Lecture 6

Introduction to Message Passing

Message Passing

- Introduction to message passing
- JR's message passing model
  - Operations
  - Asynchronous message passing
  - Synchronous message passing

Shared Variables?

- So far we considered synchronisation mechanisms based on shared variables
  - Concurrent programs require hardware in which processors share memory
    - SMP
    - What about NUMA or similar architectures?
  - Networked (distributed) architectures are not based on shared memory
- Message passing is the natural model for distributed systems and alike

Shared Variables?

- Shared state is the main source of synchronisation problem – critical section
  - Locks
  - Semaphores
  - Monitors
- Can we throw away the shared state?
  - Message passing

Overview of Message Passing

- One process sends a message
- Another process awaits for a message

- We will consider two dimensions of this approach:
  - What form of synchronisation is required
  - What form of process naming is involved in message passing

Synchronisation

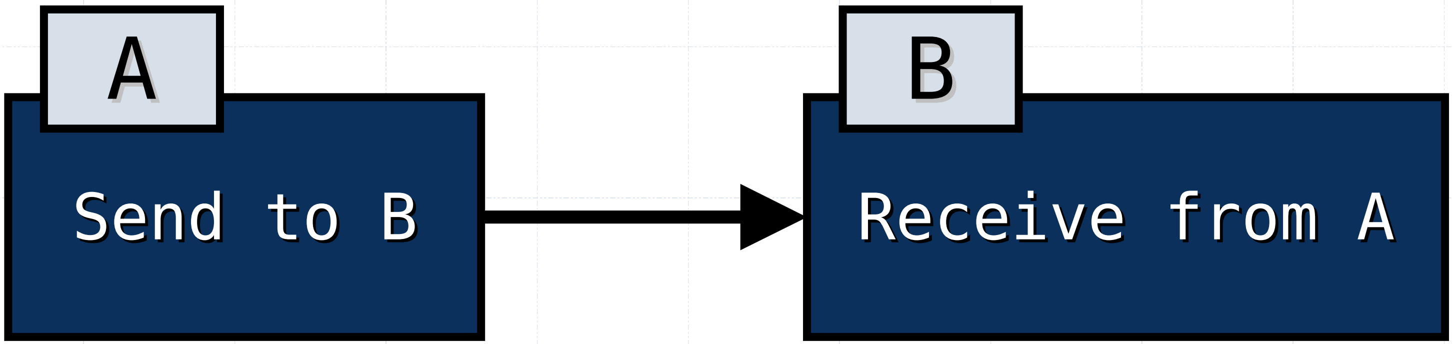
- Consider the behaviour of the sender of a message
  - Asynchronous send
    - Send and continue working (e-mail, SMS)
  - Synchronous send
    - Send and wait for the message to be received (fax)
  - Rendezvous / Remote invocation
    - send and wait for reply (phone call)

Examples

- JR combines all three types of MP
- Erlang has asynchronous MP
- Ada has rendezvous
  - Previously used at Chalmers as a main teaching language
- Java has libraries
  - Sockets – asynchronous message passing
  - RMI – can be seen as synchronous MP
    - Normal method invocation can also be seen as synchronous MP

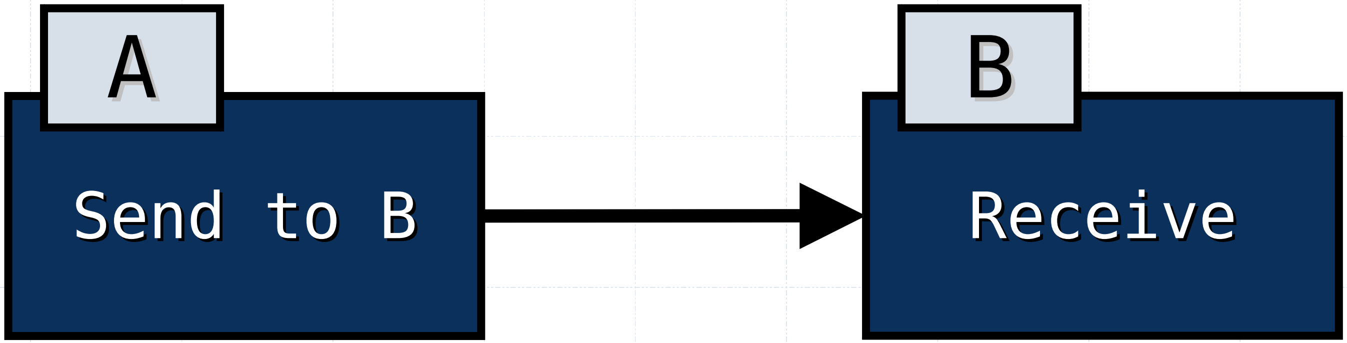
Naming

- How do sender and receiver refer to each other when message passing is used?
- Three most common disciplines
  - Direct symmetric



