# Message Passing

Generalizes semaphores, P() and V()


0

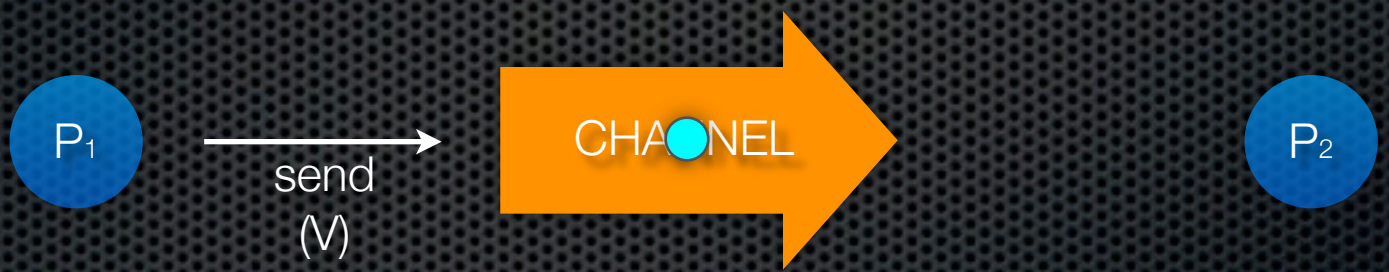
# Message Passing

Generalizes semaphores, P() and V()


0

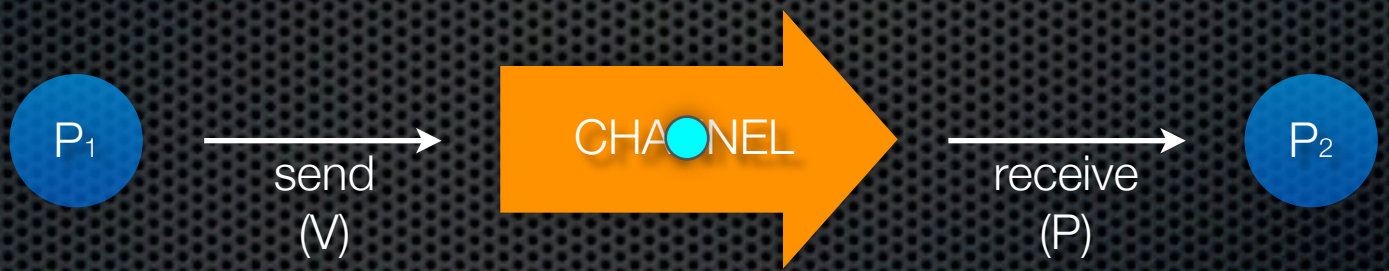
# Message Passing

Generalizes semaphores, P() and V()



# Message Passing

Generalizes semaphores, P() and V()



# Message Passing

Generalizes semaphores, P() and V()


0

# Channel

Unbounded queue of messages

```
chan name(id1: type1; ...; idN: typeN)
```

*field*

*(with libraries, messages are just streams of bytes, possibly with self-describing tags to indicate types of fields)*

# Message Passing Primitives

# Message Passing Primitives

```
send name(expr1, ..., exprN)
```

# Message Passing Primitives

```
send name(expr1, ..., exprN)
```

- Types and number of fields must match

# Message Passing Primitives

```
send name(expr1, ..., exprN)
```

- Types and number of fields must match
- Effect:

# Message Passing Primitives

```
send name(expr1, ..., exprN)
```

- Types and number of fields must match
- Effect:
  - Evaluate the expressions and produce a message M

# Message Passing Primitives

```
send name(expr1, ..., exprN)
```

- Types and number of fields must match
- Effect:
  - Evaluate the expressions and produce a message M
  - Atomically append M to the end of the named channel

# Message Passing Primitives

```
send name(expr1, ..., exprN)
```

- Types and number of fields must match
- Effect:
  - Evaluate the expressions and produce a message M
  - Atomically append M to the end of the named channel

➡ send is nonblocking (asynchronous)

# Message Passing Primitives

# Message Passing Primitives

```
receive name(var1, ..., varN)
```

# Message Passing Primitives

```
receive name(var1, ..., varN)
```

- Again, types and number of fields must match

# Message Passing Primitives

```
receive name(var1, ..., varN)
```

- Again, types and number of fields must match
- Effect:

# Message Passing Primitives

```
receive name(var1, ..., varN)
```

- Again, types and number of fields must match
- Effect:
  - Wait for a message on the named channel

# Message Passing Primitives

```
receive name(var1, ..., varN)
```

- Again, types and number of fields must match
- Effect:
  - Wait for a message on the named channel
  - Atomically remove first message and put the fields of the message into the variables

# Message Passing Primitives

```
receive name(var1, ..., varN)
```

- Again, types and number of fields must match
- Effect:
  - Wait for a message on the named channel
  - Atomically remove first message and put the fields of the message into the variables

➡ receive is blocking (synchronous)

# Naming

# Naming

- Who has a name?

# Naming

- Who has a name?
    - Processes?

# Naming

- Who has a name?
  - Processes? No

# Naming

- Who has a name?
  - Processes? No
  - Channels?

# Naming

- Who has a name?
  - Processes? No
  - Channels? Yes

# Example

# Example

```
chan ch(int)
```

# Example

```
chan ch(int)


process A:
        send ch(1)
        send ch(2)
```

# Example

```
chan ch(int)


process A:              process B:
        send ch(1)              receive ch(x)
        send ch(2)              receive ch(y)
```

# Example

```
chan ch(int)

process A:                  process B:
        send ch(1)                  receive ch(x)
        send ch(2)                  receive ch(y)
```

- x will contain 1 and y will contain 2

# Example

```
chan ch(int)


process A:                  process B:
        send ch(1)                  receive ch(x)
        send ch(2)                  receive ch(y)
```

- x will contain 1 and y will contain 2
- Order of messages from SAME source is the order of the sends

# Example

# Example

```
chan ch1(int), ch2(int)

process A:
  send ch1(1)
  send ch2(2)

process B:
  receive ch1(x)
  receive ch1(y)

process C:
  send ch1(3)
  send ch2(4)

process D:
  receive ch2(u)
  receive ch2(v)
```

# Example

```
chan ch1(int), ch2(int)

process A:
  send ch1(1)
  send ch2(2)

process B:
  receive ch1(x)
  receive ch1(y)

process C:
  send ch1(3)
  send ch2(4)

process D:
  receive ch2(u)
  receive ch2(v)
```

What is received now?

