MPI

MPI = message passing interface

- No shared memory
- More language-neutral than OpenMP
  - ○ Library (no new compiler)

    ⇒ essentially a grown-up **`bmsg.c`**
  - ○ Biased toward C and Fortran, but also implemented in other languages
- Run-time manager helps launch processes

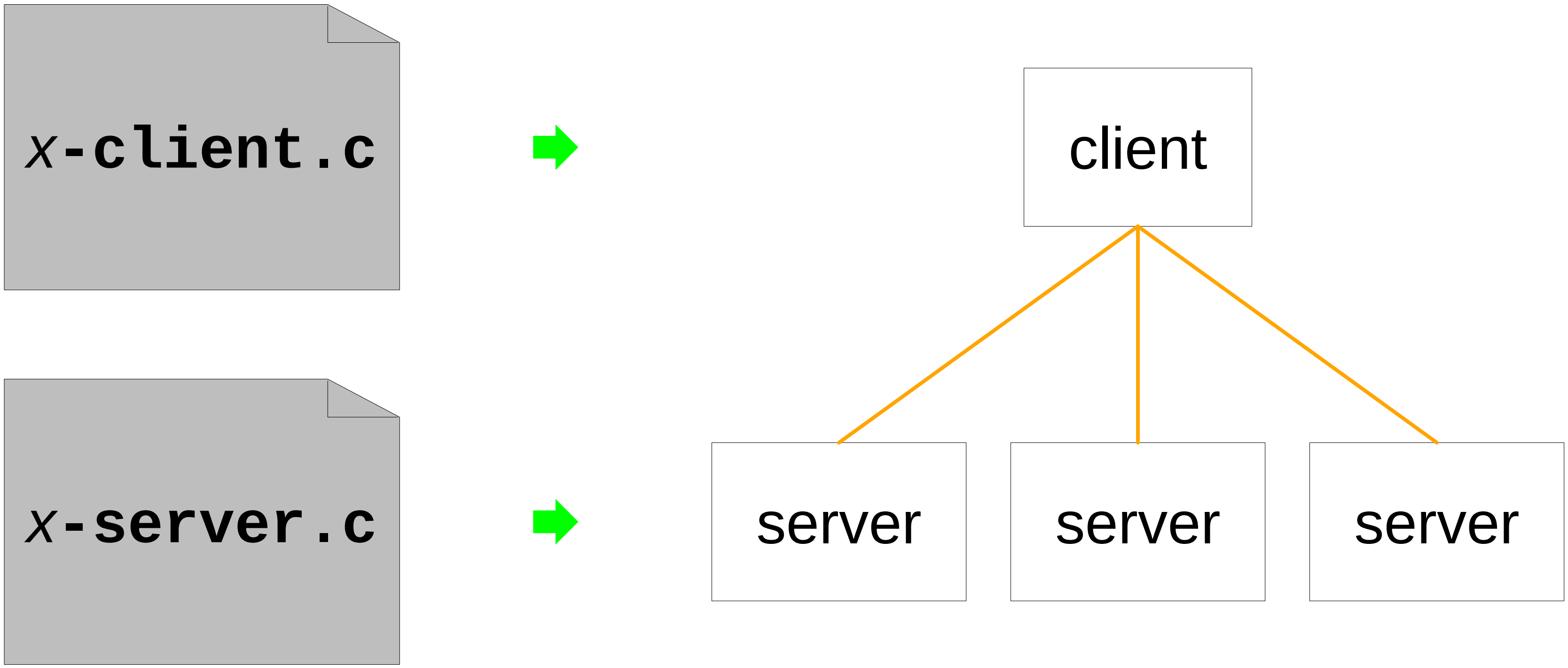
Latest version is 2.0, but 1.3 is enough for our purposes

MPI Program Model

Write one program...

