincronización:
  - Bajo nivel: **Semáforos**
    - acquire()/release()
  - Alto nivel: **Monitores**
    - wait()/notify()
    - Conceptuales (bibliografía) vs. Reales (Java)

Memoria Compartida

- Comunicación entre procesos/threads:
    - Usando memoria (variables) compartidas (i.e. **global** en hydra)

- Qué pasa si elimino la memoria compartida?

Ejemplo

```
process P1:
  // computo costoso
  // en funcion de x
```

```
process P2:
  // computo costoso
  // en funcion de y
```

```
process P3:
  // computo en funcion de los
  // resultados de P1 y P2
```

Ejemplo

```
process P1:
  // computo costoso
  // en funcion de x
```

```
process P2:
  // computo costoso
  // en funcion de y
```

```
process P3:
  // computo en funcion de los
  // resultados de P1 y P2
```

- Sin memoria compartida (ni otro mecanismo de comunicación), P3 no puede realizar su tarea

Mensajes

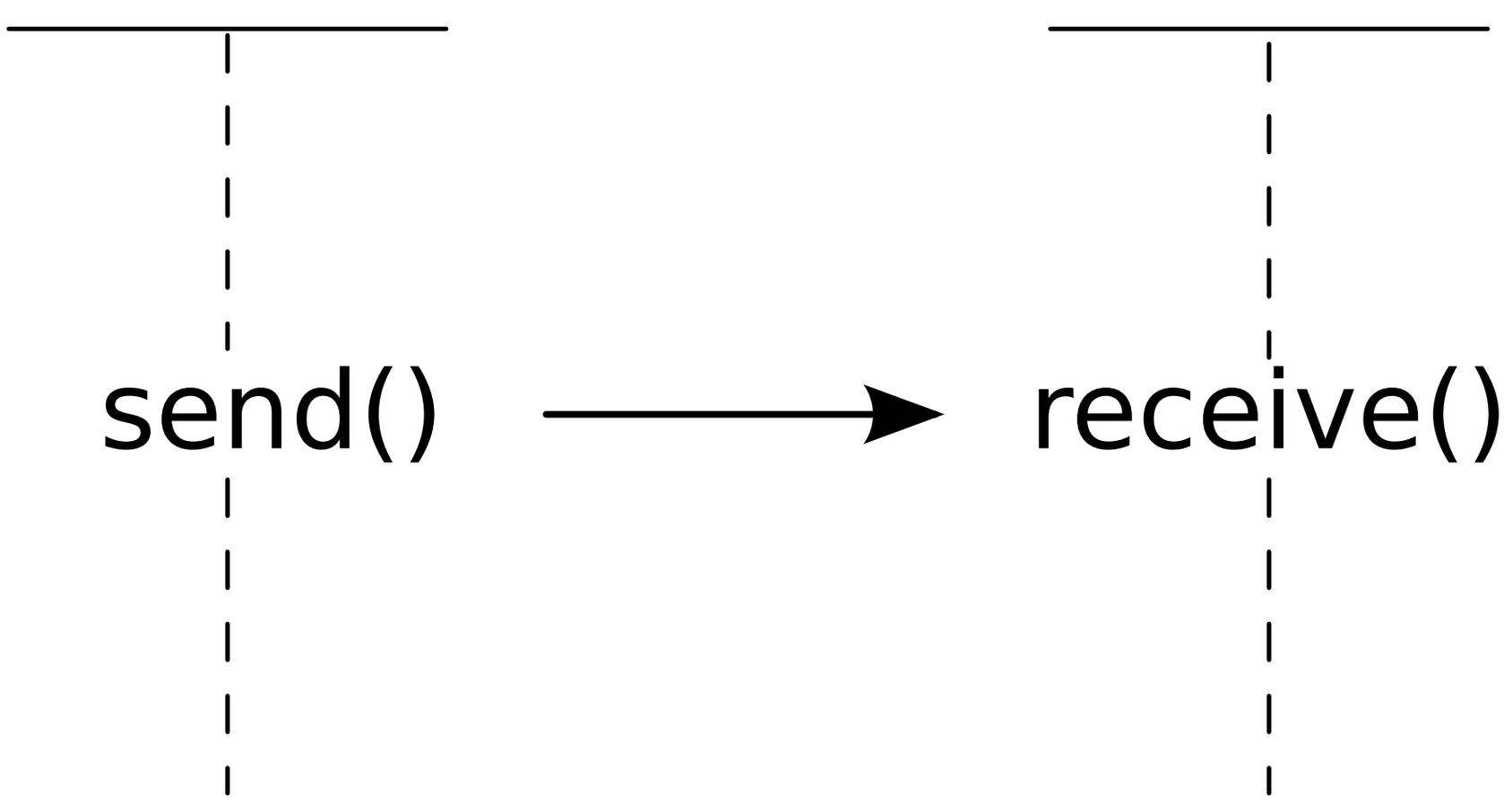
- **Mensajes**: mecanismo de sincronización que permite a un proceso enviar y recibir datos a través de un canal de comunicación
- En general:
  - Threads: usan memoria compartida (i.e. procesos livianos)
  - Procesos: usan mensajes (no comparten memoria)