# Example

```
chan ch1(int), ch2(int)

process A:
  send ch1(1)
  send ch2(2)

process B:
  receive ch1(x)
  receive ch1(y)

process C:
  send ch1(3)
  send ch2(4)

process D:
  receive ch2(u)
  receive ch2(v)
```

What is received now?

- x will get 1 or 3 and y will get 3 or 1

# Example

```
chan ch1(int), ch2(int)

process A:
  send ch1(1)
  send ch2(2)

process B:
  receive ch1(x)
  receive ch1(y)

process C:
  send ch1(3)
  send ch2(4)

process D:
  receive ch2(u)
  receive ch2(v)
```

What is received now?

- x will get 1 or 3 and y will get 3 or 1
- u will get 2 or 4 and v will get 4 or 2

# Process Interaction Patterns

# Process Interaction Patterns

# Process Interaction Patterns

- Filters: *One way*

# Process Interaction Patterns

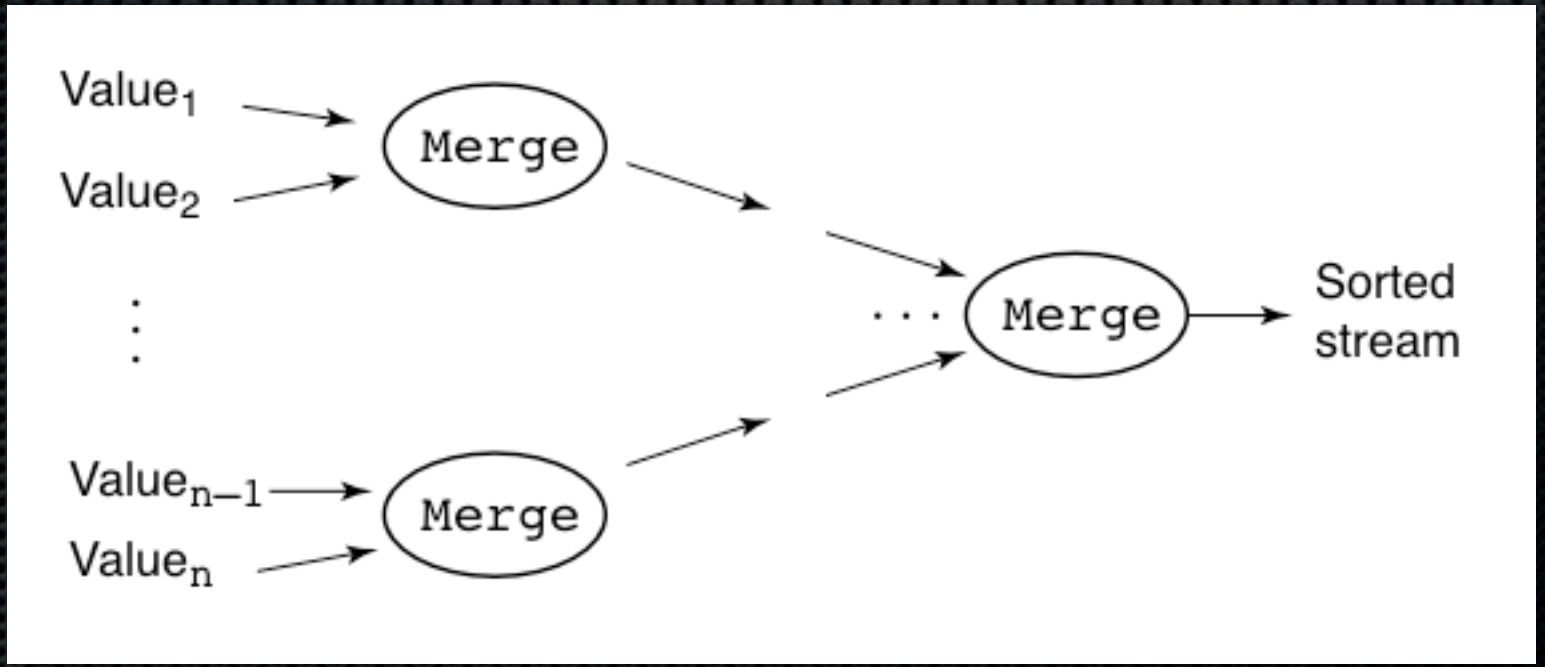
- Filters: *One way*
- Client/server: *Two way as master/slave*

# Process Interaction Patterns

- Filters: *One way*
- Client/server: *Two way as master/slave*
- Interacting peers: *Two way as equals*

# Filters

# Merge Process and Sorting Network



# Merge Process and Sorting Network

```
chan in1(int), in2(int), out(int);
process Merge {
  int v1, v2;
  receive in1(v1);  # get first two input values
  receive in2(v2);
  # send smaller value to output channel and repeat
  while (v1 != EOS and v2 != EOS) {
    if (v1 <= v2)
      { send out(v1); receive in1(v1); }
    else   # (v2 < v1)
      { send out(v2); receive in2(v2); }
  }
  # consume the rest of the non-empty input channel
  if (v1 == EOS)
    while (v2 != EOS)
      { send out(v2); receive in2(v2); }
  else   # (v2 == EOS)
    while (v1 != EOS)
      { send out(v1); receive in1(v1); }
  # append a sentinel to the output channel
  send out(EOS);
}
```