- Run-time manager runs it *P* times
- Each process discovers its ***rank*** ⇒ role
- Processes coordinate through explicit messages

Old Message-Passing Architecture



x`-client.c` ➡

x`-server.c` ➡

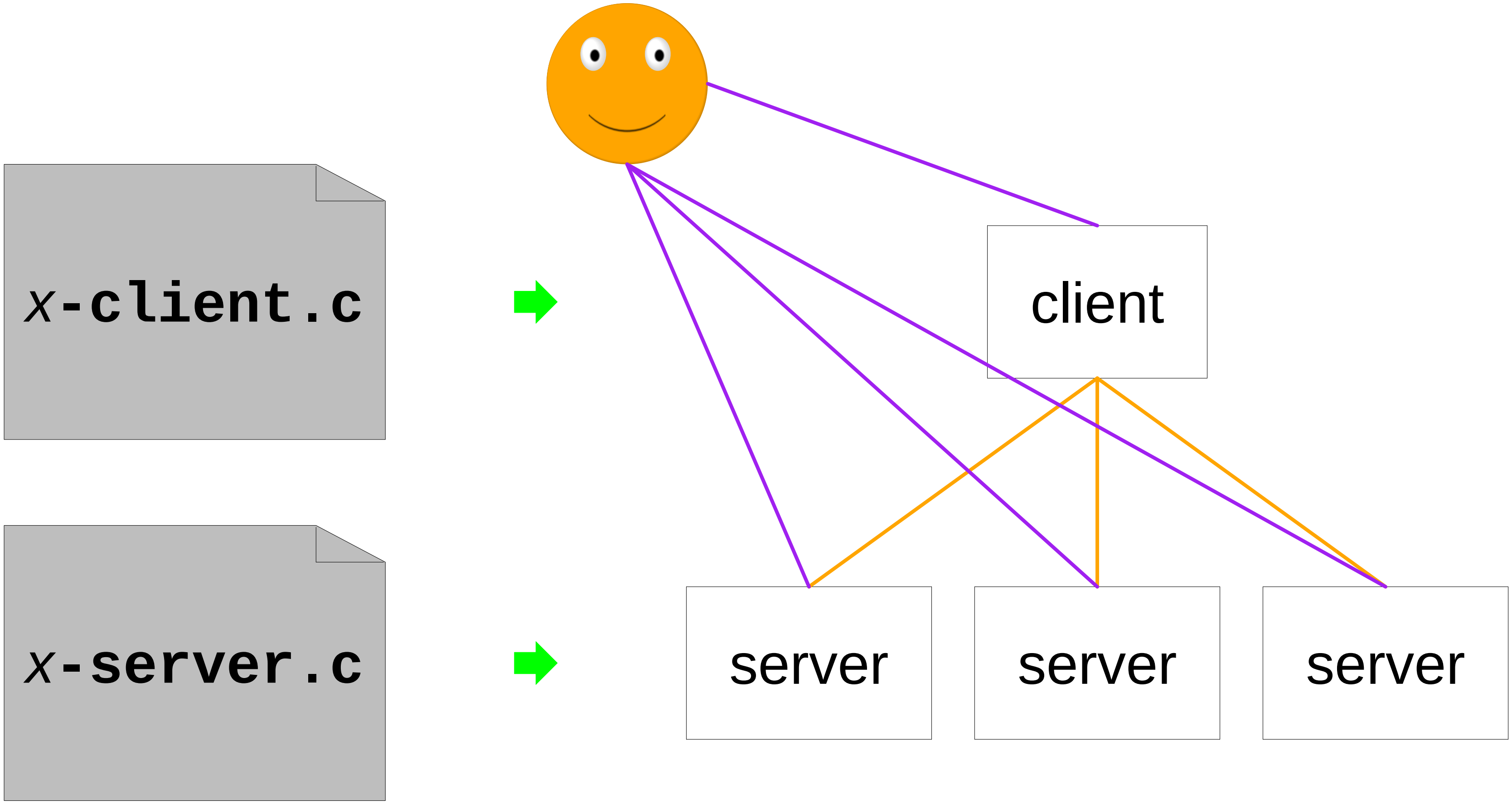
client

server

server

server

Old Message-Passing Architecture



x-**client.c**

x-**server.c**