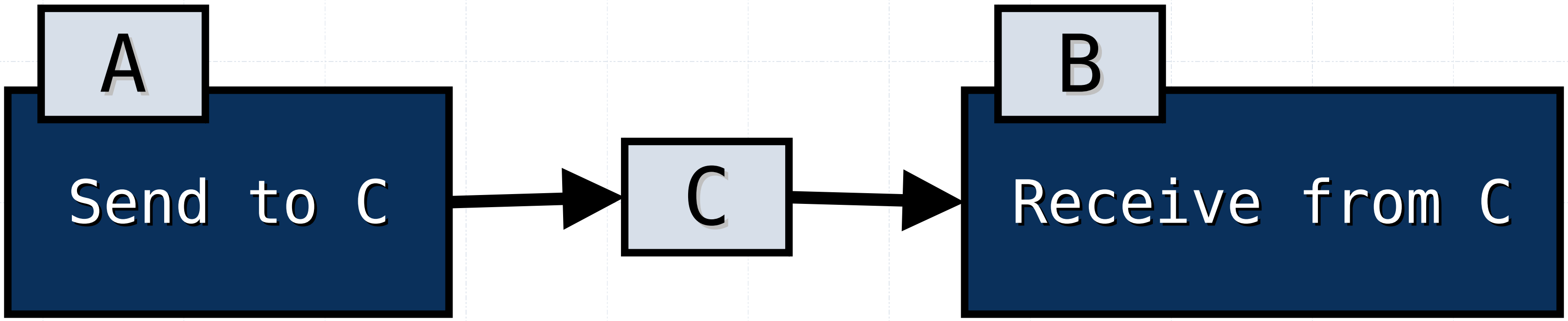
Naming

- Three most common disciplines – cont.
  - Direct asymmetric
    - Erlang
    - Sockets



Naming

- Three most common disciplines – cont.
  - Indirect
    - Naming a process is not always convenient
    - Naming an intermediary can be more flexible
    - A channel, service name, or a mailbox
    - Potentially many-to-many communication



Message Passing in JR

- Operations
  - Key JR extension of Java
- Servicing operations
  - Method
  - Receive statement
  - Input statement
- Invocations
  - Call
  - Send

Message Passing in JR

- Operations
  - Key JR extension of Java
- Servicing operations
  - Method
  - Receive statement
  - Input statement
- Invocations
  - Call
  - Send

} Today

Next lecture

Operations

- Generalisation of methods
- Syntax: keyword op

```
private op void buy();

public op E acquire(int n);
```

- Specifies parameter and return types
  - Parameter names are unimportant
- Can be serviced in several ways

Servicing Operations 1

- Methods
  - Full syntax

```
private op void buy();

private void buy() {
   while (true) {
     window.flash("Buy @ Cremona!");
     JR.nap(buy_pause);
   }
}
```

Servicing Operations 1

- Methods
  - Shorthand

```
private op void buy() {
   while (true) {
     window.flash("Buy @ Cremona!");
     JR.nap(buy_pause);
   }
}
```

Op-methods – Call

- Ordinary Java method call

```
...
Cremona c = new Cremona();
...
c.buy();
...
```

- Explicit `call` statement

```
...
call c.buy();
...
```

Op-methods – Send

- Asynchronous `send` statement
  - Starts a new process
  - Runs the servicing method in the new process
  - The return value is discarded
  - The caller continues execution independently

```
...
Cremona c = new Cremona();
...
send c.buy();
...
```

Dynamic Process Creation

- Process declarations are only a shorthand

```
private process buy {
   //Code
}
```

```
private op void buy();
private void buy() {
   //Code
}
public Constructor(…) {
   …
   send buy();
}
```

Process Families

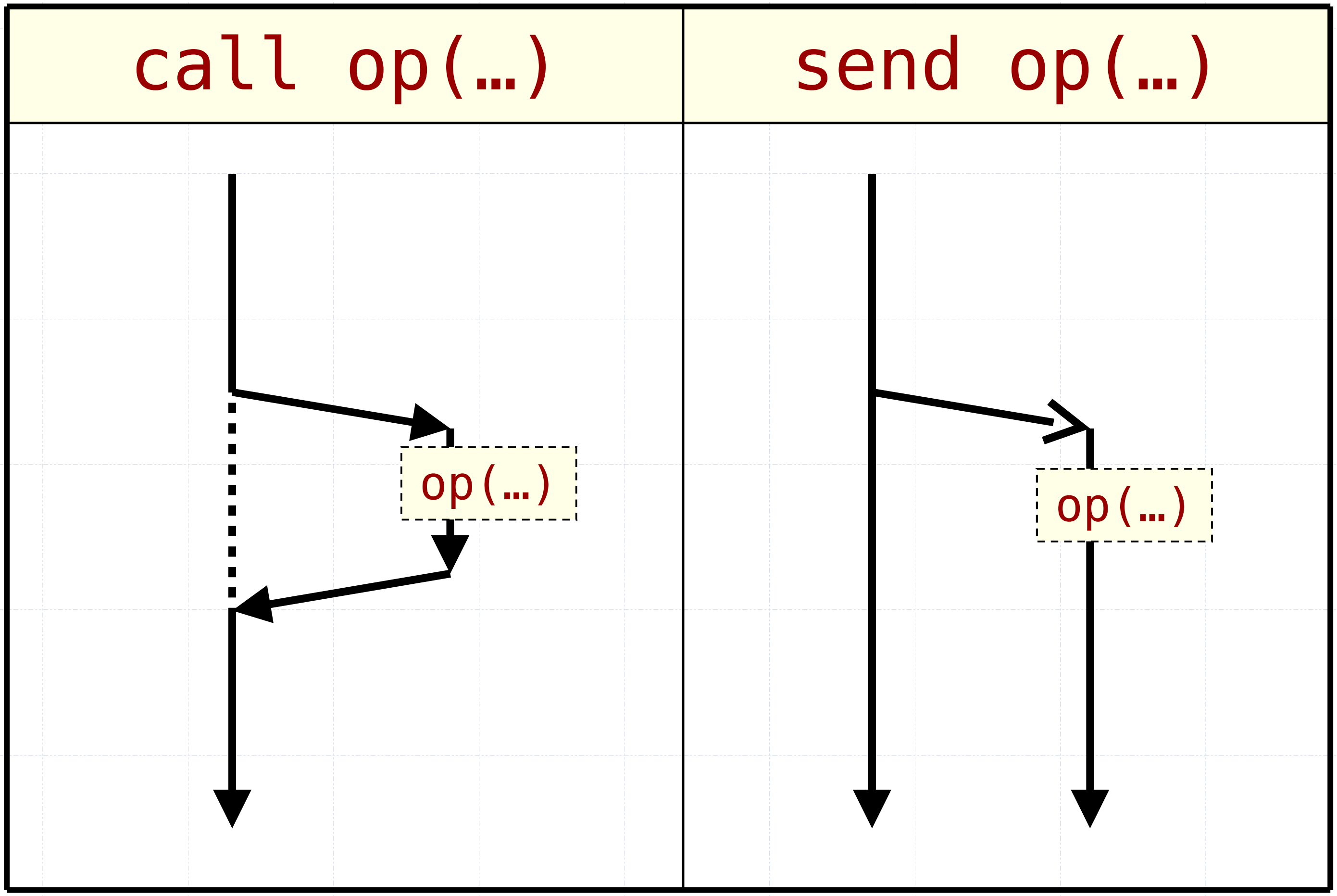
- Process families are also a shorthand

```
private process buy ((quantifier),
                     …,
                     (quantifier)) {
   //Code
}
```

```
public Constructor(…) {
   …
   for (quantifier)
      …
         for (quantifier)
            send buy();
}
```