Mensajes

- Ejemplo: Dado un canal llamado "channel"
  - channel.send(object):
    - envía el objeto a través del canal
  - object = channel.receive():
    - recibe un objeto a través del canal

Send/Receive

- Idealmente, receive() se debiera ejecuta al mismo tiempo que el send()

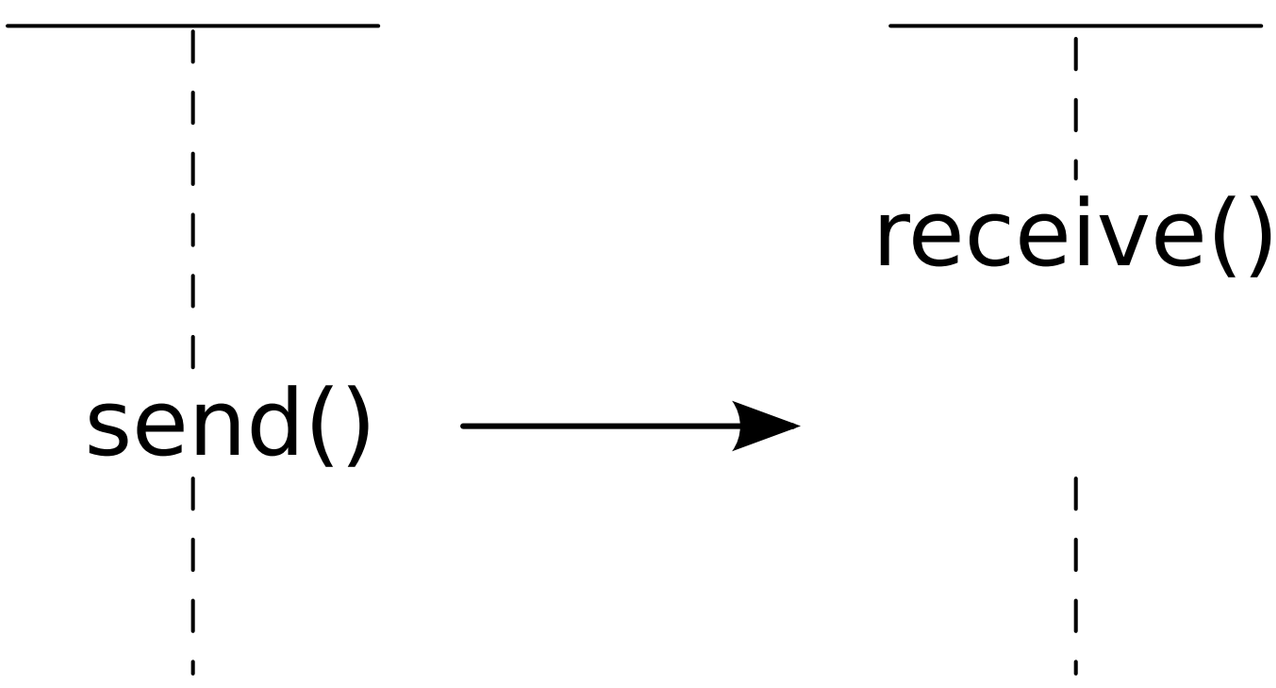


Send/Receive

- ¿Qué pasa si receive() se ejecuta **antes** del send()?