# Clients and Servers

# Clients and Server with One Operation

```
chan request(int clientID, types of input values);
chan reply[n](types of results);

process Server {
  int clientID;
  declarations of other permanent variables;
  initialization code;
  while (true) {    ## loop invariant MI
    receive request(clientID, input values);
    code from body of operation op;
    send reply[clientID](results);
}

process Client[i = 0 to n-1] {
  send request(i, value arguments);      # "call" op
  receive reply[i](result arguments);    # wait for reply
}
```

# Clients and Server with One Operation

```
chan request(int clientID, types of input values);
chan reply[n](types of results);

process Server {
  int clientID;
  declarations of other permanent variables;
  initialization code;
  while (true) {    ## loop invariant MI
    receive request(clientID, input values);
    code from body of operation op;
    send reply[clientID](results);
}

process Client[i = 0 to n-1] {
  send request(i, value arguments);      # "call" op
  receive reply[i](result arguments);    # wait for reply
}
```

*syntax for creating multiple processes*

# Clients and Server with One Operation

```
chan request(int clientID, types of input values);
chan reply[n](types of results);

process Server {
  int clientID;
  declarations of other permanent variables;
  initialization code;
  while (true) {    ## loop invariant MI
    receive request(clientID, input values);
    code from body of operation op;
    send reply[clientID](results);
}

process Client[i = 0 to n-1] {
  send request(i, value arguments);      # "call" op
  receive reply[i](result arguments);    # wait for reply
}
```

# Clients and Server with One Operation

```
chan request(int clientID, types of input values);
chan reply[n](types of results);

process Server {
  int clientID;
  declarations of other permanent variables;
  initialization code;
  while (true) {    ## loop invariant MI
    receive request(clientID, input values);
    code from body of operation op;
    send reply[clientID](results);
}

process Client[i = 0 to n-1] {
  send request(i, value arguments);      # "call" op
  receive reply[i](result arguments);    # wait for reply
}
```

*how processes know who they are*

# Actors

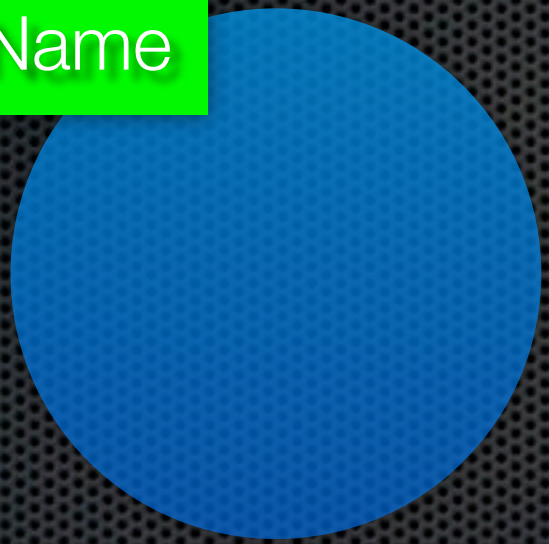
# Actors

- Actors are self-contained, interactive, autonomous components of a computing system that communicate by asynchronous message passing
- Basic primitives
  - send ( a , v ) sends message v to actor a
  - newactor ( e ) creates a new actor and returns name
  - ready ( b ) changes behavior to b and gets ready to accept another message