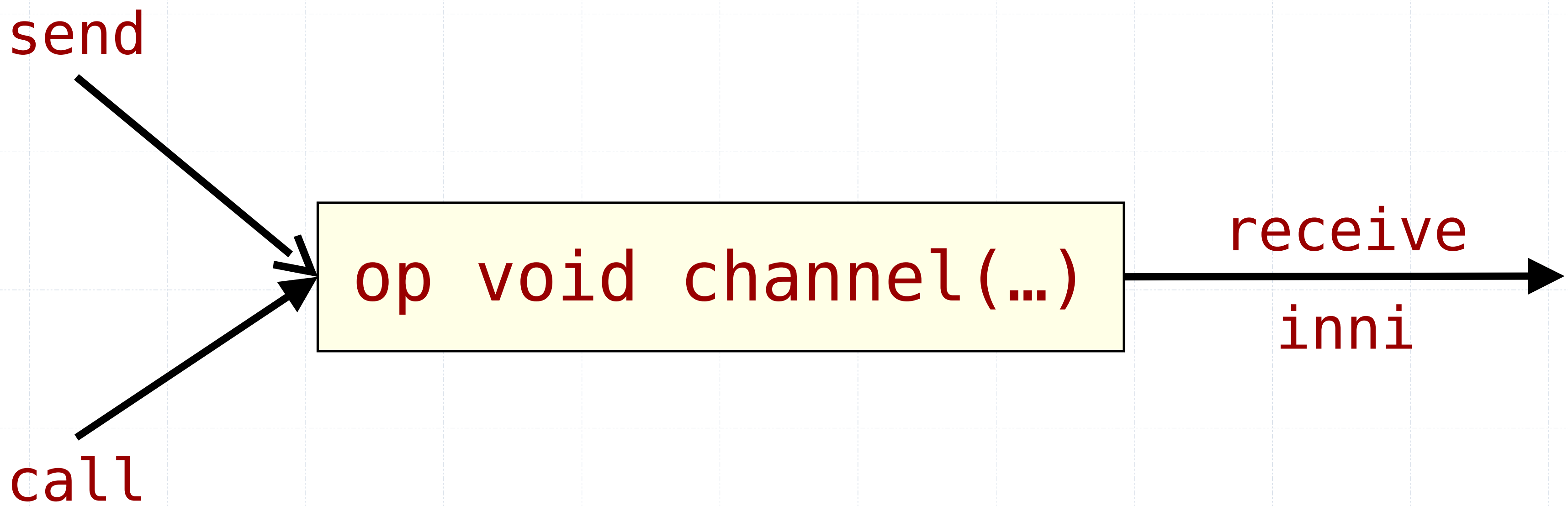
Op-methods: Send vs Call

- Operation `op(…)` serviced by a method



Servicing Operations 2

- Message queues – channels
  - No corresponding method, but
  - Unbounded buffer of messages
  - Return type must be `void`



Channels – Receive

- `receive` statement

```
receive op(x1, ..., xn);
```

- Wait for a message on the named channel `op`
- Atomically remove the first message and put the fields of the message into the variables `x1, …, xn`

Channels – Send

- send statement

```
send op(exp1, ..., expn);
```

- Evaluate the expressions exp1,…,expn and produce a message M
- Atomically append M to the end of the named channel op
- Send is a non-blocking action
    - Asynchronous message passing

Example 1

Book notation `chan ch(int)`

```
private op void ch(int x);

private process p {
   send ch(1);
   send ch(2);
}

private process q {
   int x,y;
   receive ch(x);
   receive ch(y);
}
```

Two sends from the same source implies …

x will get 1, and y will get 2

Example 2

```
private op void ch1(int x);
private op void ch2(int x);
```

```
private process p {
    send ch1(1);
    send ch2(2);
}

private process q {
    int x,y;
    receive ch1(x);
    receive ch1(y);
}
```

```
private process r {
    send ch1(3);
    send ch2(4);
}

private process s {
    int x,y;
    receive ch2(x);
    receive ch2(y);
}
```

Expressive Power

- Semaphores and monitors
  - Equally expressive
  - Any synchronisation with await statement
- Asynchronous message passing vs. XXX
  - Can we implement semaphores?
  - Can we implement monitors?
    - Important theoretical question
    - An illustrative example, but not normal practice
    - Implementing a low-level language construct in a high-level language is not normally a good idea