Send/Receive

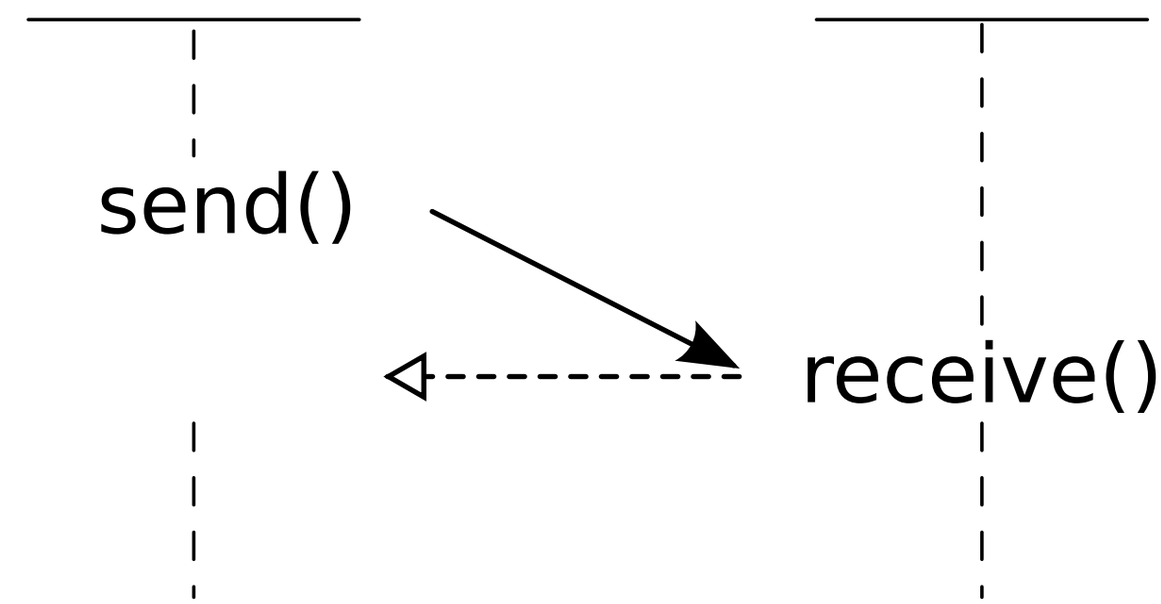
- ¿Qué pasa si receive() se ejecuta **antes** del send()?



- El proceso receptor (el que ejecuta receive) se bloquea hasta que se produzca un send()

Send/Receive

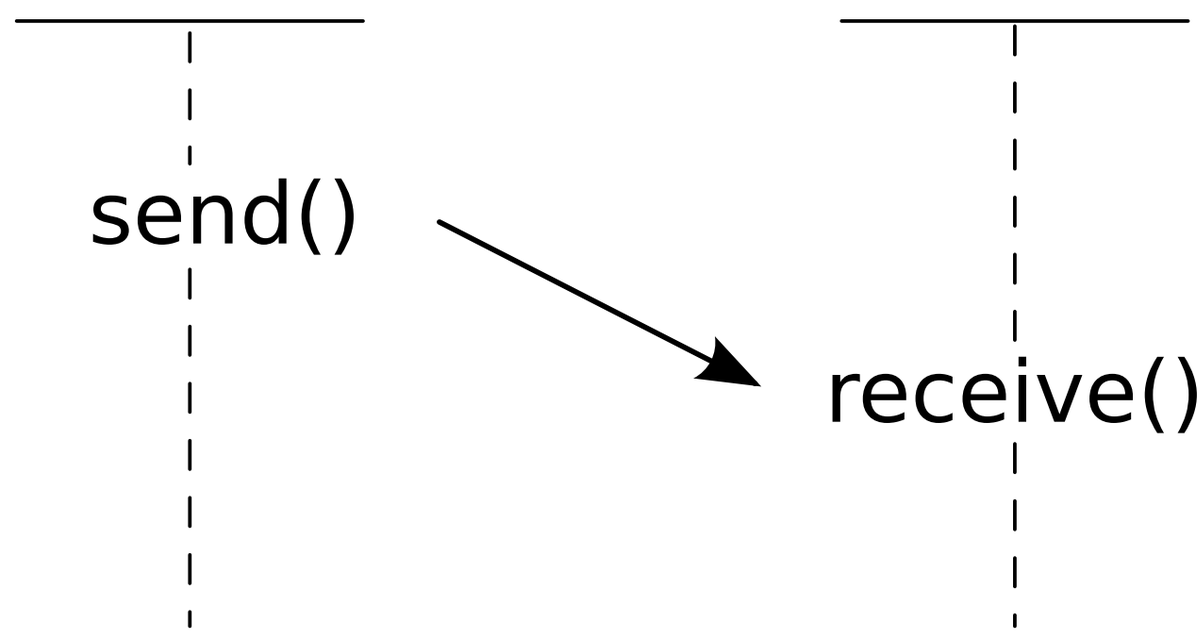
- ¿Qué pasa si send() se ejecuta **antes** del receive()?