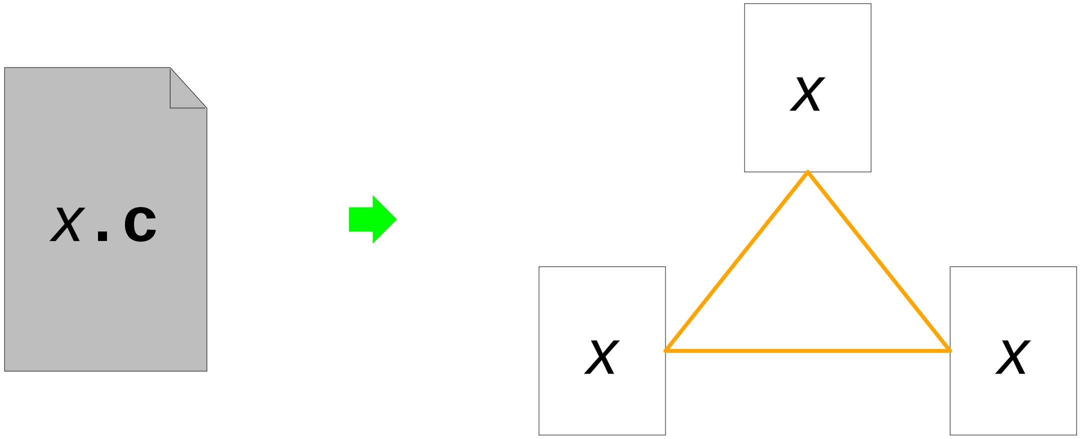
client

server

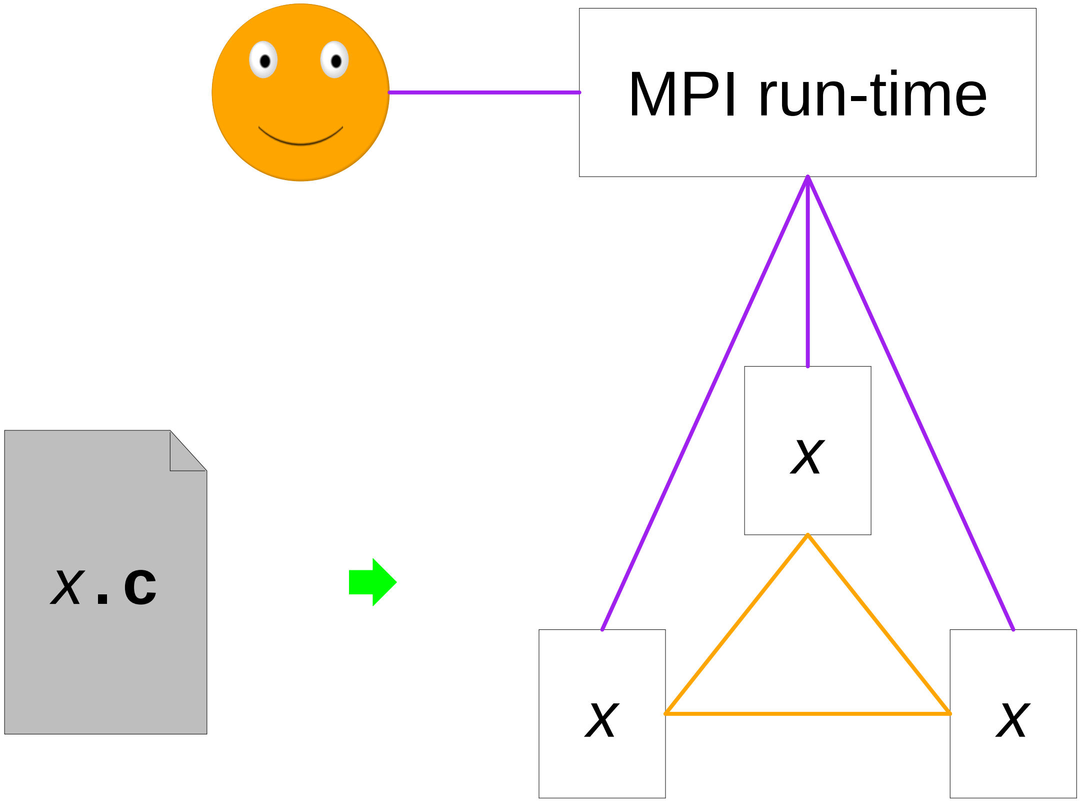
server

server

MPI Architecture



MPI Architecture



MPI run-time

x.c

x

x

x

MPI “Hello World” in C

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(processor_name, &namelen);

  printf("Process %d on %s out of %d\n", rank,
         processor_name, numprocs);