There is No Semaphore

Semaphore	Channel

`sem s = N`	`op void s();`
	`for(int x=0;x<N;x++)`
	`   send s();`

`P(s);`	`receive s();`
`V(s);`	`send s();`

Expressive Power

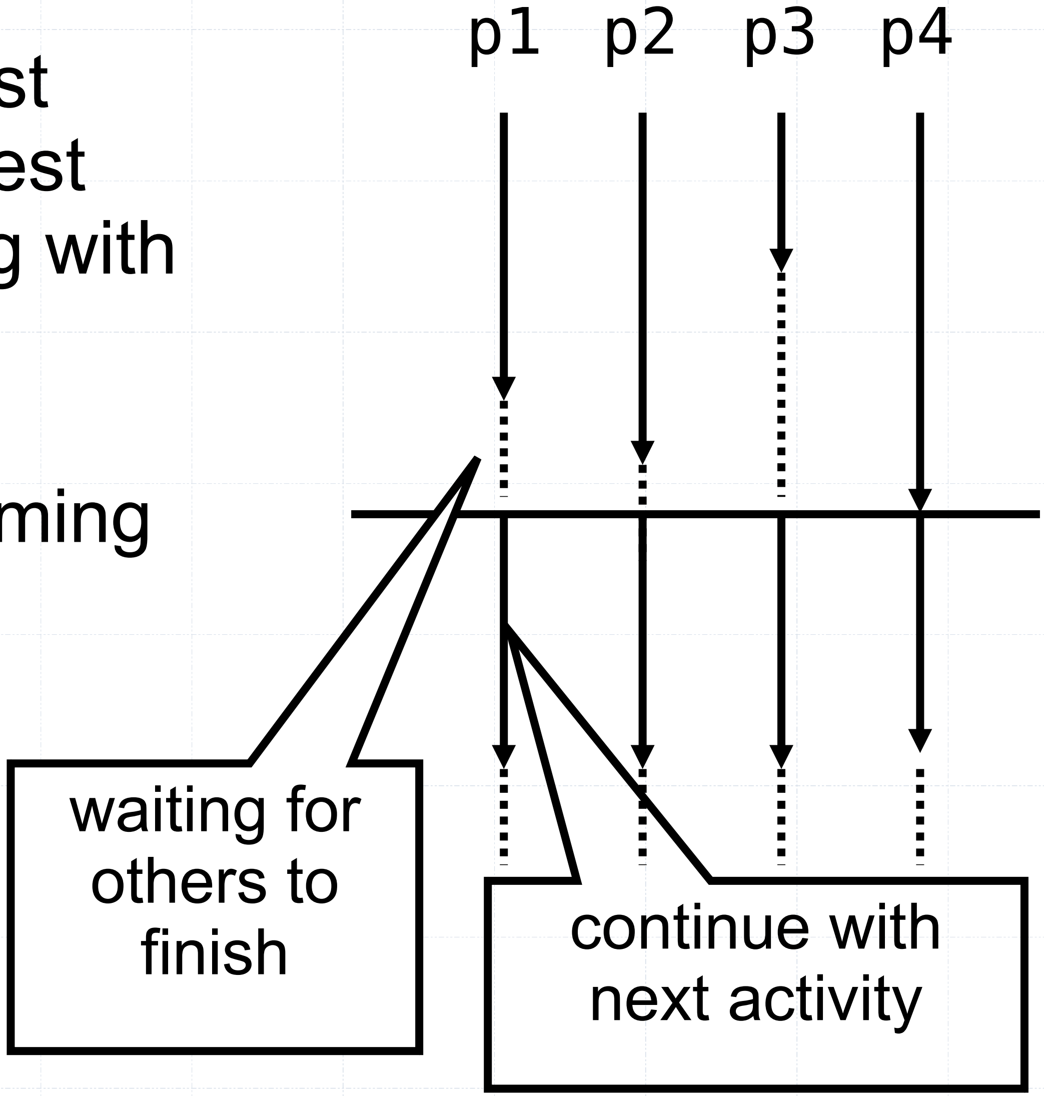
- Semaphores are asynchronous channels without values
- JR (message passing) is implemented using monitors in Java

$$\Downarrow$$

- The same expressive power
  - Can implement any await statement
  - Important theoretical result

Barrier Synchronisation Revisited

- N processes must wait for the slowest before continuing with the next activity
- Widely used in parallel programming



Barrier Synchronisation Exercise 2

- Ball freezing with two semaphores?

```
op void done();
op void go();
```

```
process ball ((int i=0;i<4;i++)) {
   //move
   send done();
   receive go();
}
```

```
process coordinator {
   for(int i=0;i<4;i++)
      receive done();
   for(int i=0;i<4;i++)
      send go();
}
```

Barrier Synchronisation Exercise 2

- Ball freezing with two semaphores?

```
op void done();
op void go();
```

```
process ball ((int i=0;i<4;i++)) {
   //move
   send done();
   receive go();
}
```

```
process coordinator {
   for(int i=0;i<4;i++)
      receive done();
   for(int i=0;i<4;i++)
      send go();
}
```

A fast ball can steal go

Barrier Synchronisation Exercise 2

- Ball freezing with N+1 semaphores

```
op void done();
cap void() go[]; //some init
```

```
process ball ((int i=0;i<4;i++)) {
   //move
   send done();
   receive go[i]();
}
```

```
process coordinator {
   for(int i=0;i<4;i++)
      receive done();
   for(int i=0;i<4;i++)
      send go[i]();
}
```

Operation Capabilities

- A reference to an operation
  - Just as an ordinary Java object reference
  - Usage
    - Variables
    - Passing as parameters
    - Dynamic operation creation
  - Example:

```
private op void buy();

private cap void() ref = buy;
```