Canal Sincrónico: El proceso emisor se bloquea hasta que el proceso receptor efectua el receive()

Send/Receive

- ¿Qué pasa si send() se ejecuta **antes** del receive()?



Canal Asincrónico: El proceso emisor continúa su ejecución. El mensaje se almacena en el canal (cola FIFO) hasta que el proceso receptor efectué un receive().

Send/Receive

- Receive(): El receptor se bloquea siempre hasta que exista un mensaje.
- Send(): El emisor se bloquea únicamente si el canal es sincrónico.
    - Vamos a suponer que los canales tienen capacidad para almacenar infinitos mensajes.

Identificación de Procesos

- ¿Qué formas hay de identificar los procesos que envían y reciben mensajes?

Identificación de Procesos

- ¿Qué formas hay de identificar los procesos que envían y reciben mensajes?

- **Simétrica Directa:** “El proceso A sabe que desea enviar al proceso B, y el proceso B sabe que desea recibir mensajes del proceso A”

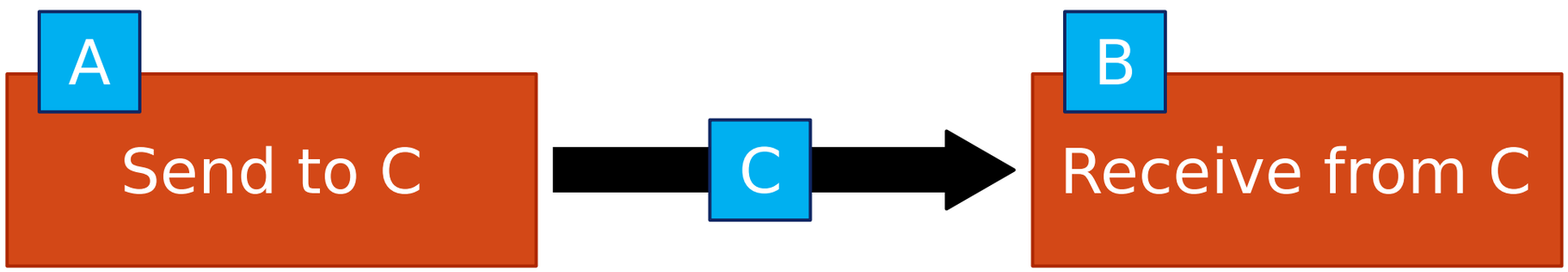
Identificación de Procesos

- ¿Qué formas hay de identificar los procesos que envían y reciben mensajes?

- **Asimétrica Directa:** “El proceso A sabe que desea enviar al proceso B, y el proceso B sólo sabe que desea recibir mensajes de algún proceso no identificado”

Identificación de Procesos

- ¿Qué formas hay de identificar los procesos que envían y reciben mensajes?



- **Indirecta:** “El proceso A sólo sabe que desea enviar al canal C, y el proceso B sólo sabe que desea recibir mensajes del canal C”

Canales

- **Canales**: Un canal es un medio de comunicación entre procesos que permite hacer el intercambio de mensajes.

- Lo veremos como un tipo abstracto de datos ignorando los detalles de su implementación.

- Podemos entender un **Canal** como un **Buffer**:

  - Si el canal es **asincrónico**, entonces la capacidad del Buffer es ilimitada

  - Si el canal es **sincrónico**, entonces la capacidad del Buffer es 0.

