mpipipe

A Parallelized Unix Pipe for Serial Filter Programs

Nathan Weeks <weeks@iastate.edu>

Pipes

- Task: align DNA sequences in two files to the soybean genome. Keep alignments >100 in length.
- Example without pipes:

```
$ ls

DNA1.fasta.gz DNA2.fasta.gz

$ gunzip DNA?.fasta.gz

$ blastn -infile DNA1.fasta -db Glycine_max > temp1.txt

$ blastn -infile DNA2.fasta -db Glycine_max > temp2.txt

$ awk '$5 - $4 > 100' temp1.txt temp2.txt > output.txt
```

- Note the use of temporary files

Pipes

- Pipe: a one-way communication channel between two processes on the same host
- Previous example with pipes

```
$ gzcat DNA?.fasta.gz |

    blastn -db Glycine_max |

        awk '$5 - $4 > 100' > finaloutput.txt
```

- Note the lack of temporary intermediate files

Problem

- Task: parallelize the previous example to run multiple simultanous instances of blastn on 32 nodes
- Data distribution
  - Typical solution: split the two files into 32 files (and hope you don't exceed your quota or available disk space) and submit 32 blastn jobs
- Load Balancing
  - What if the time to process one of the 32 input files is much greater than the others (e.g., slow cluster node, longer sequences, or repetitive sequence?)

Solution

```
gzcat DNA?.fasta.gz |
    mpirun -np 33 \
    mpipipe -n 2000 'blastn -db Glycine_max' |
        awk '$5 - $4 > 100' > output.txt
```

Solution

```
gzcat DNA?.fasta.gz |
    mpirun -np 33 \
    mpipipe -n 2000 'blastn -db Glycine_max' |
        awk '$5 - $4 > 100' > output.txt
```

Uncompress & concatenate input files

Solution

```
gzcat DNA?.fasta.gz |
    mpirun -np 33 \
    mpipipe -n 2000 'blastn -db Glycine_max' |
        awk '$5 - $4 > 100' > output.txt
```

(Parallelization) Run 32 worker processes, each of which runs the user-specified command , plus + 1 manager process that sends input to the workers & collects output from the workers

Solution

```
gzcat DNA?.fasta.gz |
    mpirun -np 33 \
    mpipipe -n 2000 'blastn -db Glycine_max' |
        awk '$5 - $4 > 100' > output.txt
```

(Data distribution & load balancing) Send 2000 “records” at a time to each worker process, collecting output from the worker process before sending it another 2000.

Solution

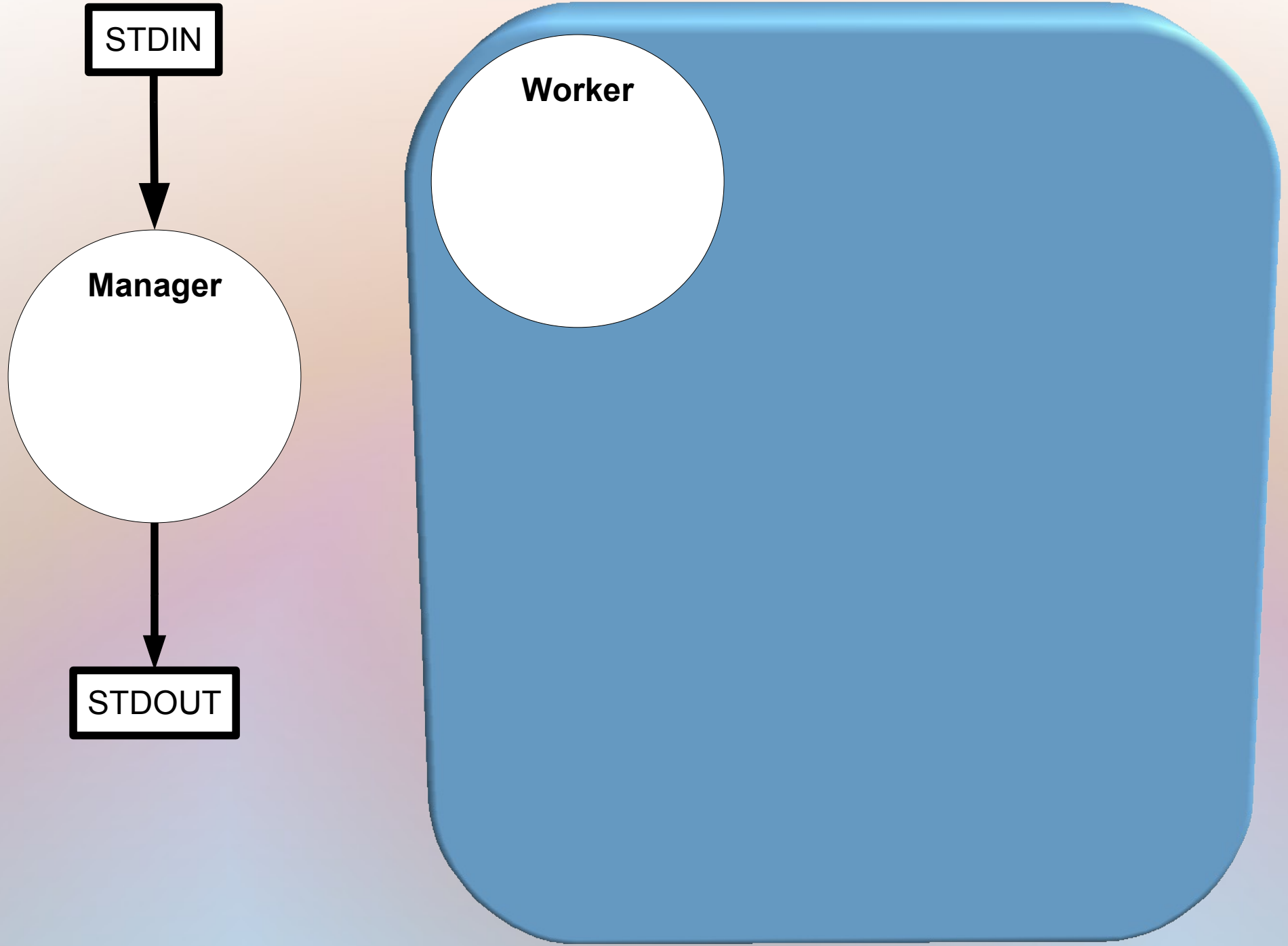
```
gzcat DNA?.fasta.gz |
    mpirun -np 33 \
    mpipipe -n 2000 'blastn -db Glycine_max' |
        awk '$5 - $4 > 100' > output.txt
```

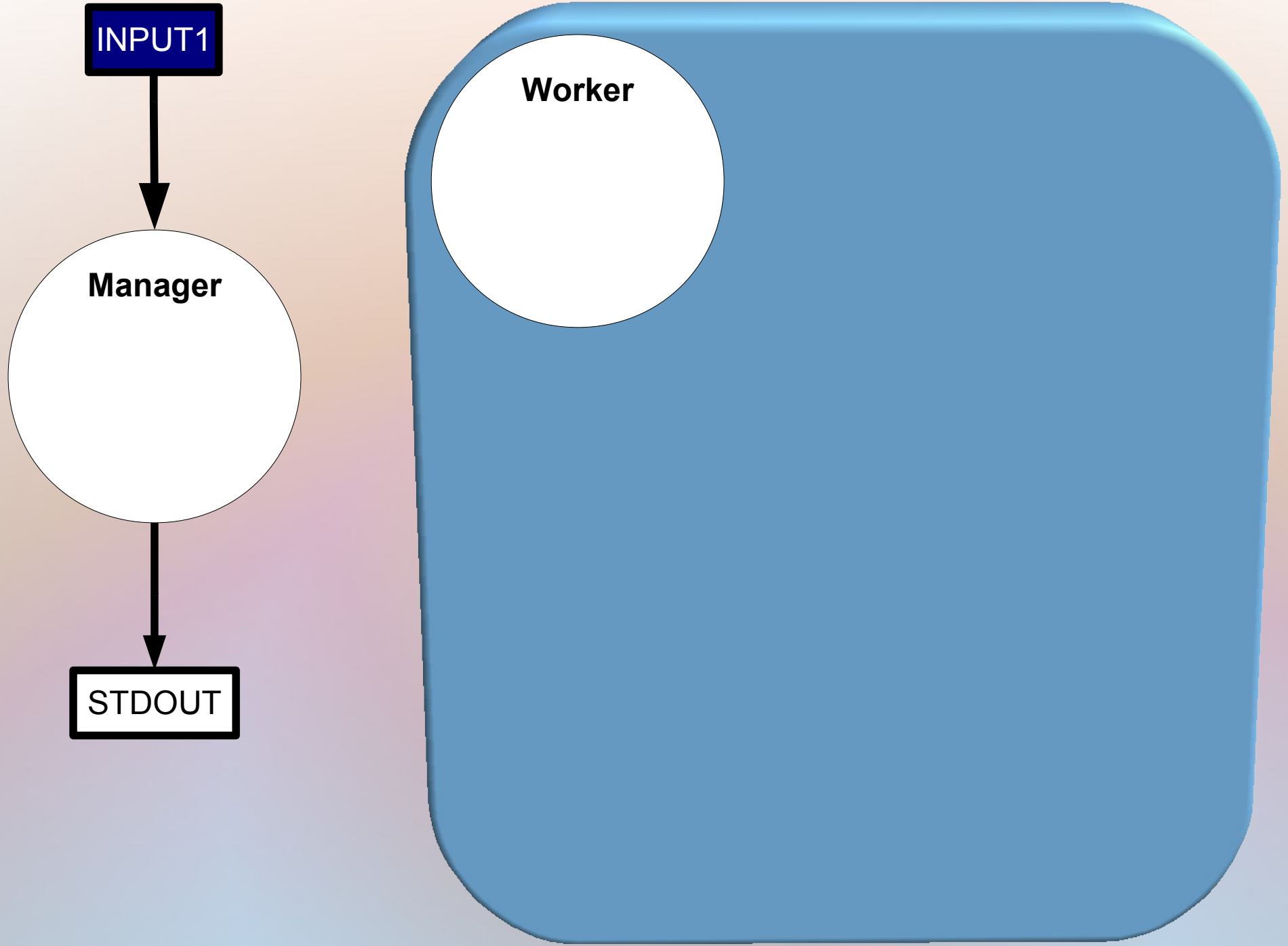
Output is printed (non-interleaved) to standard output; in this case, .

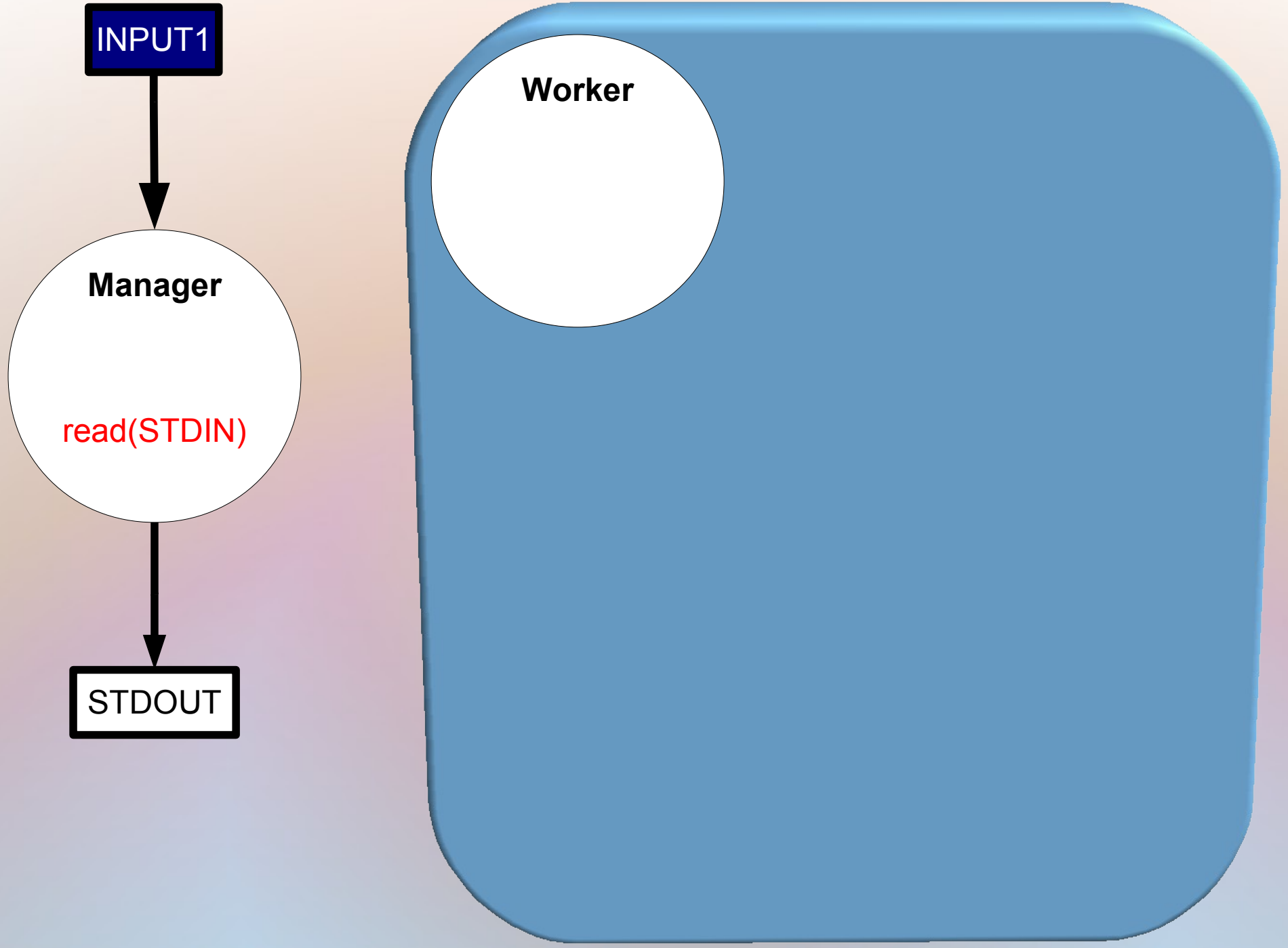
fork()