Operation Capabilities

- Example
  - Dynamic operation creation
  - Array of operations (semaphores)

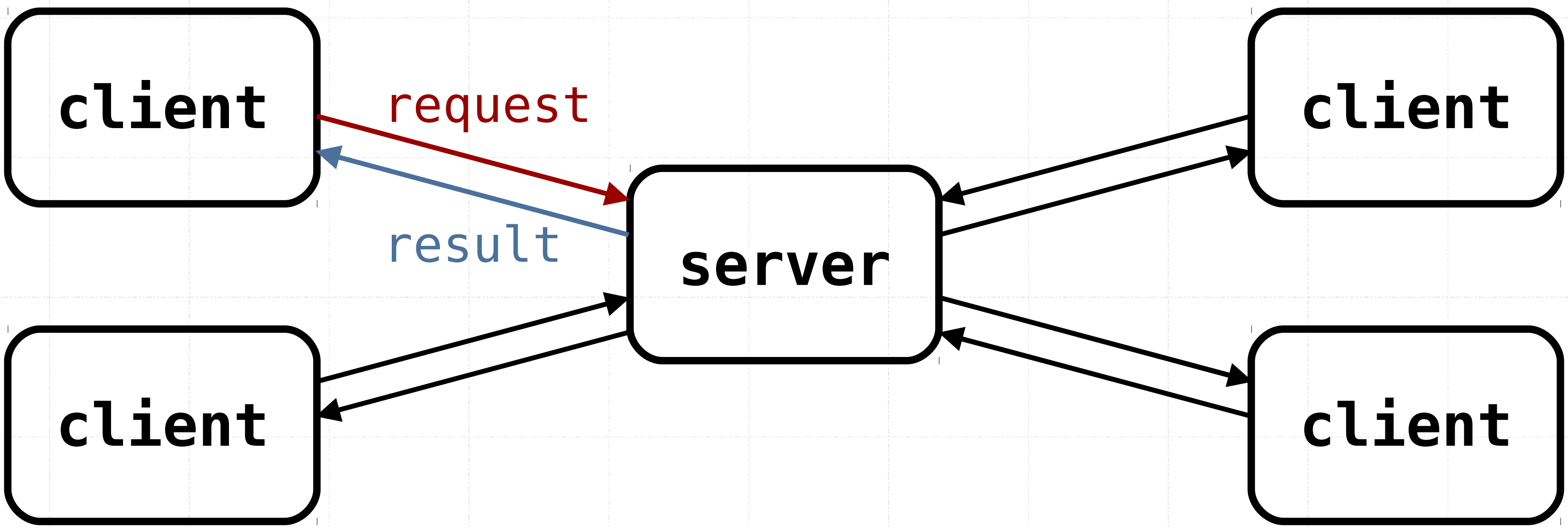
```
private cap void() go[];

public Ball() {
   go = new cap void()[4];
   for(int i=0;i<4;i++)
      go[i] = new op void();
}
```

Operation Capabilities

- Capabilities can be tested for equality
  - Both `==` and `!=` work
  - Only the type signatures must match
- Special cases
  - Capabilities are references ⇒ `null` is a valid capability value
  - Special operation value `noop` is also provided
    - Infinite sink
    - Receiving from `noop` blocks forever

Client-Server Interaction

- Common asynchronous communication pattern
  - For example: a web server handles requests for web pages from clients (web browsers)



Simple Client-Server Model

- First attempt

```
op void request(…);
op void result(…);
```

```
process server {
   while (true) {
      receive request(…);
      // process request
      send result(…);
   }
}
```

```
process client {
   send request(…);
   // possibly do
   // something else
   receive result(…);
}
```

Simple Client-Server Model

- First attempt

```
op void request(…);
op void result(…);
```

```
process server {
   while (true) {
      receive request(…);
      // process request
      send result(…);
   }
}
```

Many clients:
Who gets the result?

```
process client {
   send request(…);
   // possibly do
   // something else
   receive result(…);
}
```

A Private Operation

- We need to pass a reference to a private reply channel
  - Operation capabilities are operation references

```
op void request(cap void(resultType) res,
                    … );
```

  - Each client needs to create a private operation as a reply channel

Private Channel

```
process server {
   while (true) {
      receive request(replyChannel, …);
      // process request
      send replyChannel(…);
   }
}

process client {
   op void myReplyChannel(resultType);
   send request(myReplyChannel, …);
   // possibly do something else
   receive myReplyChannel(…);
}
```

Resource Allocation – Single

- A controller controls access to copies of some resource
- Clients make requests to take (acquire) or return (release) one resource
  - A request should only succeed if there is a resource available,
  - Otherwise the request must block
- Adapt the passing the condition solution
  - with explicit queue of requests instead of condition variable

Resource Allocation

```
public class ResourceAllocator<E> {

   public enum Request {Allocate, Release};

   public op void request(cap void(E),
                          Request,
                          E);


   private Queue<E> units =
      new ArrayDeque<E>();
   private Queue<cap void(E)> pending =
      new ArrayDeque<cap void(E)>();

//next slide
```

Resource Allocation

```
private process server {
   cap void(E) rc; Request action; E unit;
   while (true) {
      receive request(rc, action, unit);
      if (action == Request.Allocate)
         if (units.isEmpty())
            pending.add(rc);
         else
            send rc(units.remove());
      else
         if (pending.isEmpty())
           units.add(unit);
         else
            send (pending.remove())(unit); }}
```