Canales Sincrónicos

- Asumamos que tenemos canales sincrónicos (el send es “bloqueante”)

```
process P1:
  // computo costoso
  // en funcion de x
```

```
process P2:
  // computo costoso
  // en funcion de y
```

```
process P3:
  // computo en funcion de los
  // resultados de P1 y P2
```

Canales Sincrónicos

```
global Channel c1 = new Channel()
global Channel c2 = new Channel()

process P1: {
  // computo costoso
  // en funcion de un X
  c1.send(r1);
}
```

```
process P2: {
  // computo costoso
  // en funcion de un Y
  c2.send(r2);
}
```

```
process P3: {
  r1 = c1.receive();
  r2 = c2.receive();
  print(r1 + r2);
}
```

Canales Sincrónicos

```
global Channel c1 = new Channel()
global Channel c2 = new Channel()

process P1: {
  // computo costoso
  // en funcion de un X
  c1.send(r1);
}
```

```
process P2: {
  // computo costoso
  // en funcion de un Y
  c2.send(r2);
}
```

```
process P3: {
  r1 = c1.receive();
  r2 = c2.receive();
  print(r1 + r2);
}
```

- ¿Cómo modificamos el ejemplo si P3 debe enviar a P1 y P2 los valores X e Y?

Canales Sincrónicos

```
global Channel c1 = new Channel()
global Channel c2 = new Channel()

process P1: {
  x = c1.receive();
  // computo costoso
  // en funcion de x
  c1.send(r1);
}
```

```
process P2: {
  y = c2.receive();
  // computo costoso
  // en funcion de y
  c2.send(r2);
}
```

```
process P3: {
  c1.send(X);
  c2.send(Y);
  r1 = c1.receive();
  r2 = c2.receive();
  print(r1 + r2);
}
```

Canales Sincrónicos

- ¿Qué cambia entre estas dos posibles formas de escribir el proceso P3?

```
global Channel c1 = new Channel();
global Channel c2 = new Channel();

process P3: {
  c1.send(X);
  c2.send(Y);
  r1 = c1.receive();
  r2 = c2.receive();
  print(r1 + r2);
}
```

```
process P3: {
  c1.send(X);
  r1 = c1.receive();
  c2.send(Y);
  r2 = c2.receive();
  print(r1 + r2);
}
```

Canales Sincrónicos

```
global Channel c1 = new Channel();
global Channel c2 = new Channel();

 process P3: {
    c1.send(X);
    c2.send(Y);
    r1 = c1.receive();
    r2 = c2.receive();
    print(r1 + r2);
 }
```

```
process P3: {
   c1.send(X);
   r1 = c1.receive();
   c2.send(Y);
   r2 = c2.receive();
   print(r1 + r2);
}
```

- La segunda opción **no aprovecha** la concurrencia dado que espera a recibir el resultado de P1 antes de disparar el cómputo del proceso P2.

Canales Asincrónicos

- ¿Qué pasaría si los canales son asincrónicos (el send no es bloqueante) en lugar de sincrónicos?

```
process P3: {
   c1.send(X);
   c2.send(Y);
   r1 = c1.receive();
   r2 = c2.receive();
   print(r1 + r2);
}
```

Canales Asincrónicos

- ¿Qué pasaría si los canales son asincrónicos (el send no es bloqueante) en lugar de sincrónicos?

```
process P3: {
   c1.send(X);
   c2.send(Y);
   r1 = c1.receive();
   r2 = c2.receive();
   print(r1 + r2);
}
```

1. P3 envía X por c1
2. P3 envía Y por c2
3. P3 lee “X” por c1 (P1 no lo leyó)
4. P3 supone que “X” es r1 (**ERROR**)

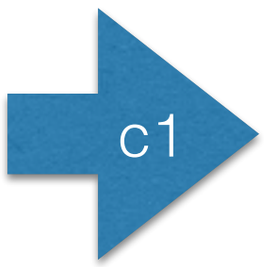
Canales Asincrónicos

- *Escribir dos procesos A y B tales que el primero genera un número aleatorio, y se lo envía al segundo que al recibirlo lo muestra por pantalla.*

Canales Asincrónicos

- *Escribir dos procesos A y B tales que el primero genera un número aleatorio, y se lo envía al segundo que al recibirlo lo muestra por pantalla.*

```
global Channel c1 = new Channel();

process A: {
   r = random();
   c1.send(r);
}
```




```
process B: {
   int r = c1.receive();
   print(r);
}
```

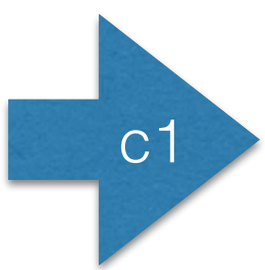
Canales Asincrónicos

- *Extender la solución para que el proceso B lea un número introducido por el usuario indicando la cantidad de números aleatorios a generar*