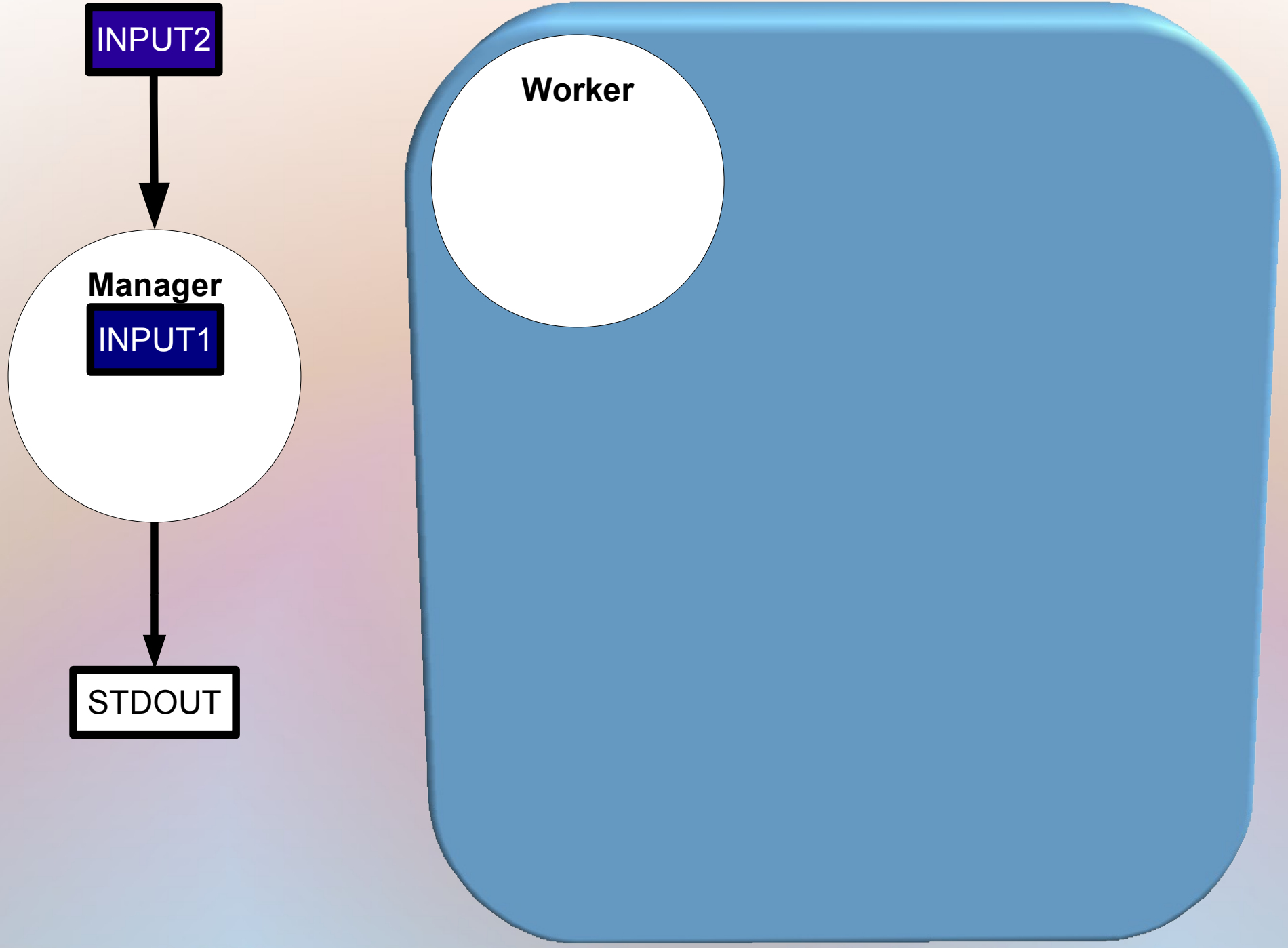
- Creates a new (“child”) process
  - clone of the parent (same program, same state)
  - has its own copy of all memory/variables in program
  - begins execution at the call to fork()

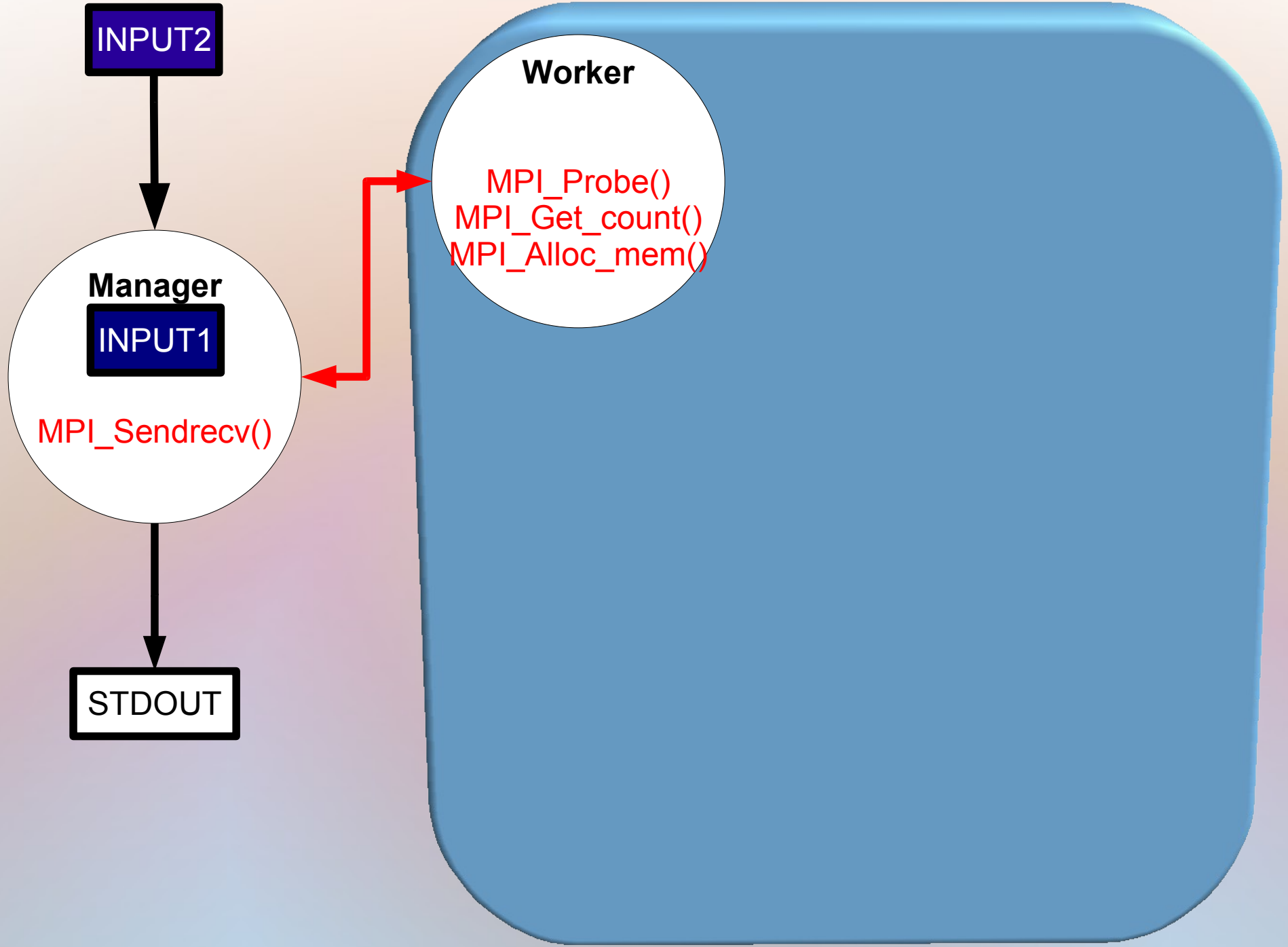
```
if (fork() == 0) {
    // child process executes this
} else {
    // parent process executes this
}
```