Channels – Call

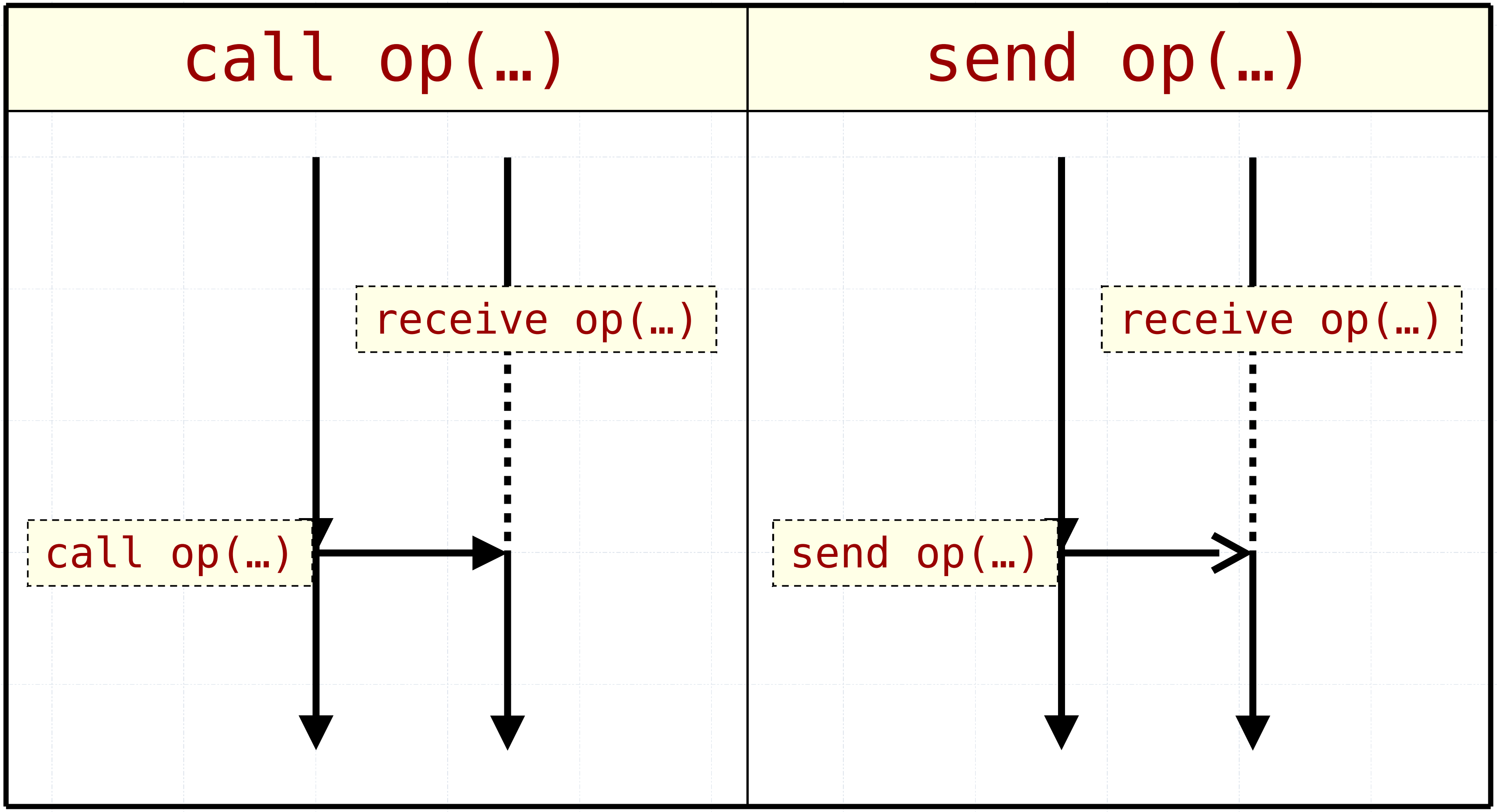
- `call` statement

```
call op(exp1, ..., expn);
```

- Evaluate the expressions `exp1,…,expn` and produce a message M
- Atomically append M to the end of the named channel `op`, and
- Wait until the message is received
  - Synchronous message passing