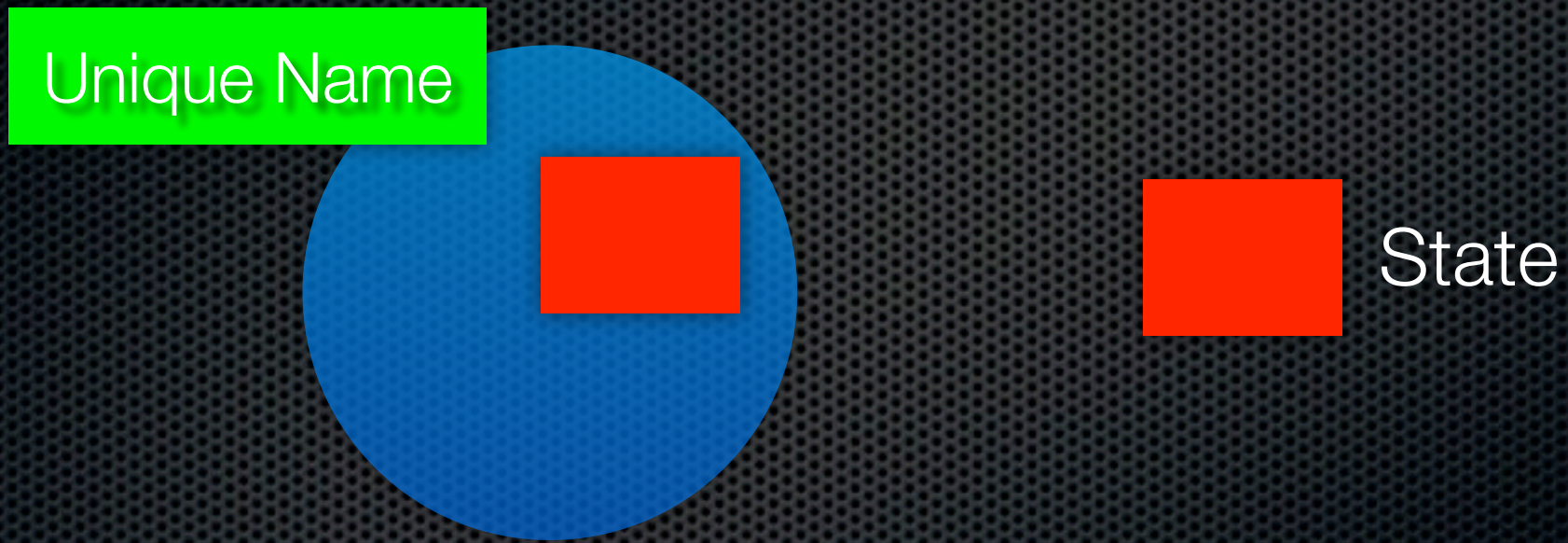
# Actors

# Actors

Unique Name

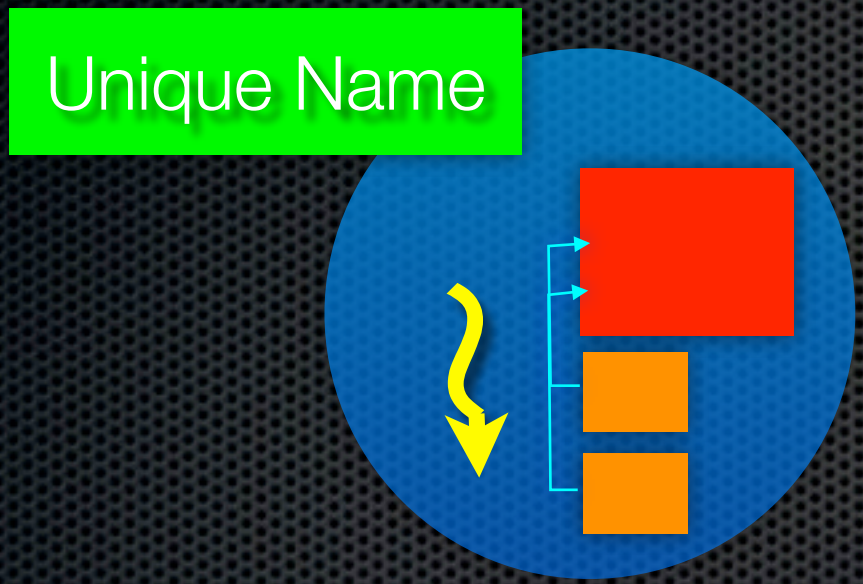


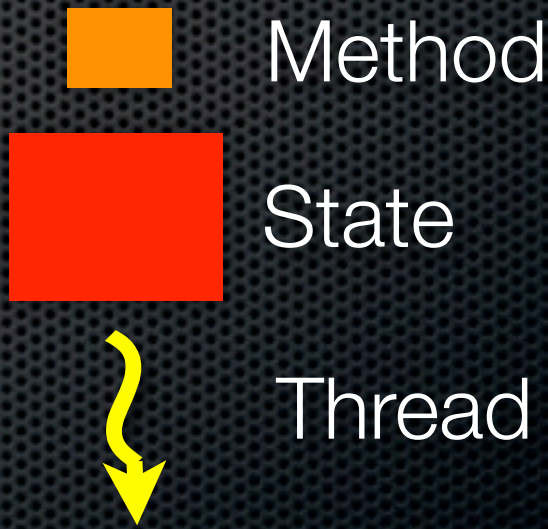