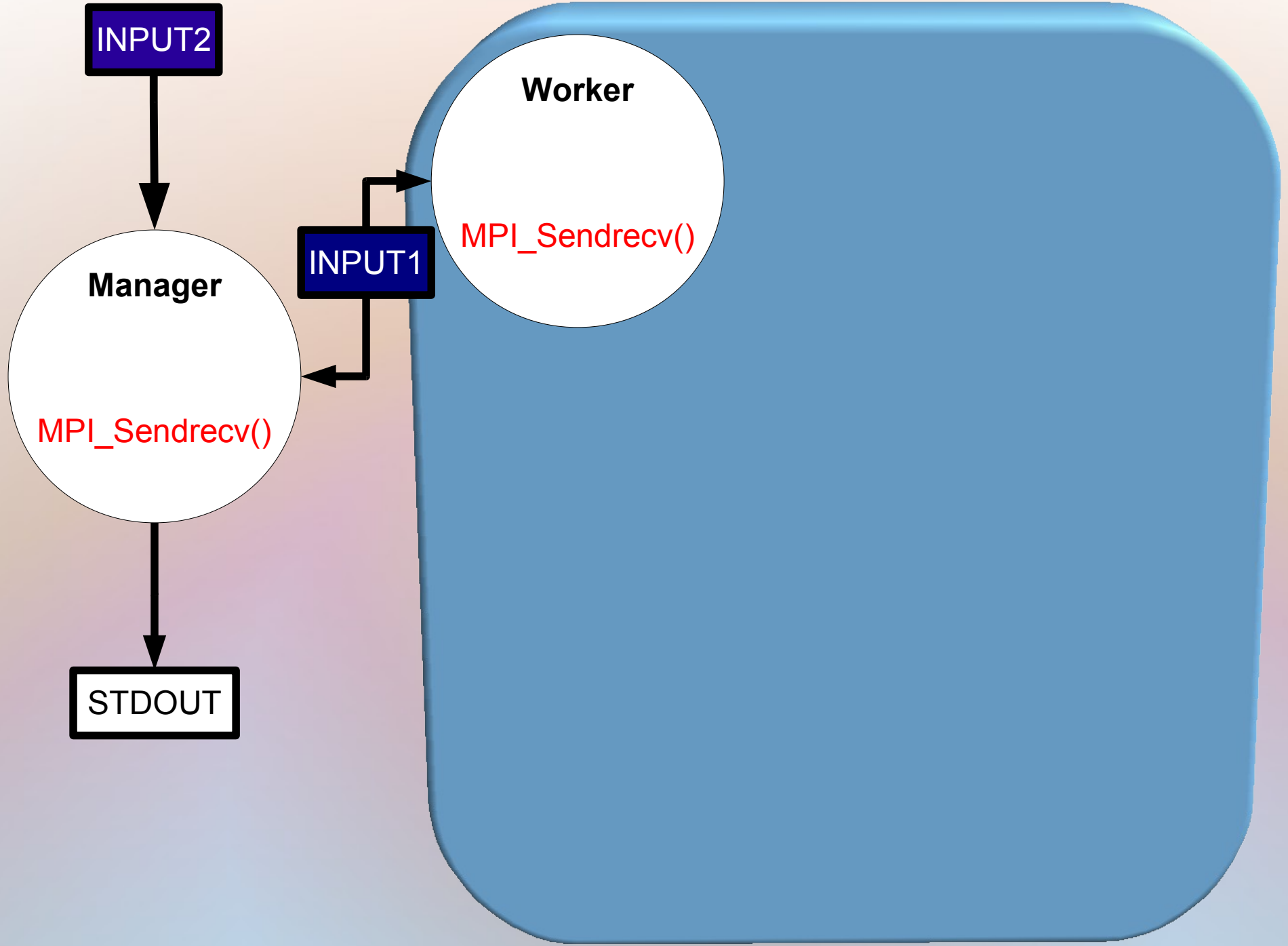

STDIN
Manager
STDOUT
Worker

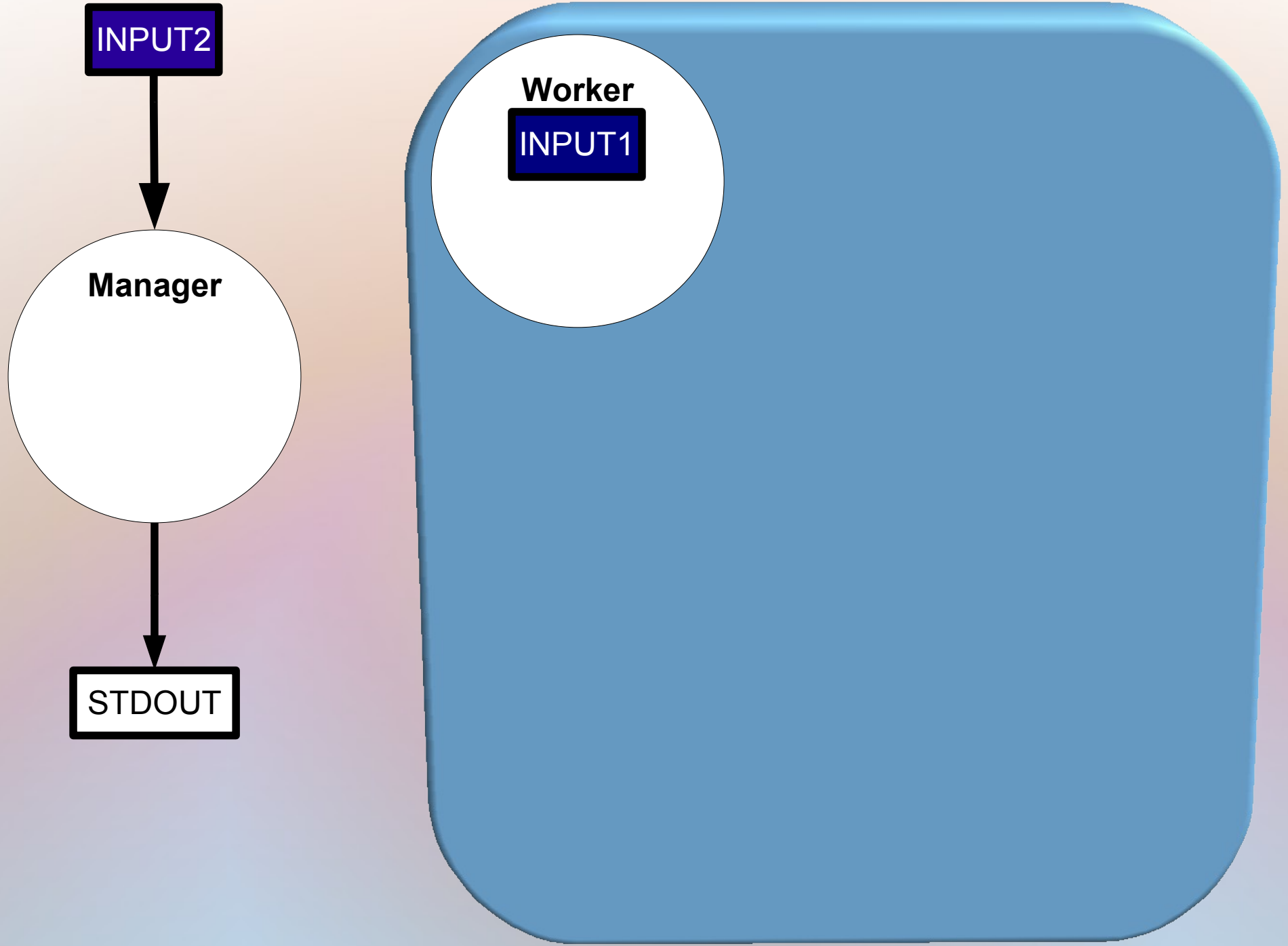

INPUT1
Manager
STDOUT
Worker

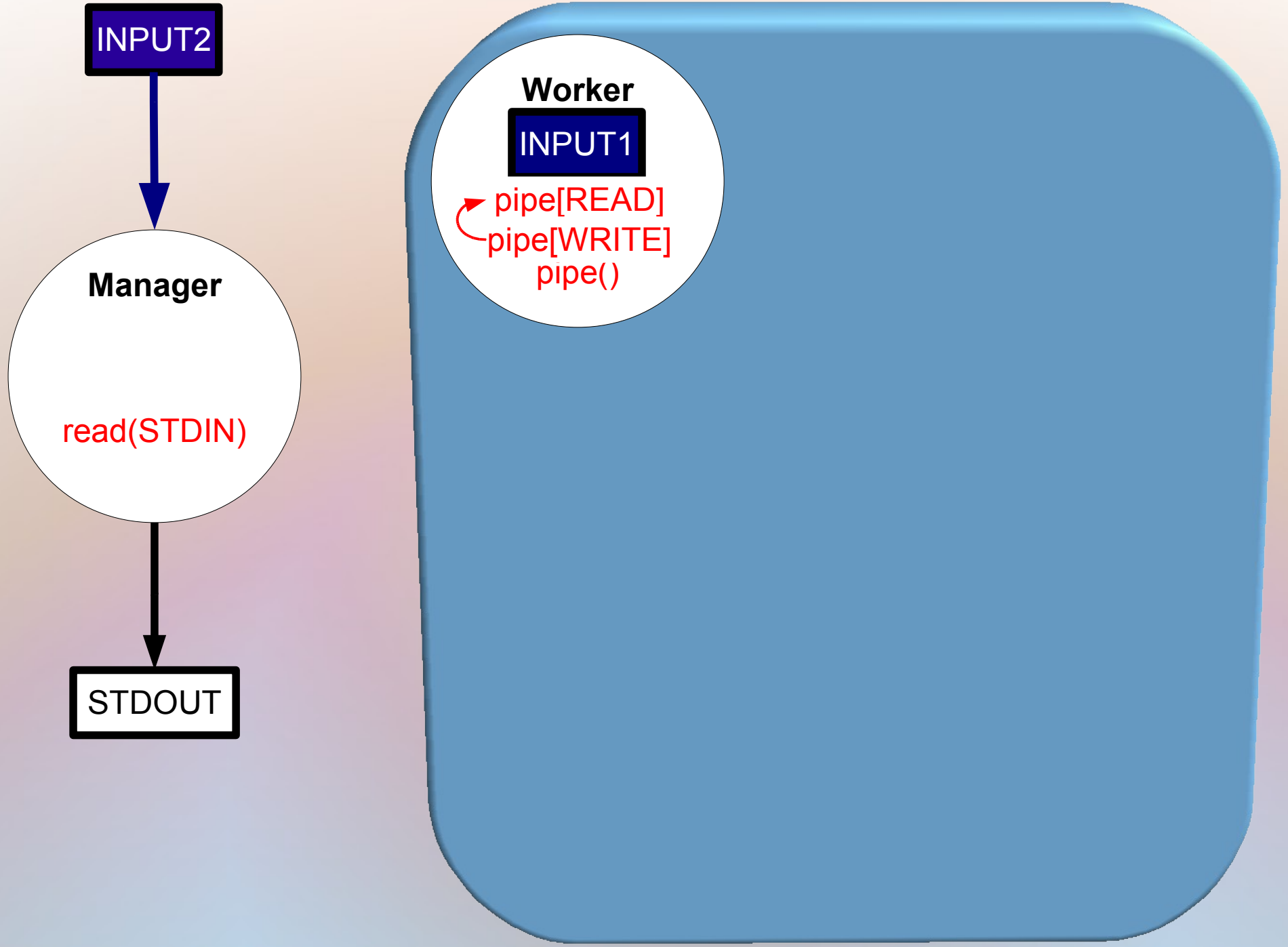

INPUT1
Manager
read(STDIN)
STDOUT
Worker

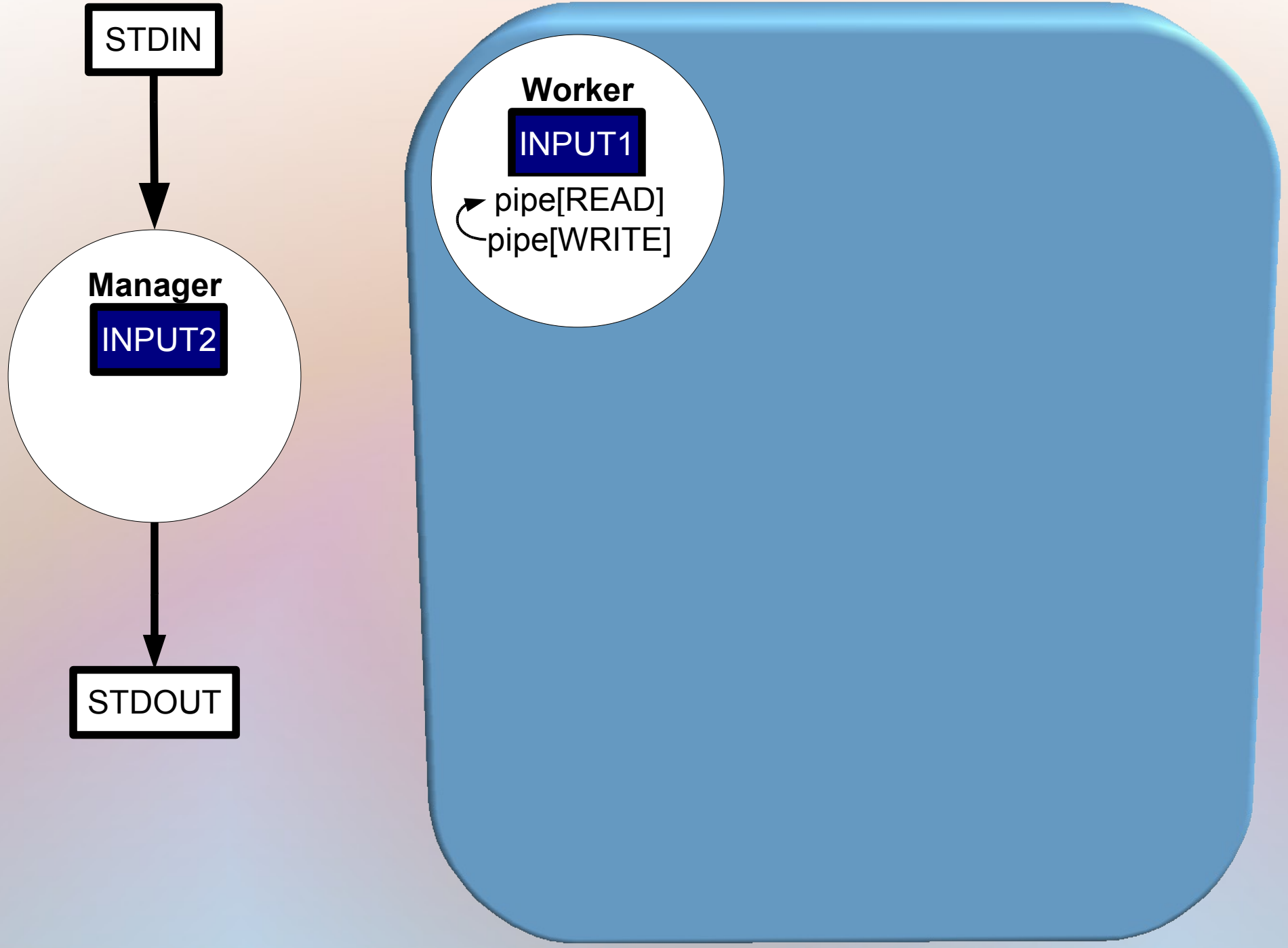

INPUT2
Manager
INPUT1
STDOUT
Worker

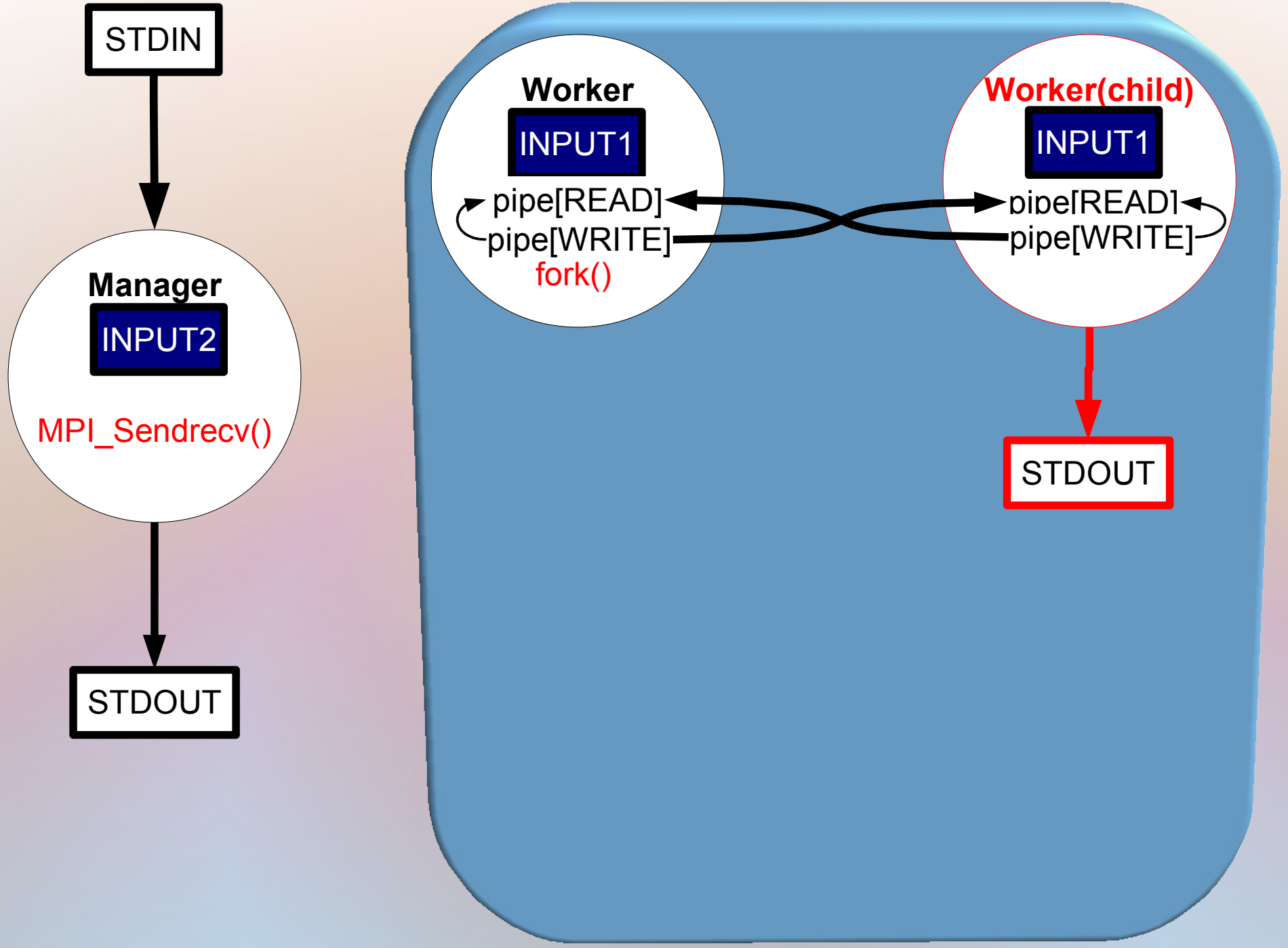

INPUT2
Worker
MPI_Probe()
MPI_Get_count()
MPI_Alloc_mem()
Manager
INPUT1
MPI_Sendrecv()
STDOUT