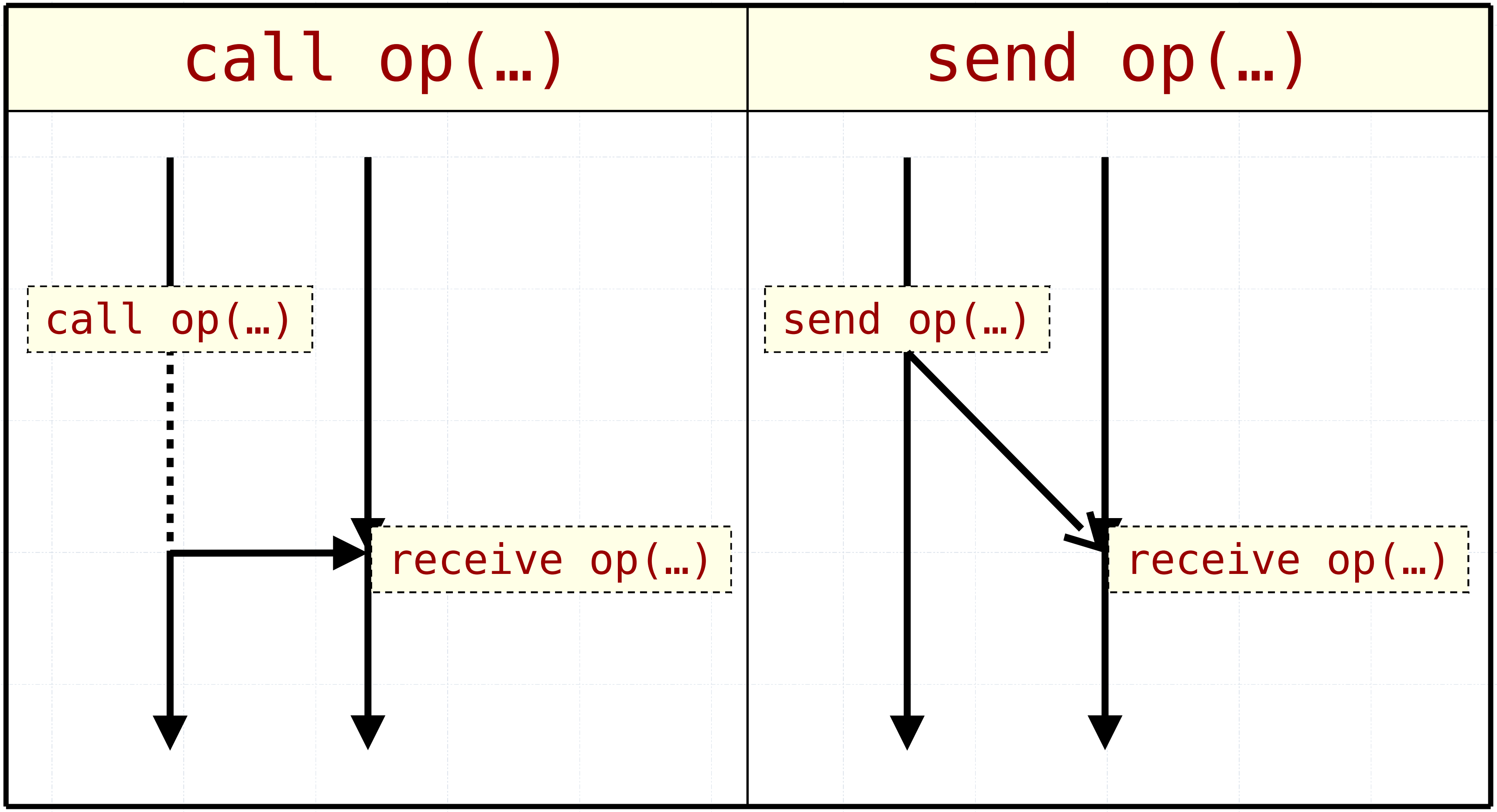
Channels: Send vs Call

- Operation `op(…)` serviced by a channel



Channels: Send vs Call

- Operation `op(…)` serviced by a channel

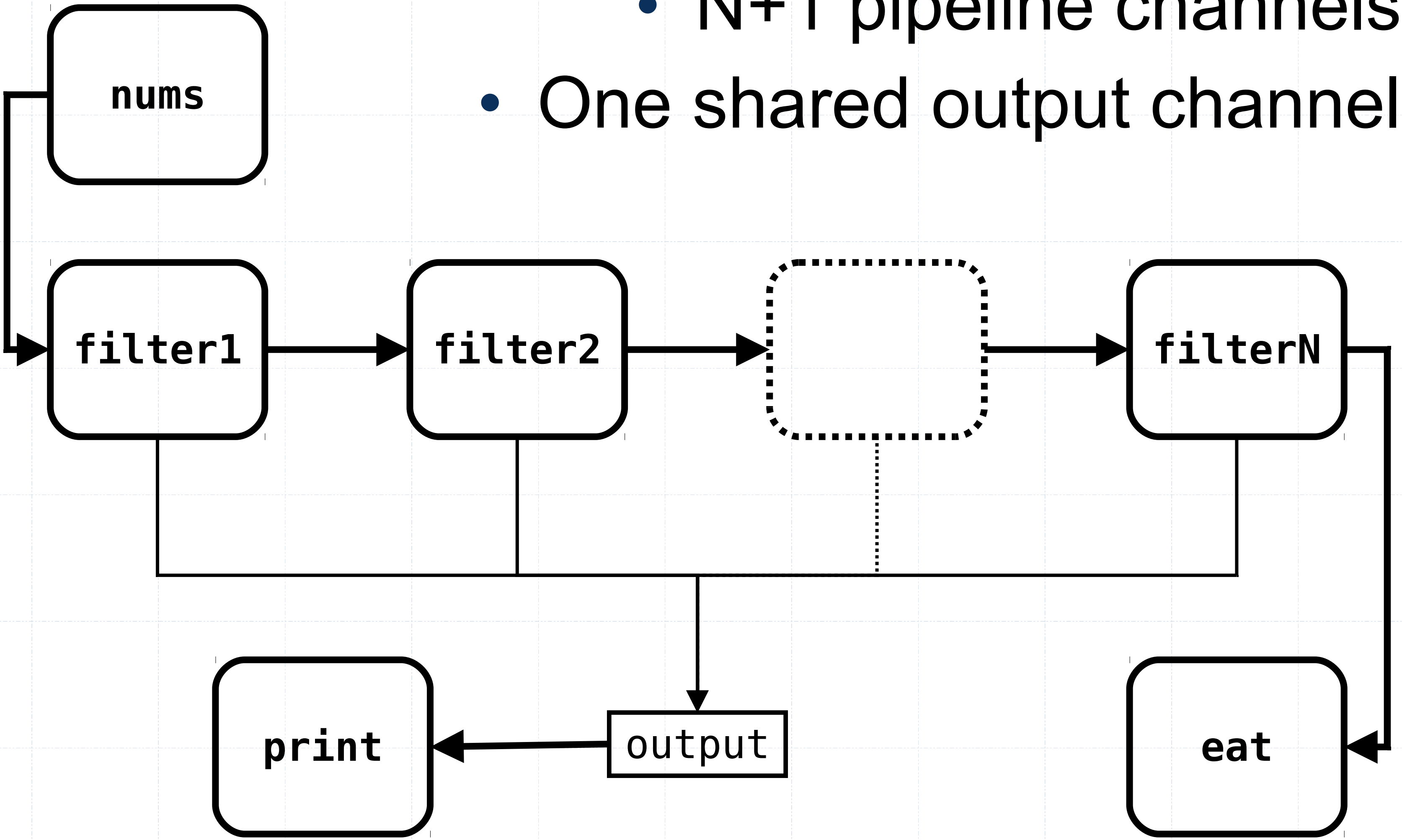


Sieve of Eratosthenes

- Starting with the sequence 2,3,4,…
- A pipeline of sieves (filters) is arranged in a line
- Each filter outputs the first number received from left (a prime!)
- For each subsequent number received from the left:
  - discard it if divisible by the first number
  - pass to the right otherwise