# Actors



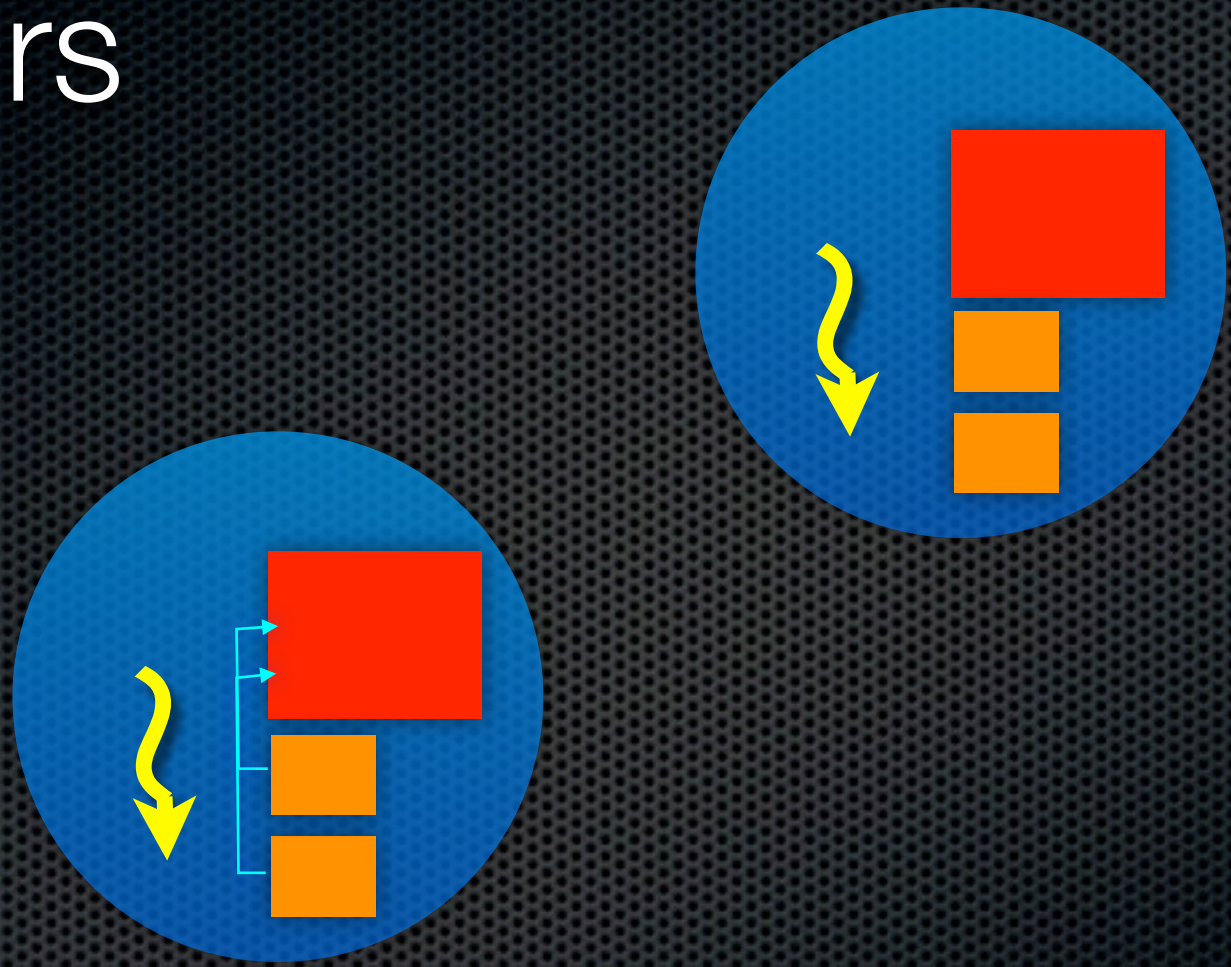
# Actors

# Actors

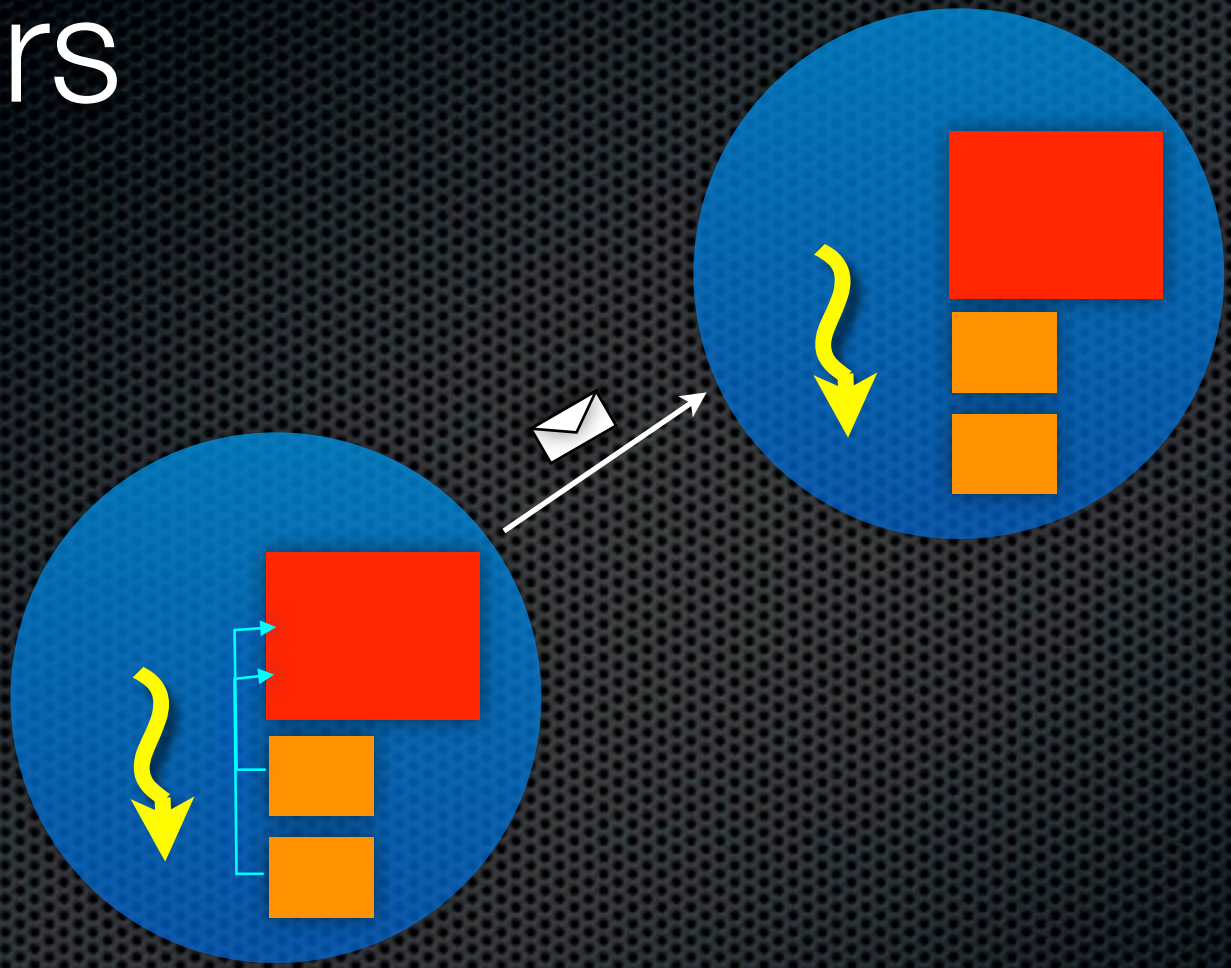


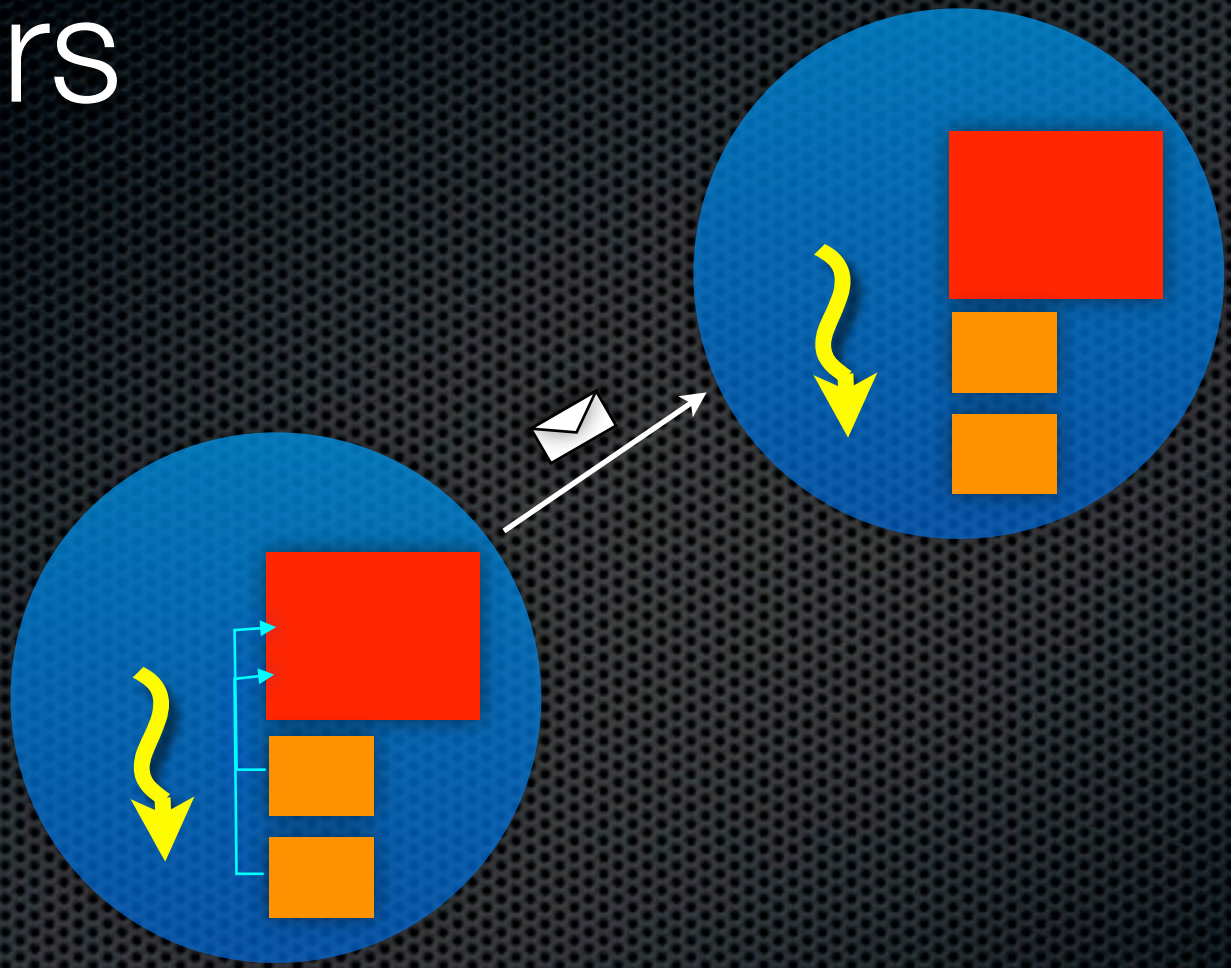