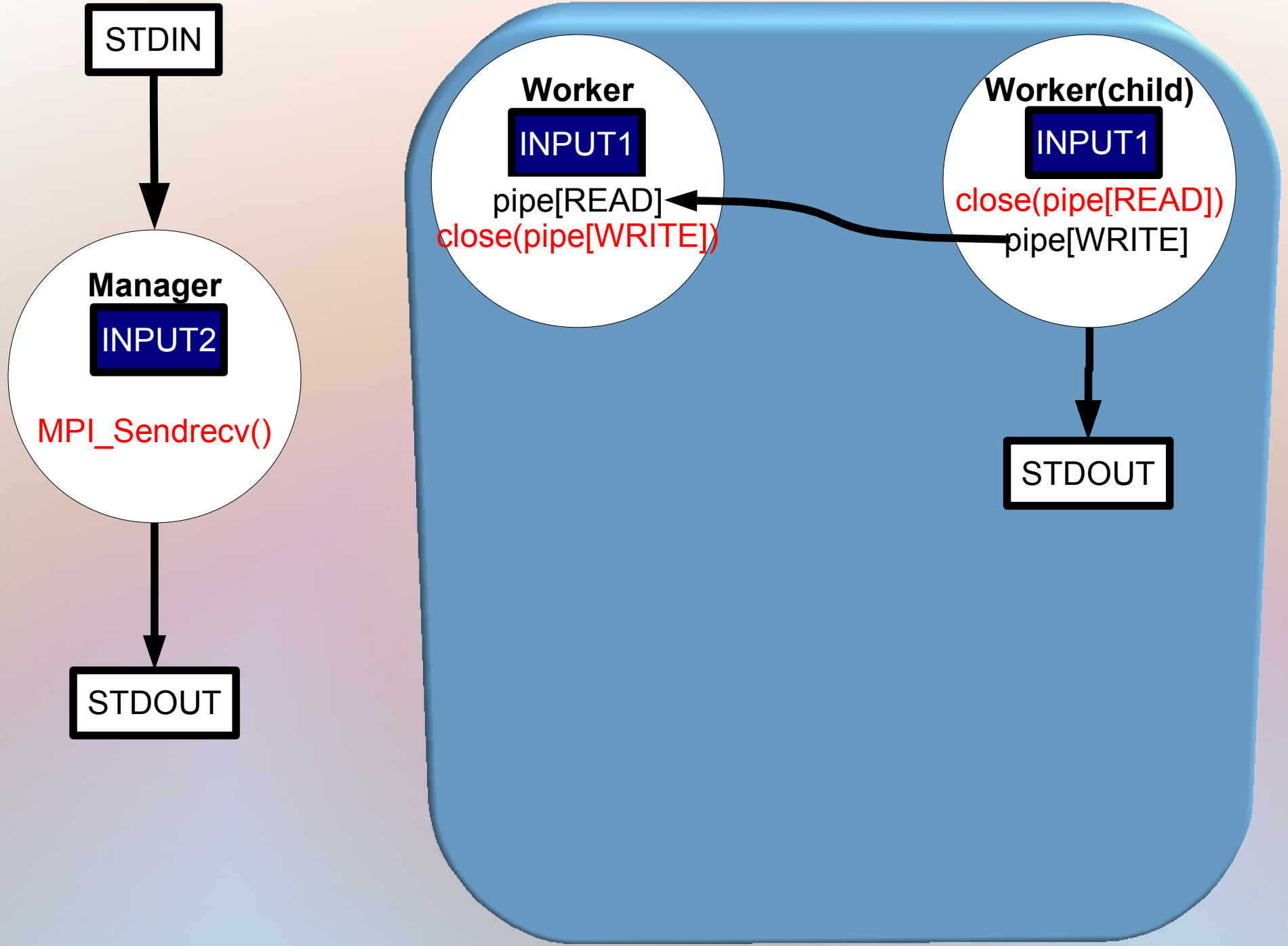

INPUT2
Worker
MPI_Sendrecv()
INPUT1
Manager
MPI_Sendrecv()
STDOUT

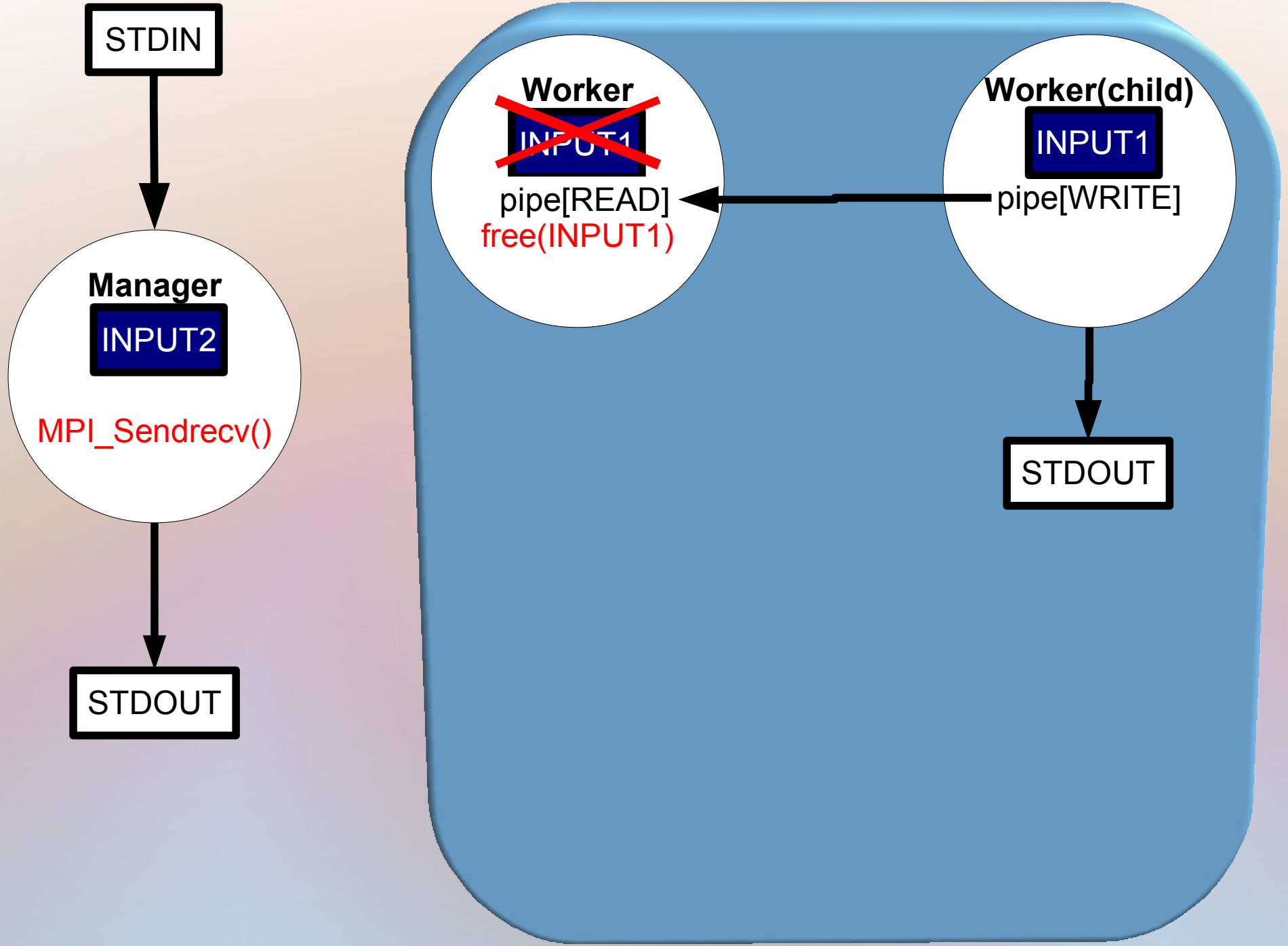

INPUT2
Manager
STDOUT
Worker
INPUT1

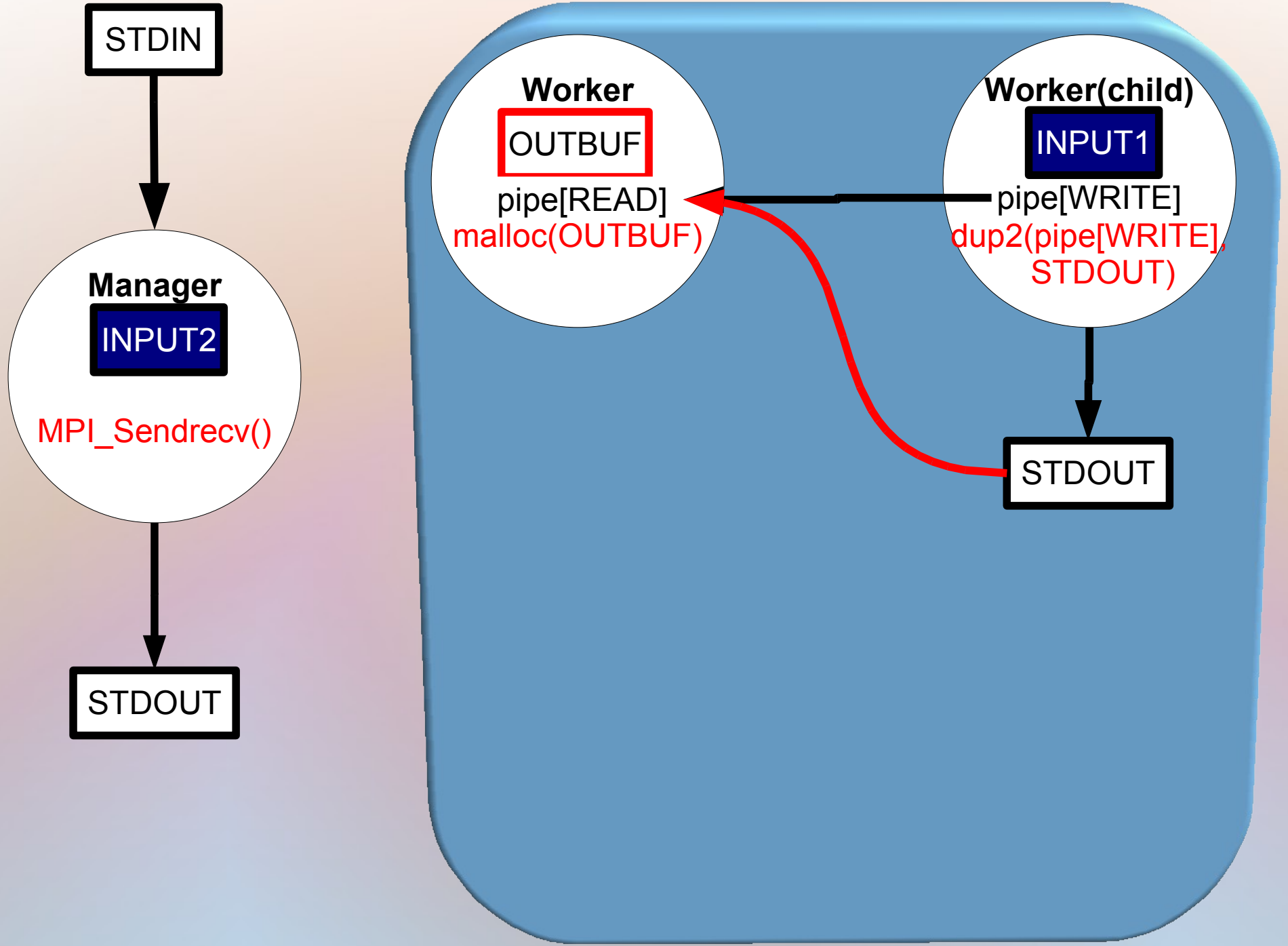

INPUT2
Manager
read(STDIN)
STDOUT
Worker
INPUT1
pipe[READ]
pipe[WRITE]
pipe()

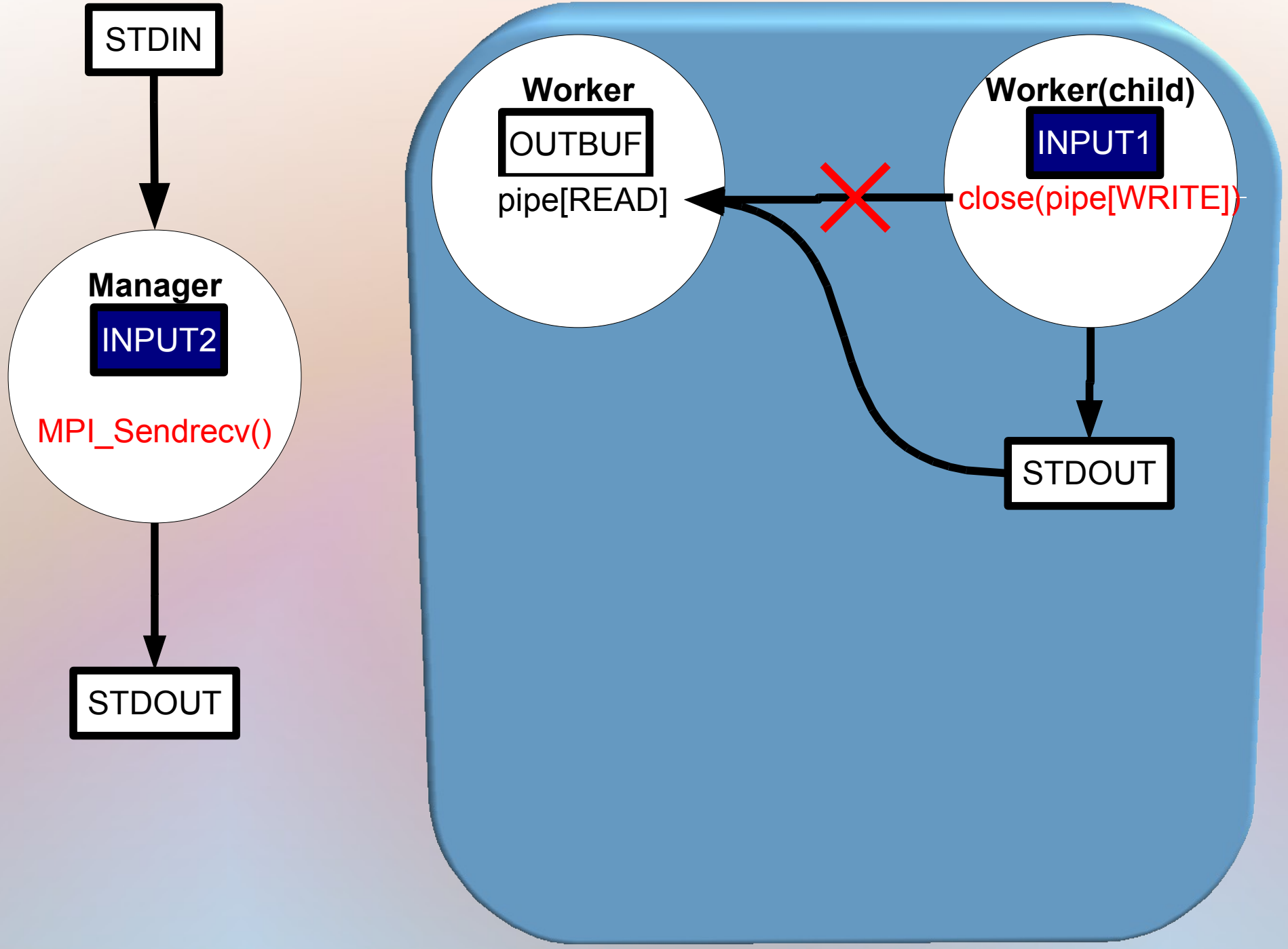

STDIN
Manager
INPUT2
STDOUT
Worker
INPUT1
pipe[READ]
pipe[WRITE]