- *Ese valor debe ser enviar al proceso A quien debe generar la cantidad de números pedida*

- *El proceso B debe mostrar todos los números por pantalla.*

```
global Channel c1 = new Channel();

process A: {
   r = random();
   c1.send(r);
}
```




```
process B: {
   int r = c1.receive();
   print(r);
}
```

Canales Asincrónicos

- Proceso B lee el input del usuario, envía la cantidad al proceso A, quien genera tantos números aleatorios como fueron pedidos.

```
global Channel c1 = new Channel();
global Channel c2 = new Channel();

   process B: {
     String str = readInput();
     int n = parseInt(str);
     c2.send(n);
     repeat (n) {
         int r = c1.receive();
      print(r);
     }
   }
```

```
process A: {
  int n = c2.receive();
  repeat (n) {
      int r = random();
      c1.send(r);
   }
}
```

Canales Asincrónicos

- ¿Cómo podemos extender esta solución para admitir tantos pedidos concurrentes como sean posibles?

```
global Channel c1 = new Channel();
global Channel c2 = new Channel();

   process B: {
     String str = readInput();
     int n = parseInt(str);
     c2.send(n);
     repeat (n) {
         int r = c1.receive();
      print(r);
     }
   }
```

```
process A: {
  int n = c2.receive();
  repeat (n) {
      int r = random();
      c1.send(r);
   }
}
```

Canales Asincrónicos + Múltiples Threads

```
global Channel c1 = new Channel();
global Channel c2 = new Channel();
```

```
process Client: {
  String str = readInput();
  int n = parseInt(str);
  c2.send(n);
  repeat (n) {
    int r = c1.receive();
    print(r);
  }}
```

```
process Server: {
  while (true) {
    int n = c2.receive();
    thread {
      repeat (n) {
        int r = random();
        c1.send(r);
      }
    }
  }}
```

Canales Asincrónicos + Múltiples Threads

- ¿Cómo podemos extender la implementación para que el Servidor provea números aleatorios en un rango provisto por el Cliente?