Architecture

- N+1 pipeline channels
- One shared output channel



Constructor and Print

```
public Sieve(int N) {
   this.N = N;
   pipeline = new cap void(int)[N+1];
   for(int i=0;i<(N+1);i++)
      pipeline[i] = new op void(int);
}

public process print {
   int number;
   while (true) {
      receive output(number);
      System.out.println(number);
   }
}
```

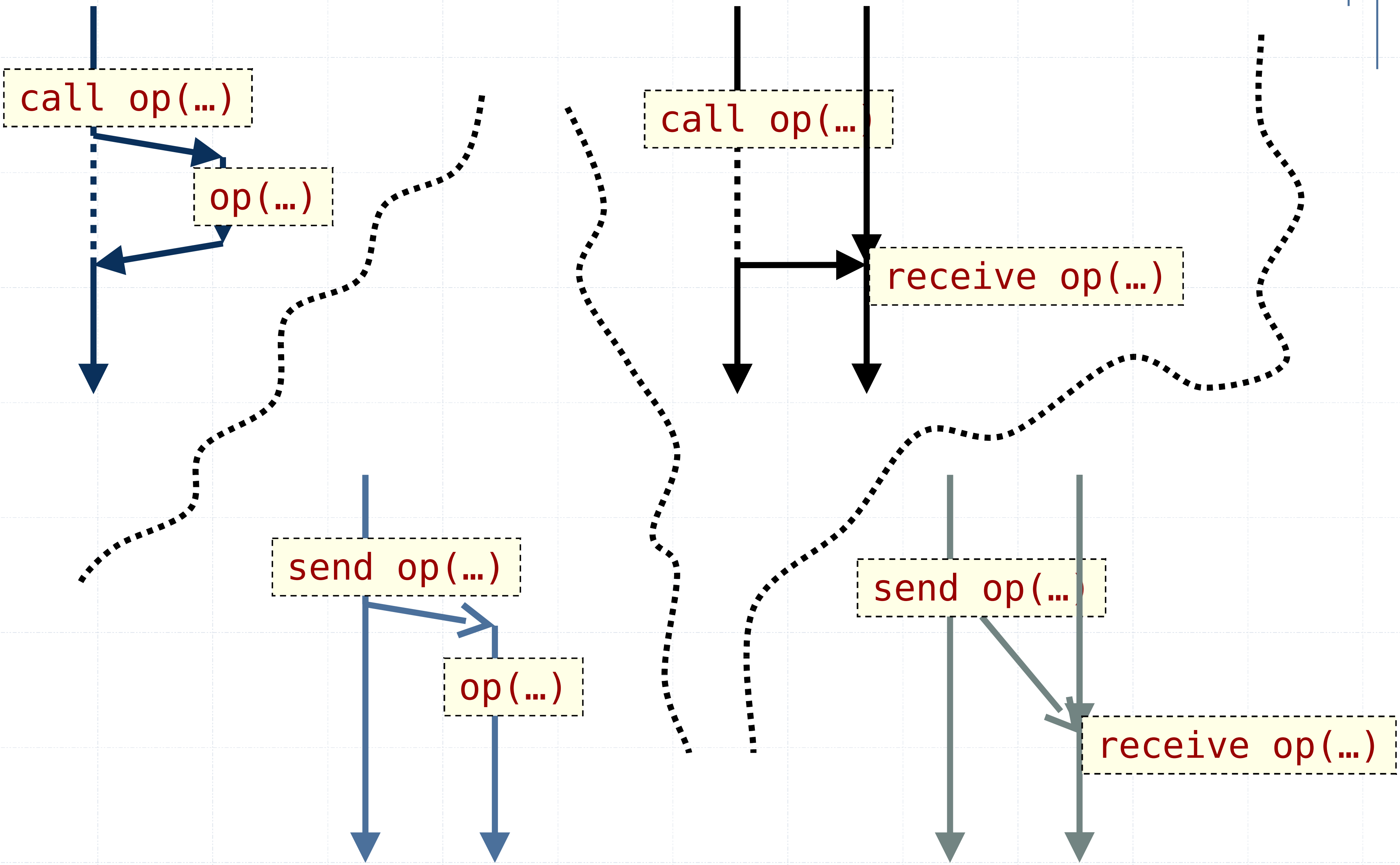
The Ends of the Pipeline

```
public process nums {
   for(int i=3;i<(20*N);i+=2) {
      call pipeline[0](i);
   }
}

public process eat {
   int number;
   while (true)
      receive pipeline[N](number);
}
```

The Filters

```
public process filter((int i=0;i<N;i++)) {
   int prime, number;

   receive pipeline[i](prime);
   call output(prime);

   while (true) {
      receive pipeline[i](number);
      if (number%prime > 0)
         call pipeline[i+1](number);
   }
}
```

Summary



Summary

- Operations
  - Methods
  - Channels
- Invocations
  - Asynchronous
  - Synchronous

- Next time
  - Remote invocation / Rendezvous