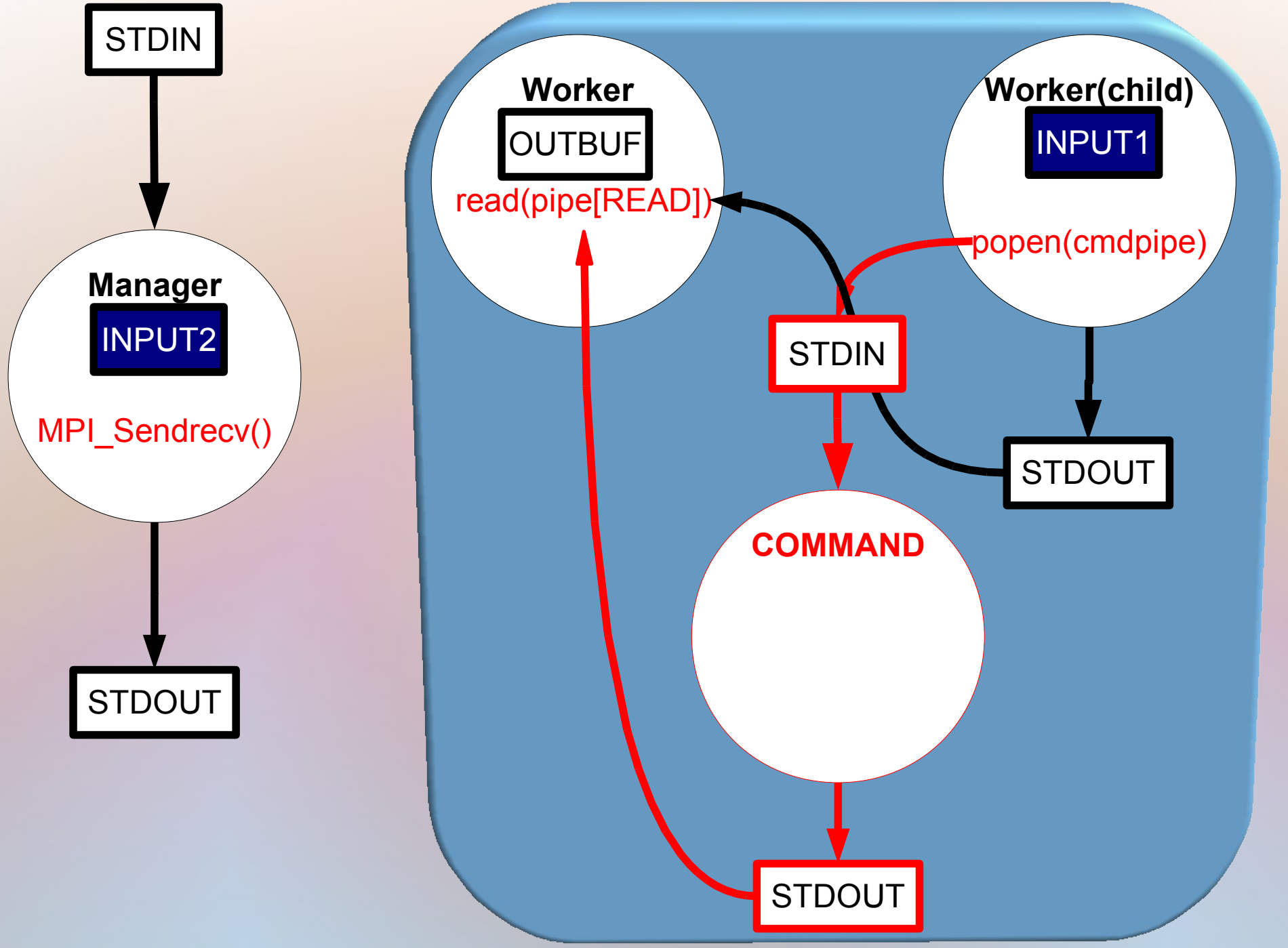

STDIN
Manager
INPUT2
MPI_Sendrecv()
STDOUT
Worker
INPUT1
pipe[READ]
pipe[WRITE]
fork()
Worker(child)
INPUT1
pipe[READ]
pipe[WRITE]
STDOUT

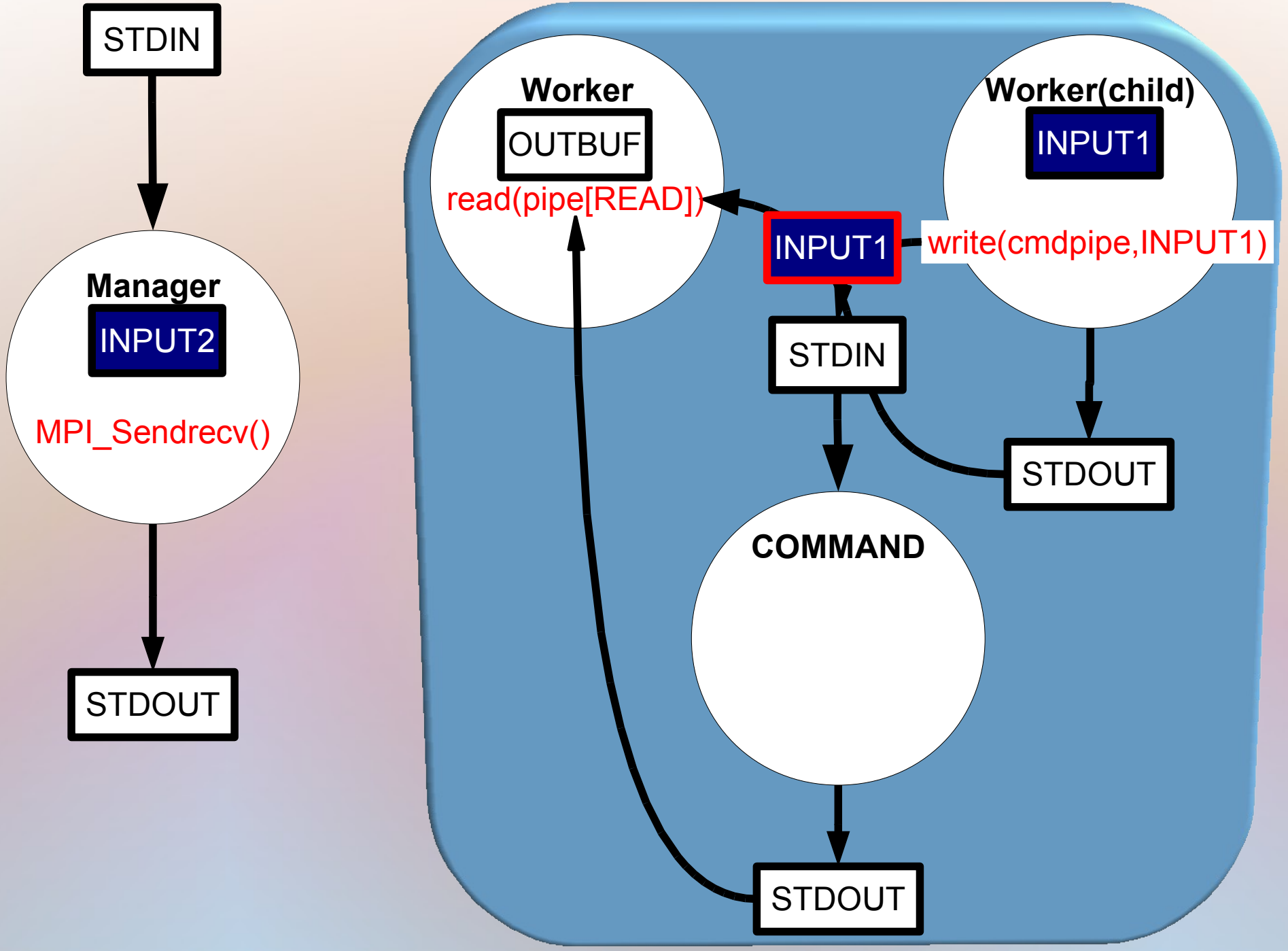

STDIN
Manager
INPUT2
MPI_Sendrecv()
STDOUT
Worker
INPUT1
pipe[READ]
close(pipe[WRITE])
Worker(child)
INPUT1
close(pipe[READ])
pipe[WRITE]
STDOUT

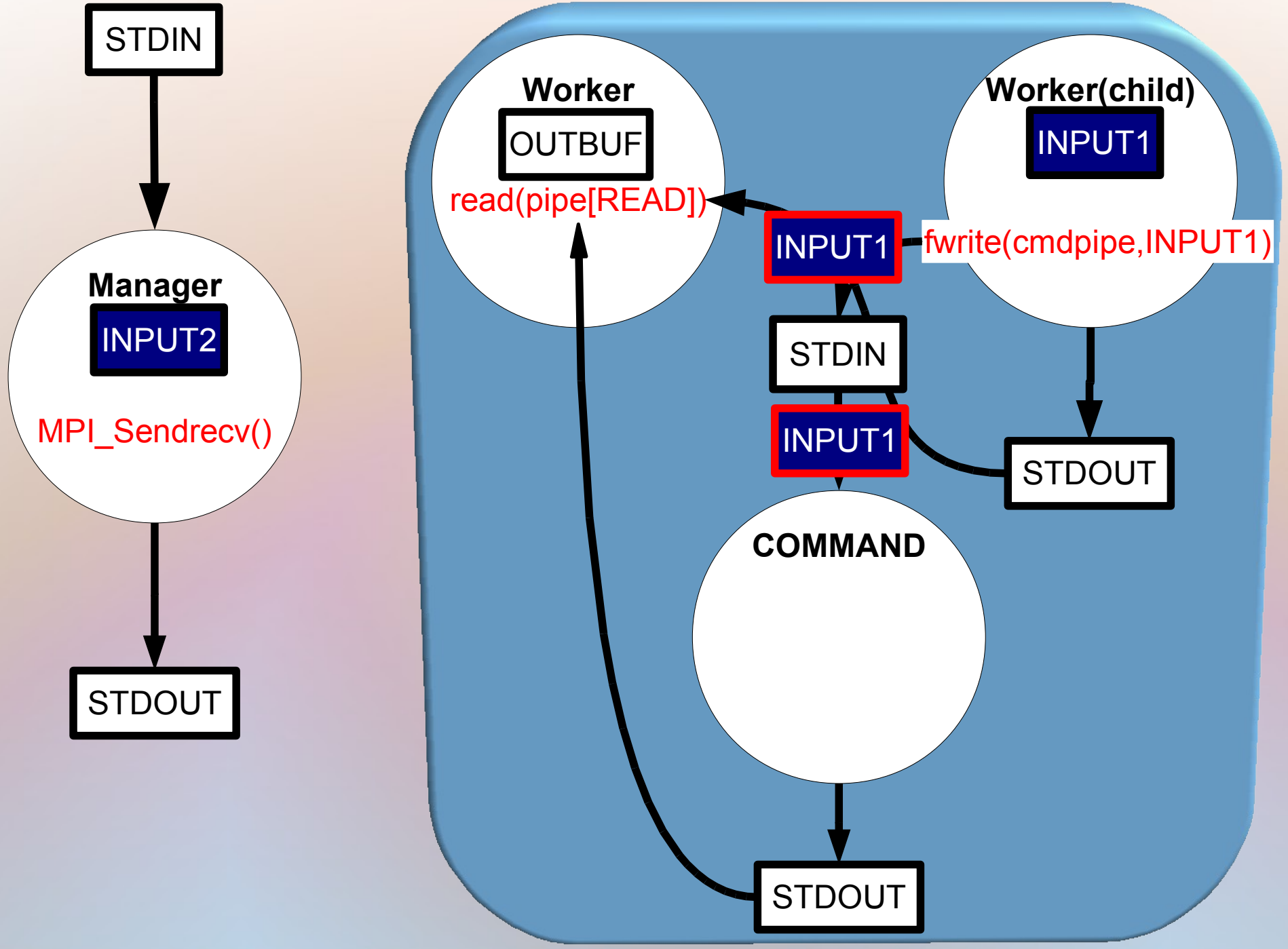

STDIN
Manager
INPUT2
MPI_Sendrecv()
STDOUT
Worker
INPUT1
pipe[READ]
free(INPUT1)
Worker(child)
INPUT1
pipe[WRITE]
STDOUT

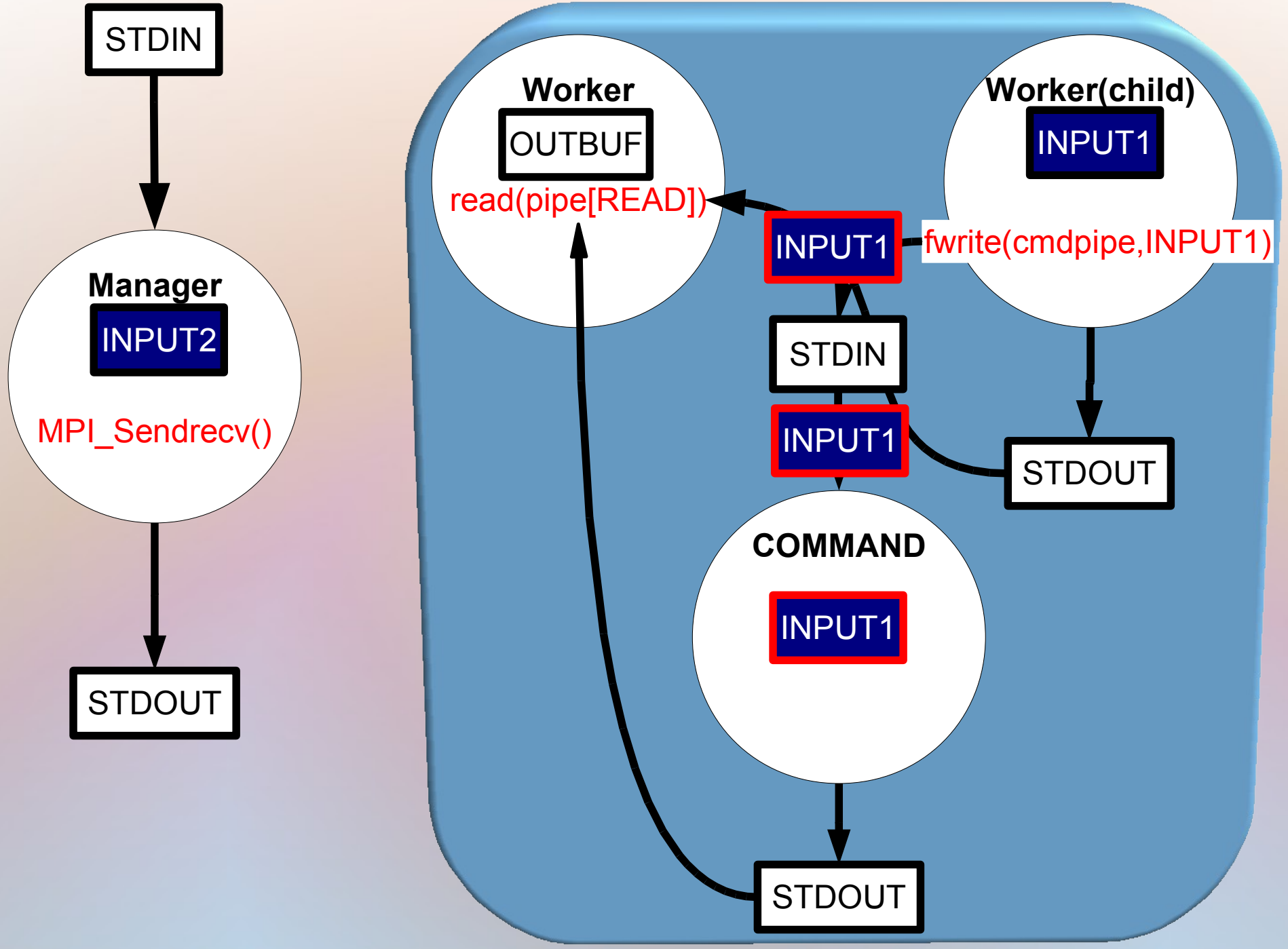

STDIN
Manager
INPUT2
MPI_Sendrecv()
STDOUT
Worker
OUTBUF
pipe[READ]
malloc(OUTBUF)
Worker(child)
INPUT1
pipe[WRITE]
dup2(pipe[WRITE], STDOUT)
STDOUT