# Actors



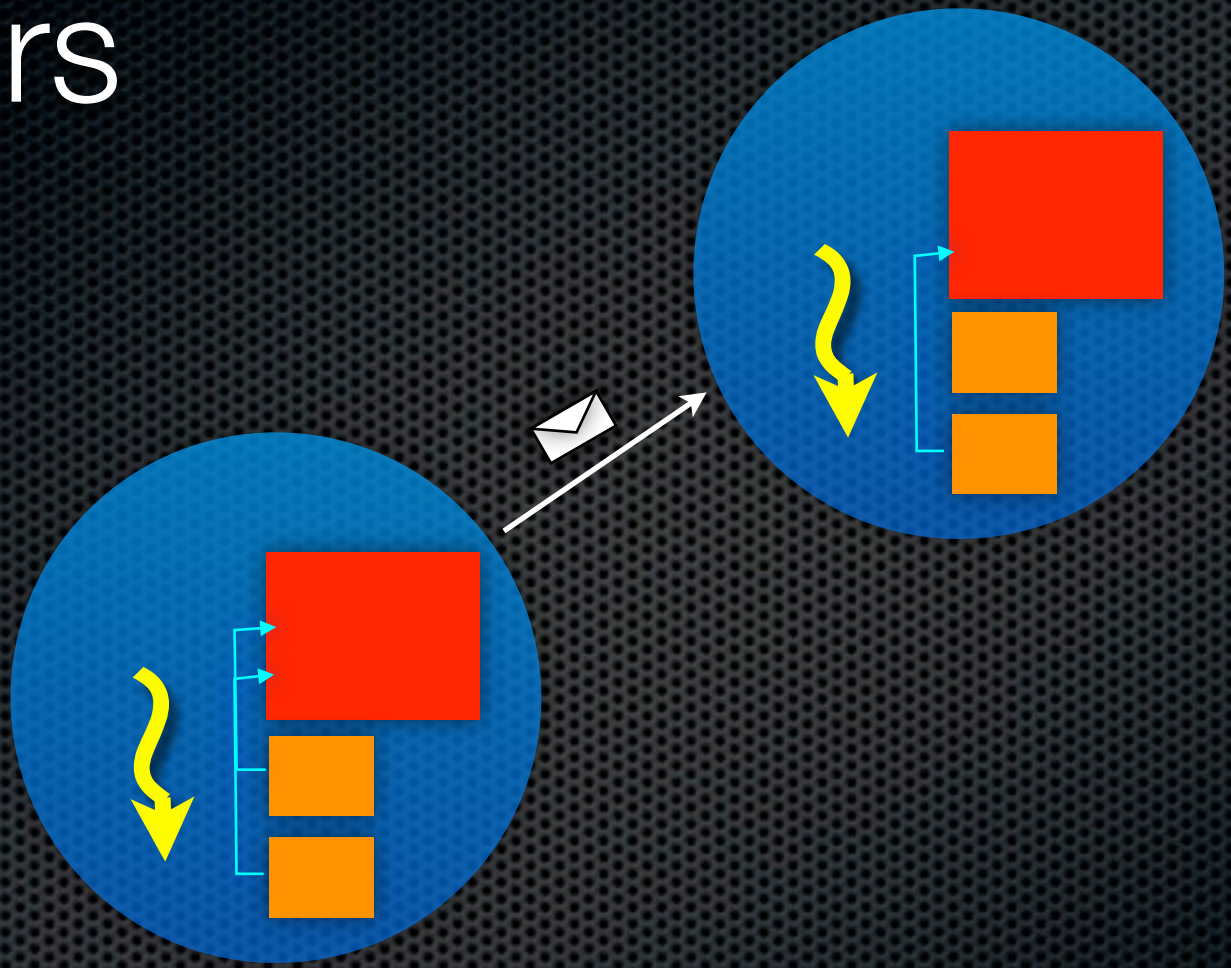
# Actors



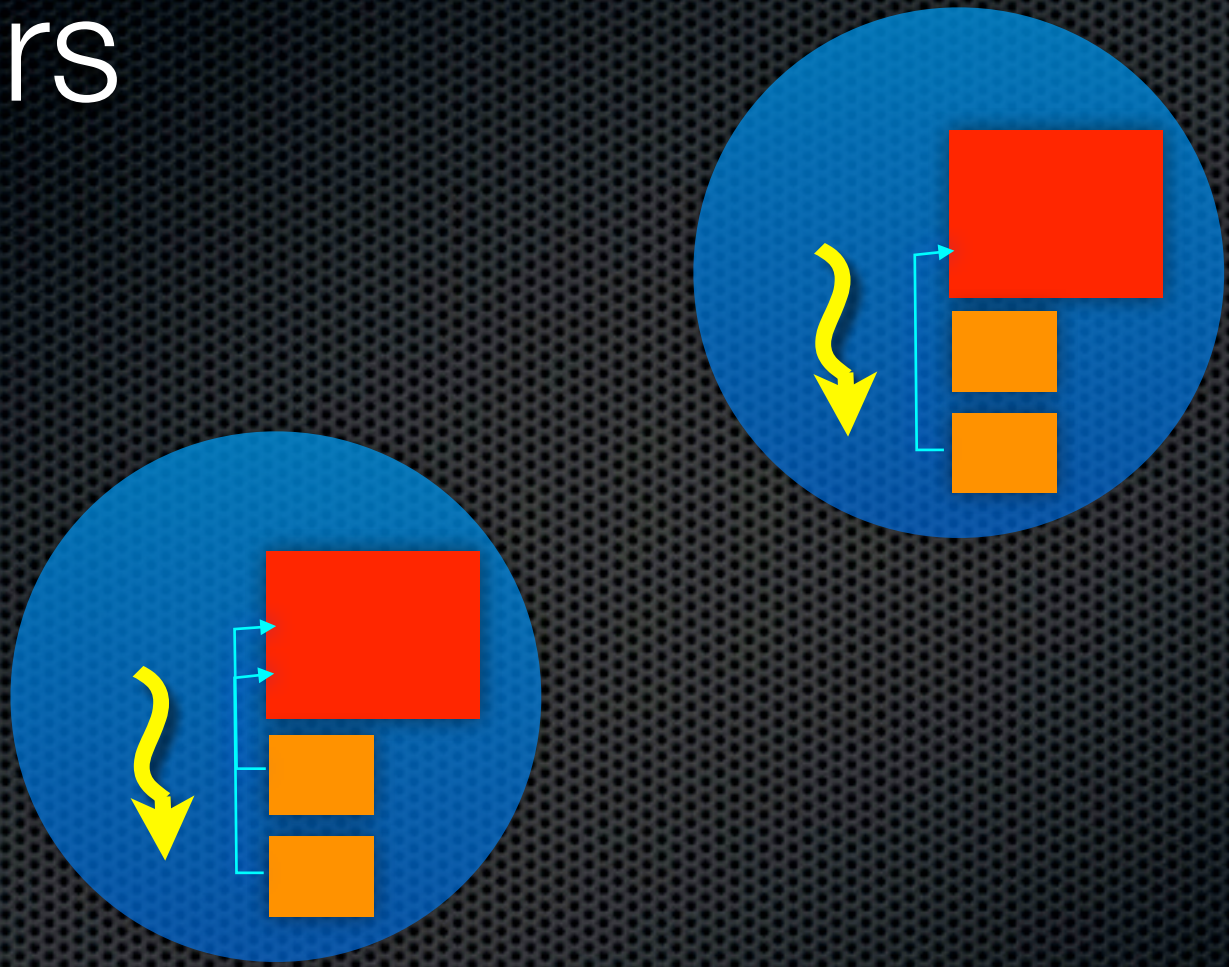
# Actors