  MPI_Finalize();
}
```

MPI “Hello World” in Java

```
import mpi.*;

class HW {
  public static void main(String[] args) {
    MPI.Init(args);

    int sz = MPI.COMM_WORLD.Size();
    int me = MPI.COMM_WORLD.Rank();
    String where = MPI.Get_processor_name();

    System.out.println("Process " + me
                       + " on " + where
                       + " out of " + sz);

    MPI.Finalize();
  }
}
```

MPI Communicators

A ***communicator*** represents a set of cooperating processes

Just use **`COMM_WORLD`**, which is initialized by **`Init`**

MPI Basic Messages

```
int me = MPI.COMM_WORLD.Rank();
int size = 1;
int array[] = new int[size];

if (me == 0) {
  array[0] = 42;
  MPI.COMM_WORLD.Send(array, 0, size, MPI.INT, 1, 8);
  System.out.println("sent " + array[0]);
} else {
  MPI.COMM_WORLD.Recv(array, 0, size, MPI.INT, 0, 8);
  System.out.println("got " + array[0]);
}
```

Sending a Message

To send:

- Specificy data as array, size, and type
- Specify target process (by its rank)
- Specify a ***tag***
  - A kind of mailbox id within the target process
  - Meaning of a tag is completely up to programmer

Receiving a Message

To receive:

- Specificy data area as array, size, and type
- Specify source process (by its rank) or use **`ANY_SOURCE`**
- Specify a tag or use **`ANY_TAG`**

MPI Send Modes

- ***standard*** — message is conceptually sent after **Send** returns; may or may not block until received
- ***buffered*** — like standard, but **Bsend** never waits for receive
- ***synchronous*** — like standard, but **Ssend** always waits for receive
- ***ready*** — **Rsend** assumes(!) that receive is currently waiting

The same **Recv** is used for all send modes

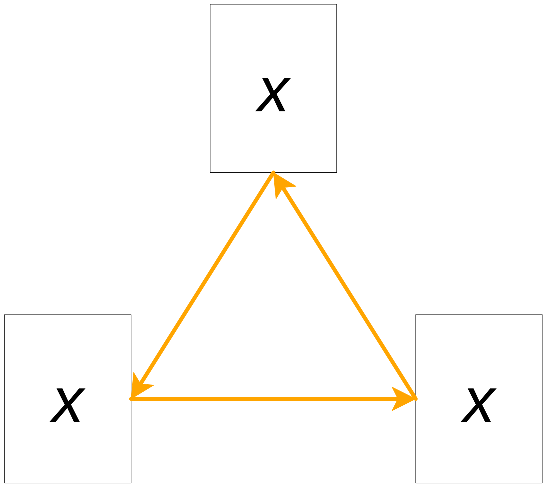
MPI Blocking

- The **`Send`**, **`Bsend`**, **`Ssend`**, **`Rsend`**, and **`Recv`** operations are all ***blocking***
  - Send or receive complete on return, buffers can be re-used
- The **`Isend`**, **`Ibsend`**, **`Issend`**, **`Irsend`**, and **`Irecv`** operations are all ***non-blocking***
  - Check back for send or receive completion: **`Wait`**, **`Test`**, **`WaitAny`**, ...
  - Buffers cannot be re-used until completion

Threads could express non-blocking with blocking, but only if you have threads and if the MPI library is thread-safe

Send plus Receive

Suppose that you need to shift data around:



If everyone sends (synchronously, non-blocking) first, then everyone is stuck

Use **SendRecv** and let the library handle ordering and efficiency