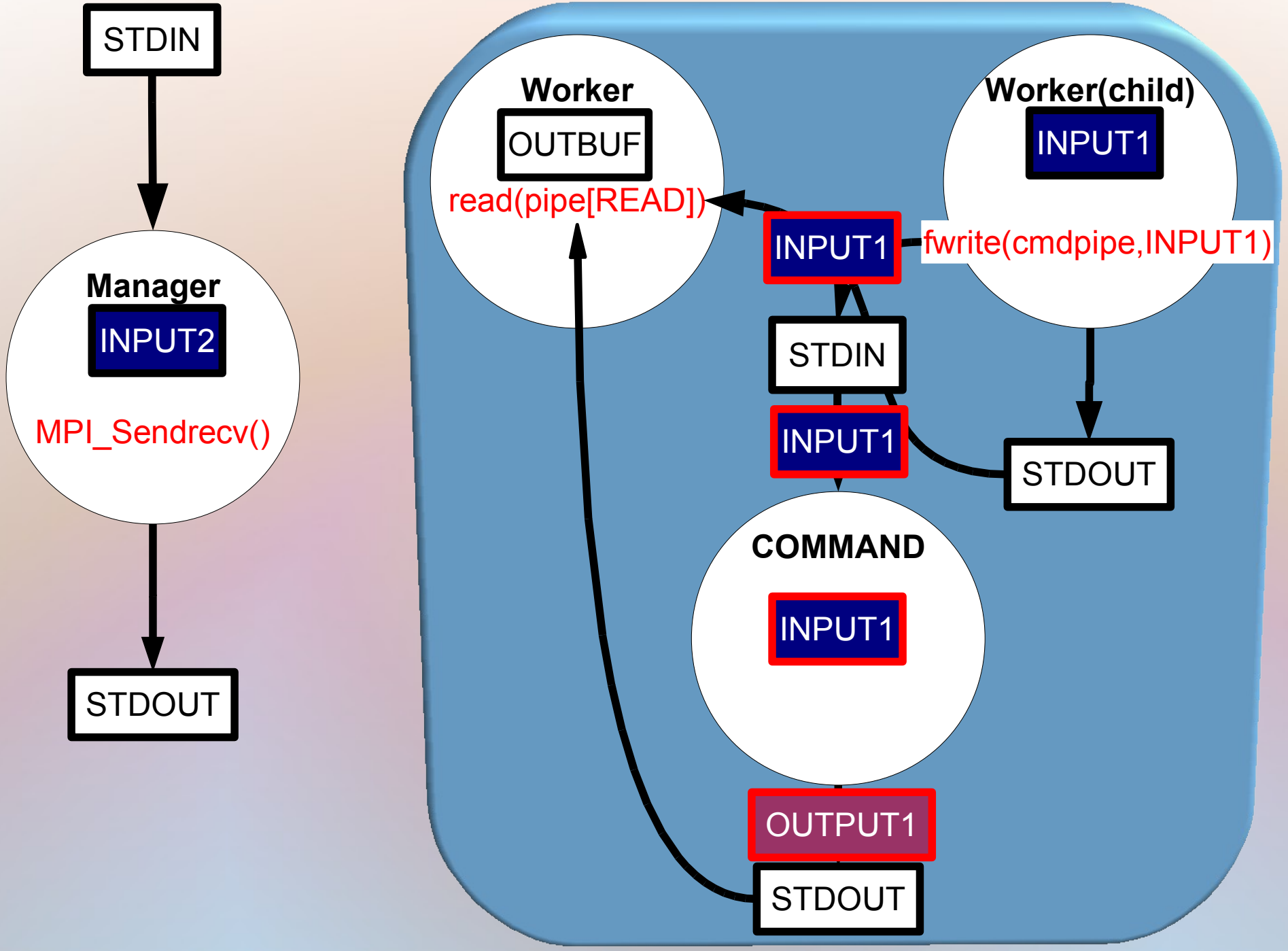

STDIN
Manager
INPUT2
MPI_Sendrecv()
STDOUT
Worker
OUTBUF
pipe[READ]
Worker(child)
INPUT1
close(pipe[WRITE])
STDOUT

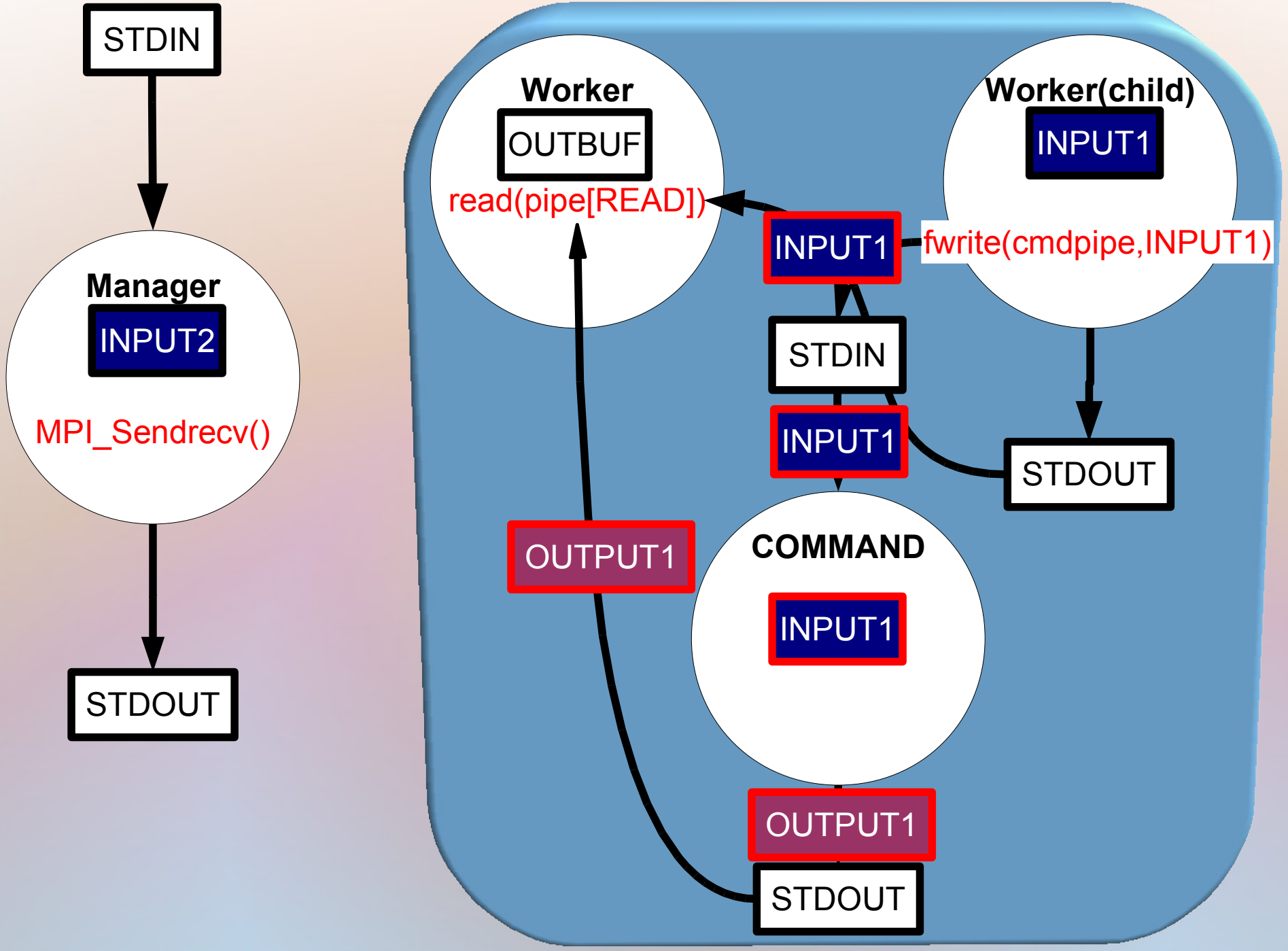

STDIN
Manager
INPUT2
MPI_Sendrecv()
STDOUT
Worker
OUTBUF
read(pipe[READ])
Worker(child)
INPUT1
popen(cmdpipe)
STDIN
STDOUT
COMMAND
STDOUT

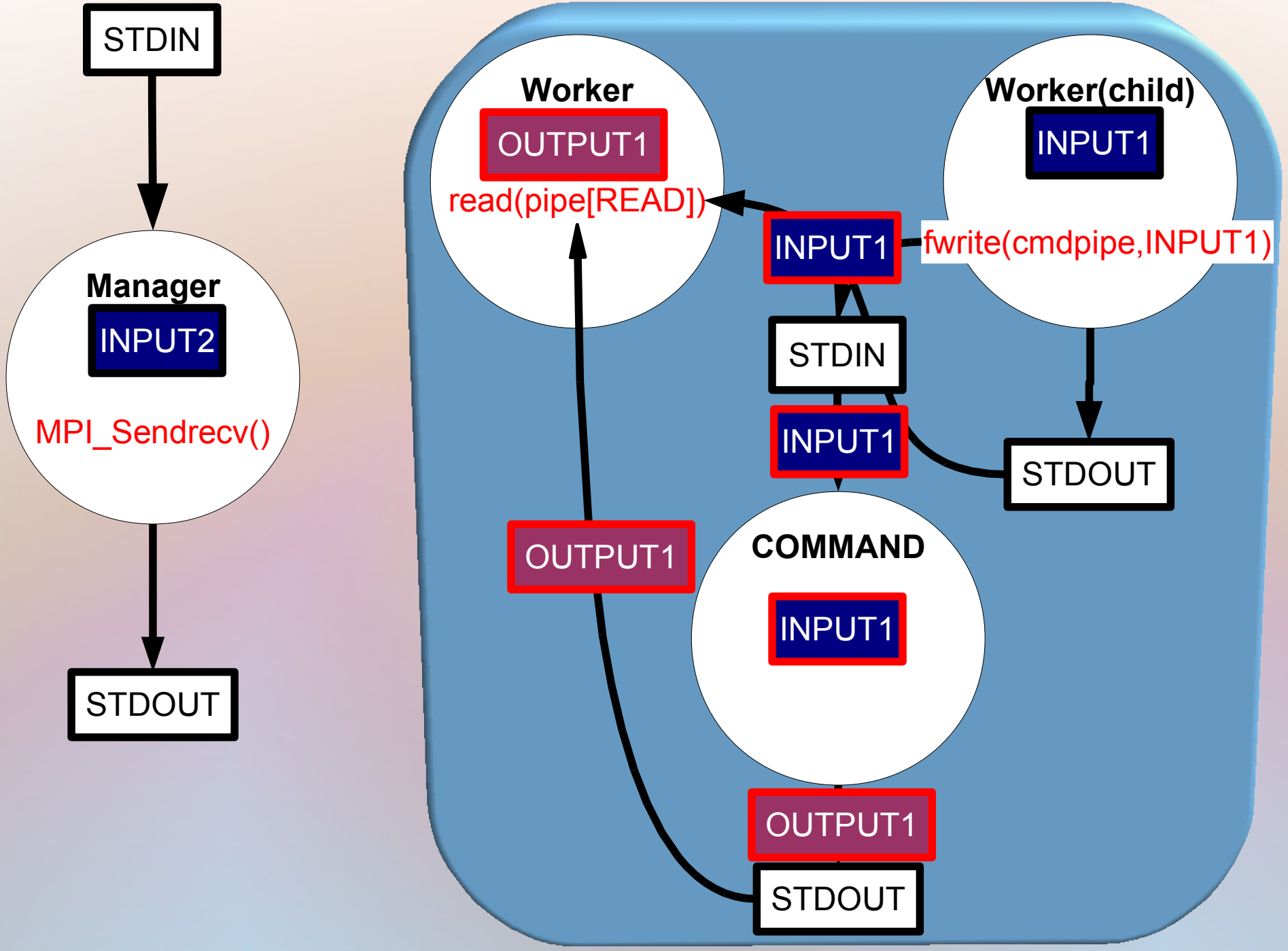

STDIN
Manager
INPUT2
MPI_Sendrecv()
STDOUT
Worker
OUTBUF
read(pipe[READ])
Worker(child)
INPUT1
INPUT1
write(cmdpipe,INPUT1)
STDIN
STDOUT
COMMAND
STDOUT