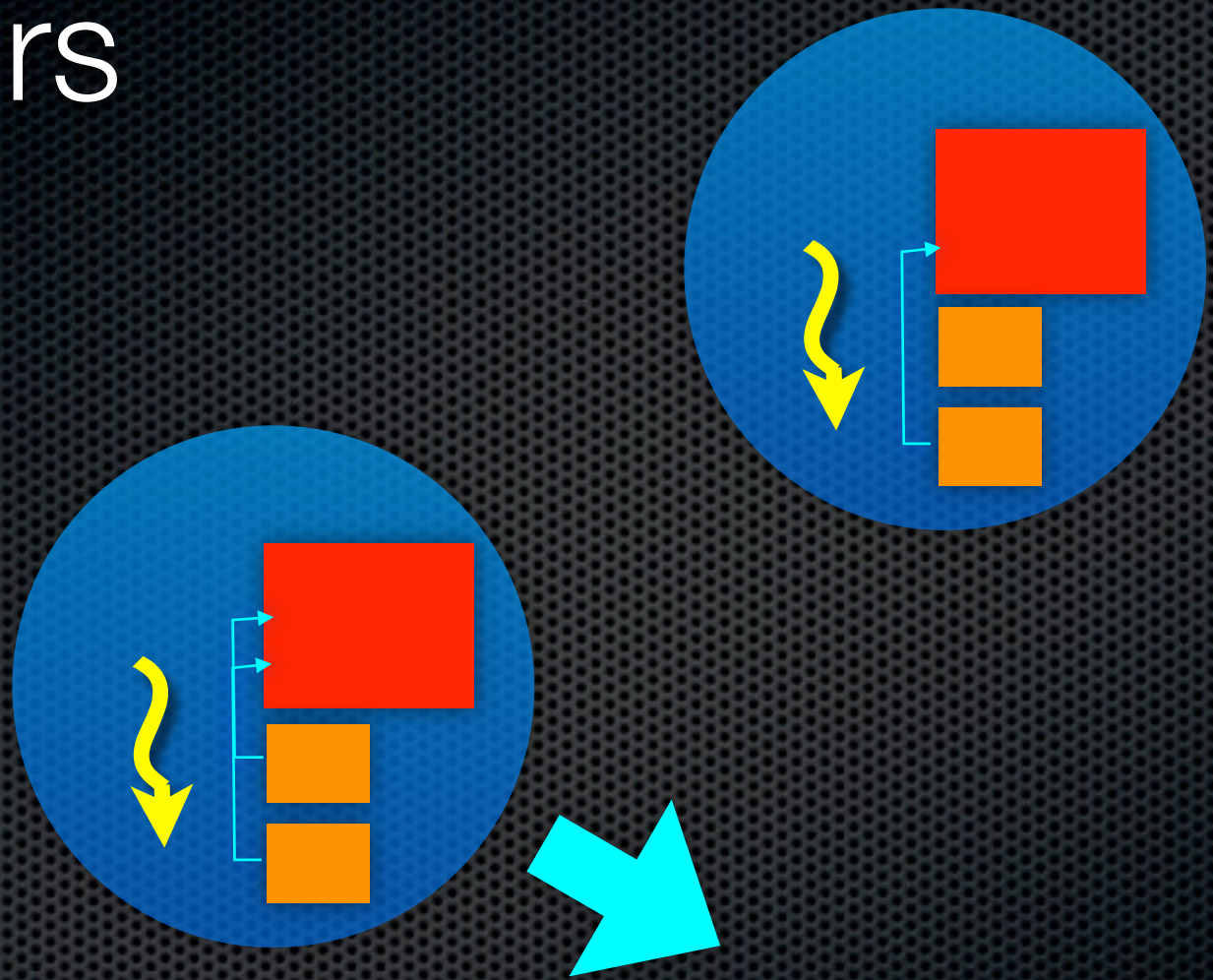
# Actors



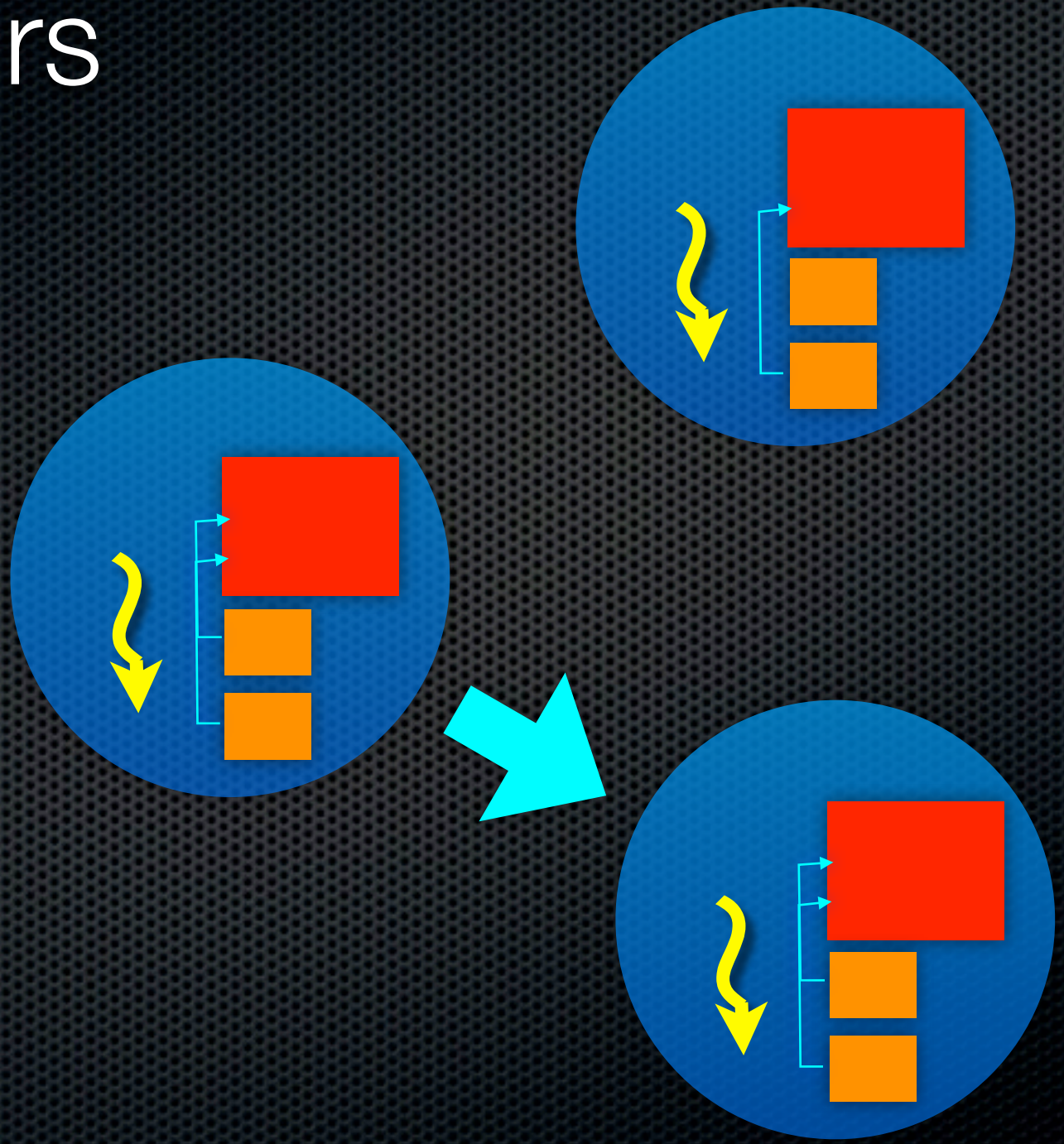
# Actors



# Actors



# Actors



# Actors

# Actors

- Actors have names

# Actors

- Actors have names
- What does that mean?

# Actors

- Actors have names
- What does that mean?
    - Each channel only has one recipient