Primer Intento

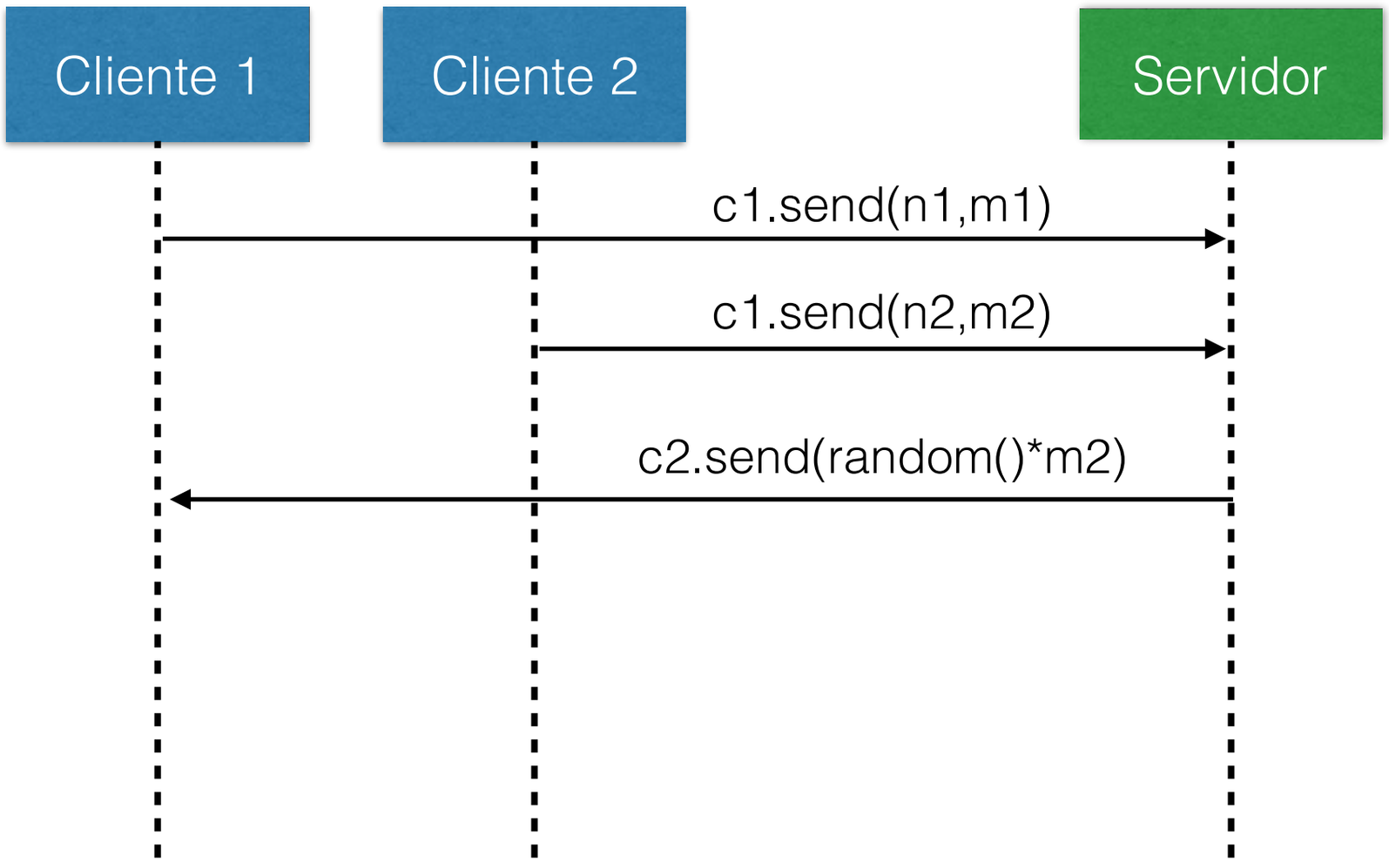
```
global Channel c1 = new Channel();
global Channel c2 = new Channel();

process Client: {
  req = record();
  req.n = parseInt(read());
  req.m = parseInt(read());
  c1.send(req);

  repeat (n) {
    print(c2.receive());
  }
}
```

```
process Server: {
  while (true) {
    req = c1.receive();
    thread {
      repeat (req.n) {
        c2.send(random()*req.m);
      }
    }
}}
```

- **Pregunta**: ¿Qué problema tiene esta implementación?

Primer Intento

- **Pregunta**: ¿Qué problema tiene esta implementación?



- Los clientes compiten entre ellos leyendo resultados que son para otros clientes (Ejemplo: Cliente 1 termina leyendo un resultado que era para Cliente 2)

Envío de Canales

- **Pregunta**: ¿Qué problema tiene esta implementación?
  - Los clientes compiten entre ellos leyendo resultados que son para otros clientes (Ejemplo: Cliente 1 termina leyendo un resultado que era para Cliente 2)
- Necesitamos un canal de respuesta dedicado para cada cliente
- Dependiendo del protocolo, el canal puede proponerlo el cliente o elegirlo el servidor

Solución

```
global Channel c1 = new Channel();


 process Client: {
   req = record();
   req.n = parseInt(read());
   req.m = parseInt(read());
   req.c = new Channel();
   c1.send(req);

   repeat (n) {
     print(req.c.receive());
   }
}
```

```
process Server: {
  while (true) {
    req = c1.receive();
    thread {
      repeat (req.n) {
        req.c.send(random()*req.m);
      }
    }
  }}
```

- El cliente crea el canal que usará para esperar resultados del servidor.

- El cliente le envía el canal al servidor, que lo usa para reportarle los resultados

Esquema Cliente-Servidor

- El Cliente y el Servidor son procesos que se comunican con canales, no con procesos (identificación indirecta)

- Existen potencialmente muchas posibles instancias de procesos Cliente, pero una única instancia de Servidor.

- El Servidor debe poder atender múltiples clientes simultáneamente (crea un thread por cada cliente)

- El Cliente suministra tantos canales auxiliares como necesite el protocolo.

Resumen

- Mensajes entre procesos es una alternativa al empleo de memoria compartida en programas concurrentes.
- Dado un canal, un proceso puede realizar envío de mensajes (send) o recepción de mensajes (receive)
- Los canales pueden ser sincrónicos o asincrónicos.
- La identificación puede ser directa (identificando procesos) o indirecta (identificando canales)
- Esquema Cliente-Servidor

LOONEY TUNES

That's all Folks.