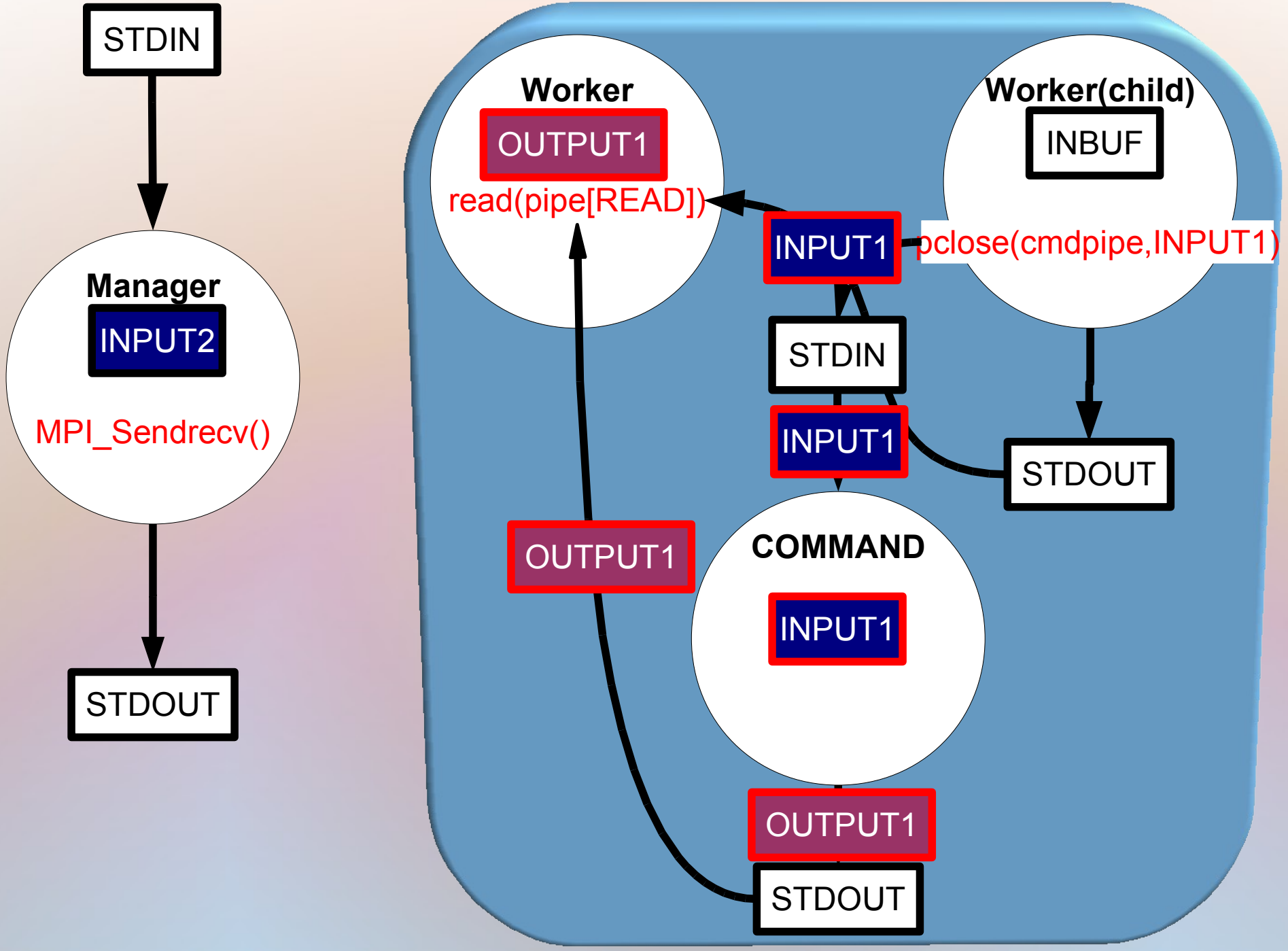

STDIN
Manager
INPUT2
MPI_Sendrecv()
STDOUT
Worker
OUTBUF
read(pipe[READ])
Worker(child)
INPUT1
INPUT1
fwrite(cmdpipe,INPUT1)
STDIN
INPUT1
STDOUT
COMMAND
STDOUT

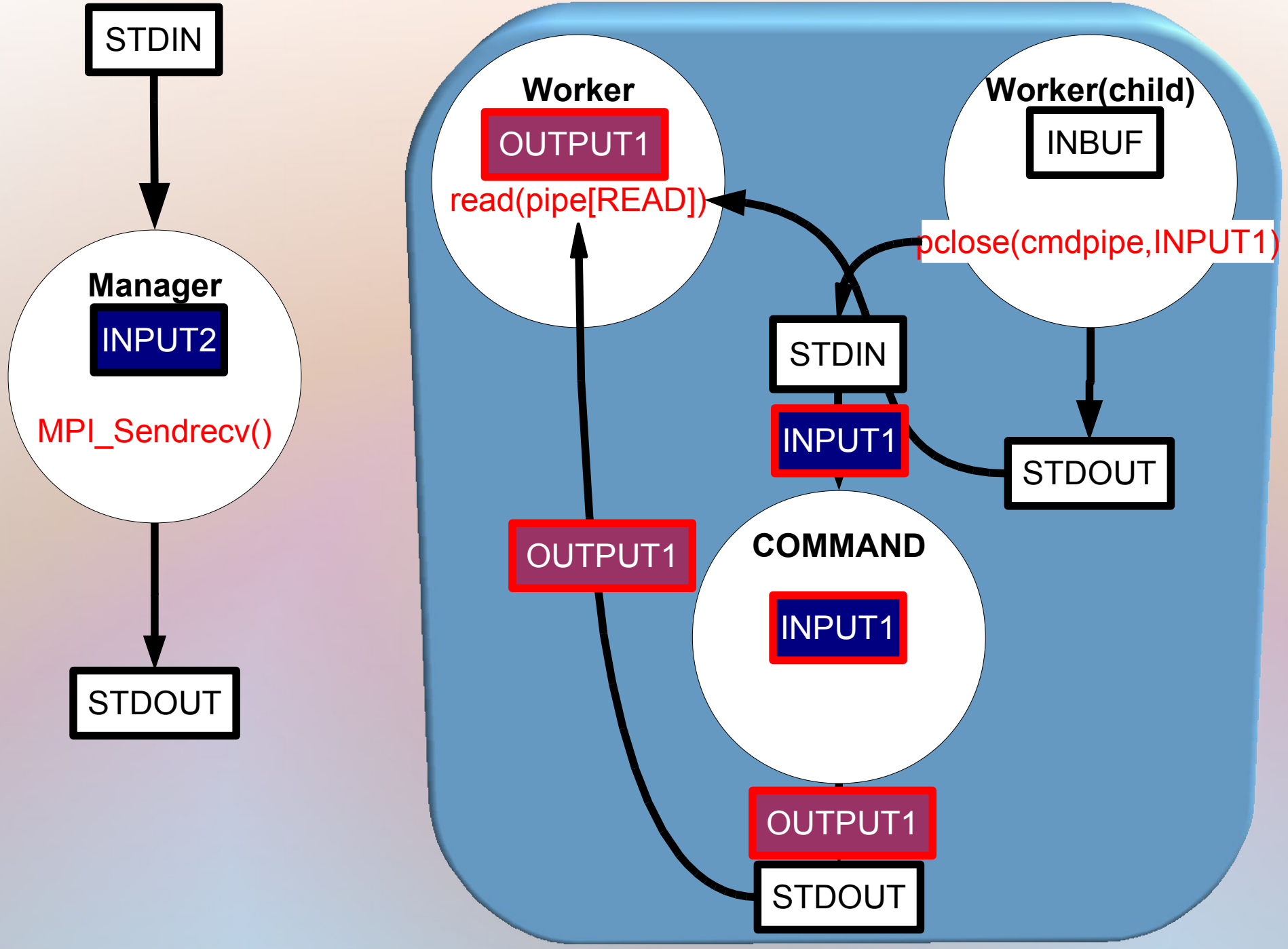

STDIN
Manager
INPUT2
MPI_Sendrecv()
STDOUT
Worker
OUTBUF
read(pipe[READ])
Worker(child)
INPUT1
INPUT1
fwrite(cmdpipe,INPUT1)
STDIN
INPUT1
STDOUT
COMMAND
INPUT1
STDOUT

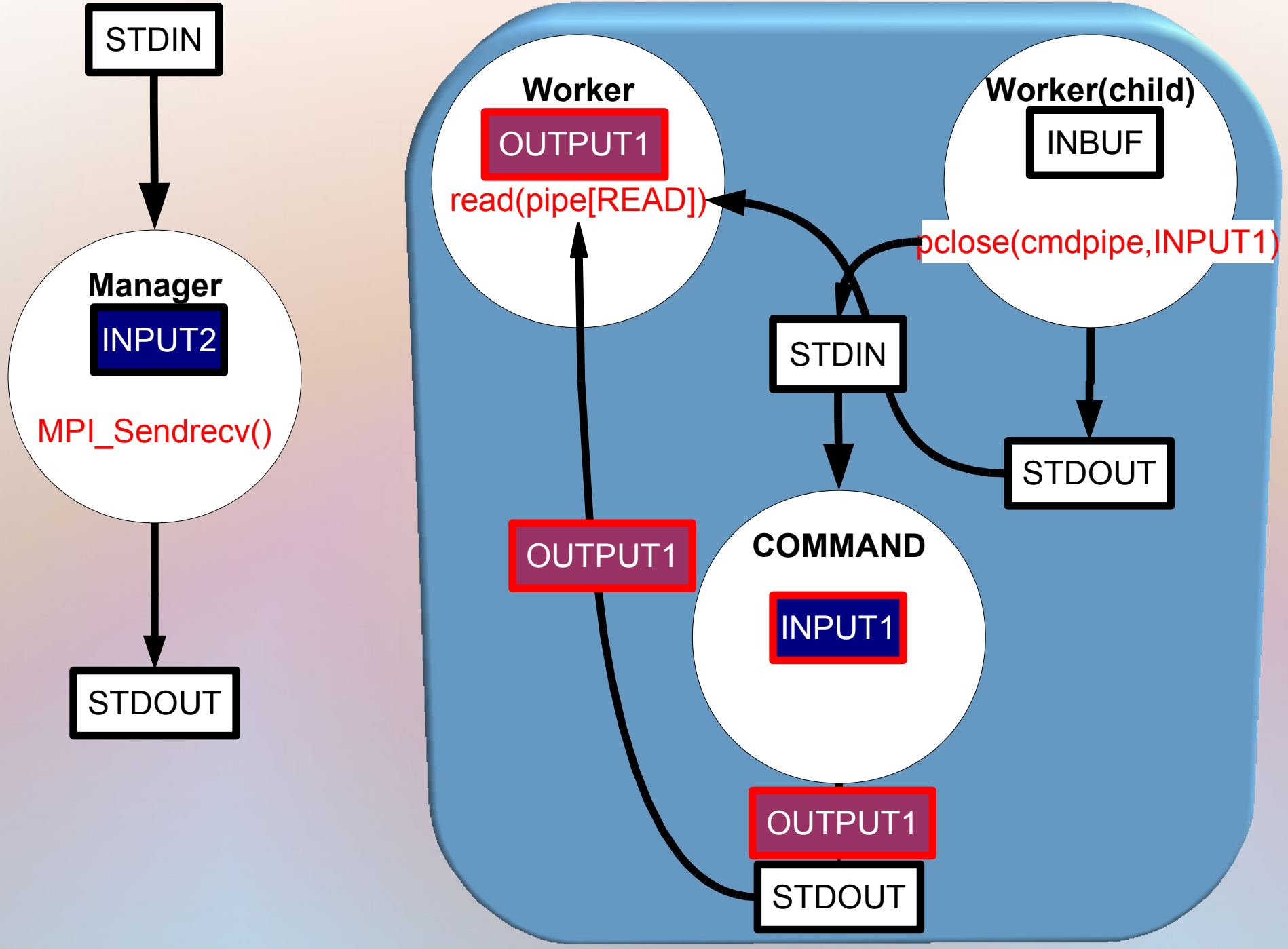

STDIN
Manager
INPUT2
MPI_Sendrecv()
STDOUT
Worker
OUTBUF
read(pipe[READ])
Worker(child)
INPUT1
INPUT1
fwrite(cmdpipe,INPUT1)
STDIN
INPUT1
STDOUT
COMMAND
INPUT1
OUTPUT1
STDOUT

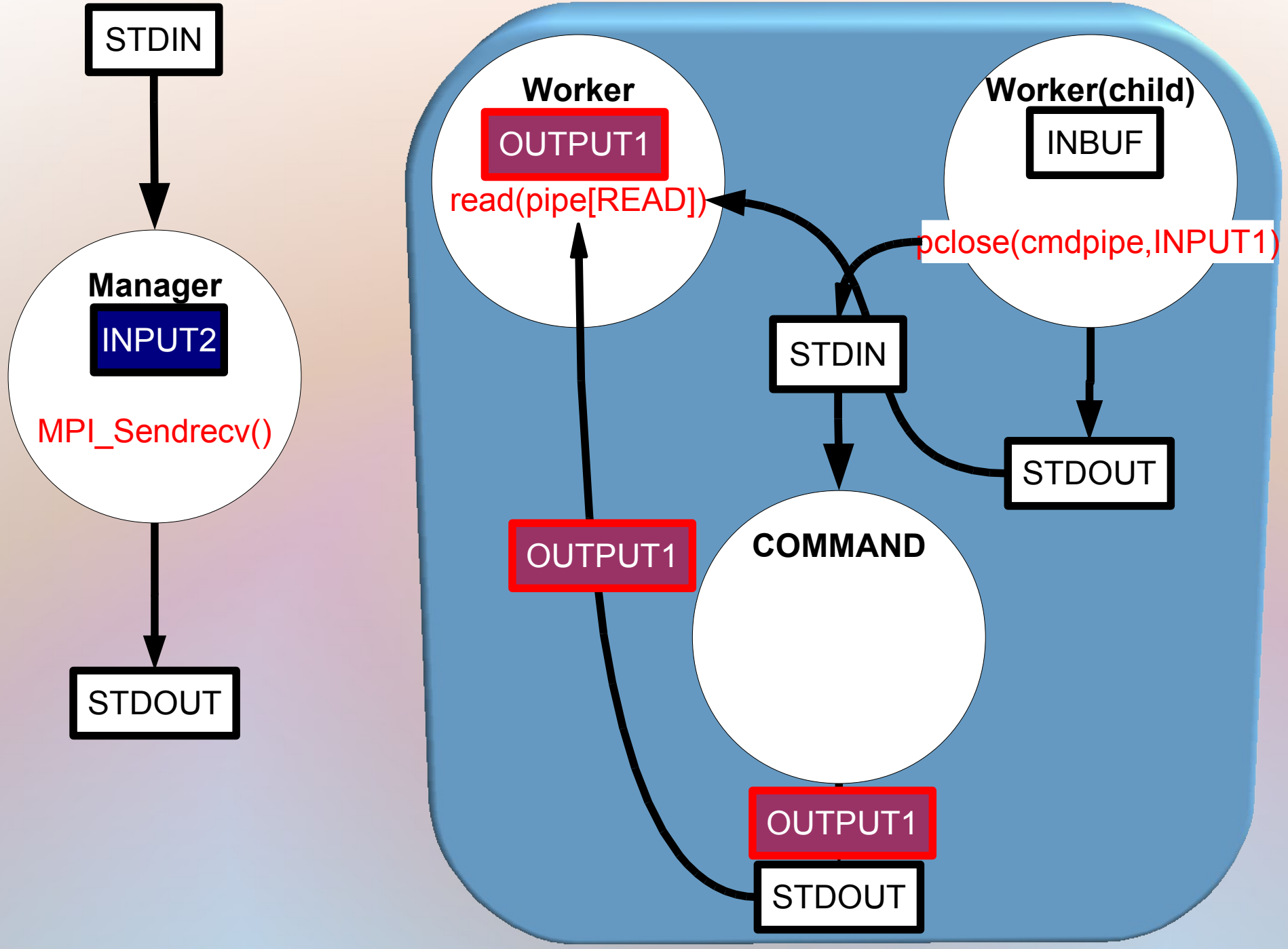

STDIN
Manager
INPUT2
MPI_Sendrecv()
STDOUT
Worker
OUTBUF
read(pipe[READ])
Worker(child)
INPUT1
INPUT1
fwrite(cmdpipe,INPUT1)
STDIN
INPUT1
STDOUT
OUTPUT1
COMMAND
INPUT1
OUTPUT1
STDOUT

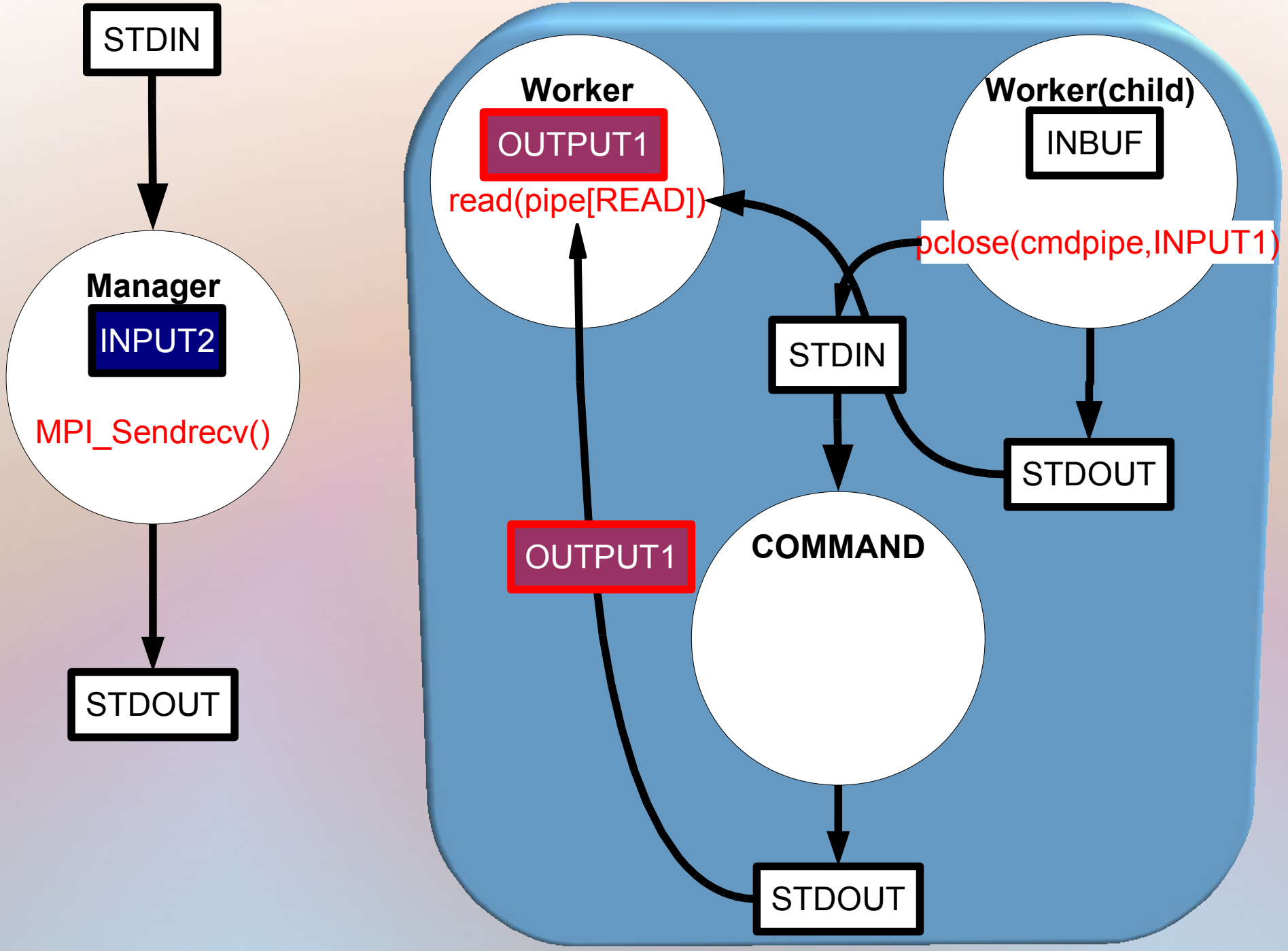

STDIN
Manager
INPUT2
MPI_Sendrecv()
STDOUT
Worker
OUTPUT1
read(pipe[READ])
Worker(child)
INPUT1
INPUT1
fwrite(cmdpipe,INPUT1)
STDIN
INPUT1
STDOUT
OUTPUT1
COMMAND
INPUT1
OUTPUT1
STDOUT

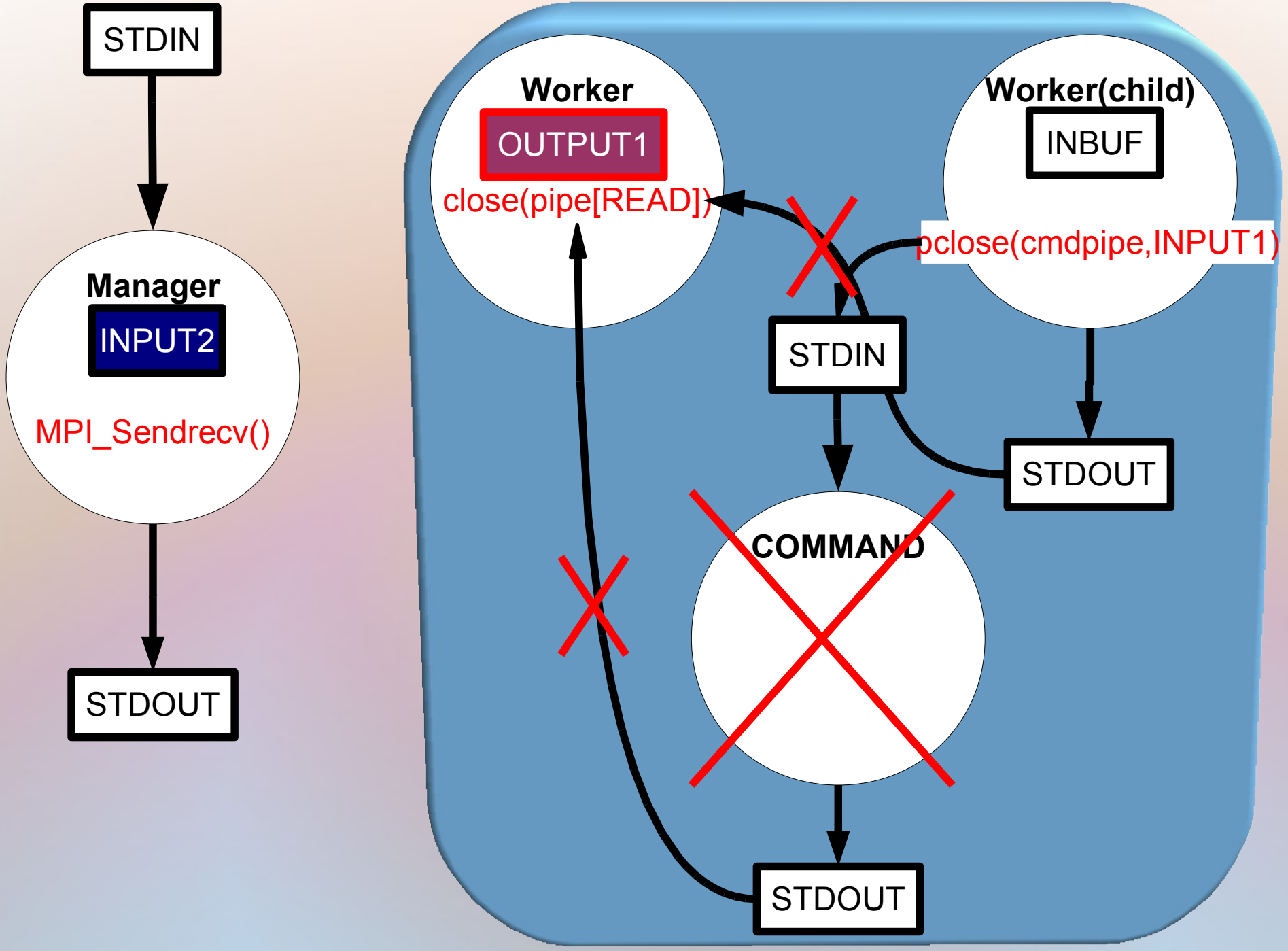

STDIN
Manager
INPUT2
MPI_Sendrecv()
STDOUT
Worker
OUTPUT1
read(pipe[READ])
Worker(child)
INBUF
INPUT1
pclose(cmdpipe,INPUT1)
STDIN
INPUT1
STDOUT
OUTPUT1
COMMAND
INPUT1
OUTPUT1
STDOUT

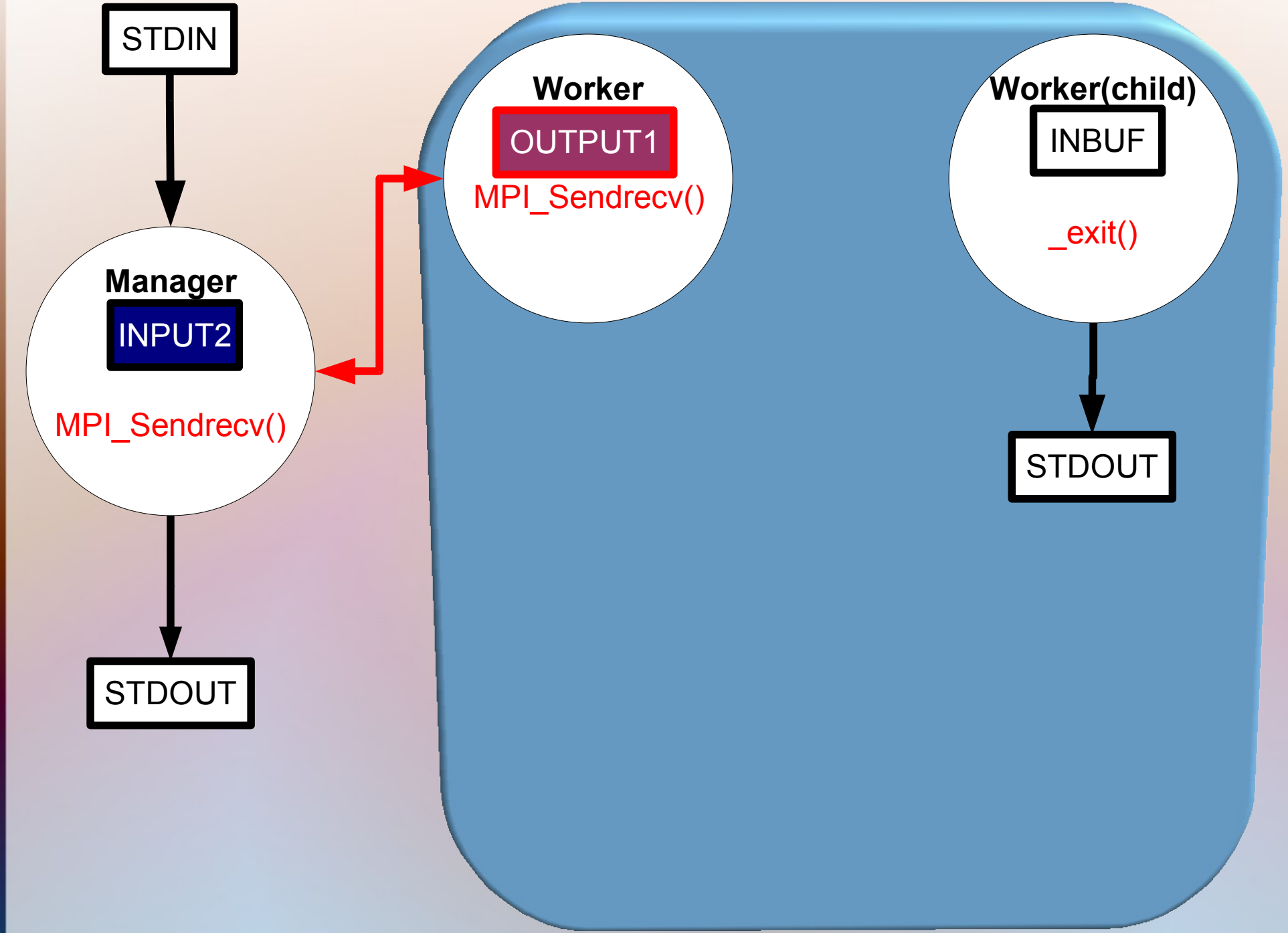

STDIN
Manager
INPUT2
MPI_Sendrecv()
STDOUT
Worker
OUTPUT1
read(pipe[READ])
Worker(child)
INBUF
pclose(cmdpipe,INPUT1)
STDIN
INPUT1
STDOUT
OUTPUT1
COMMAND
INPUT1
OUTPUT1
STDOUT

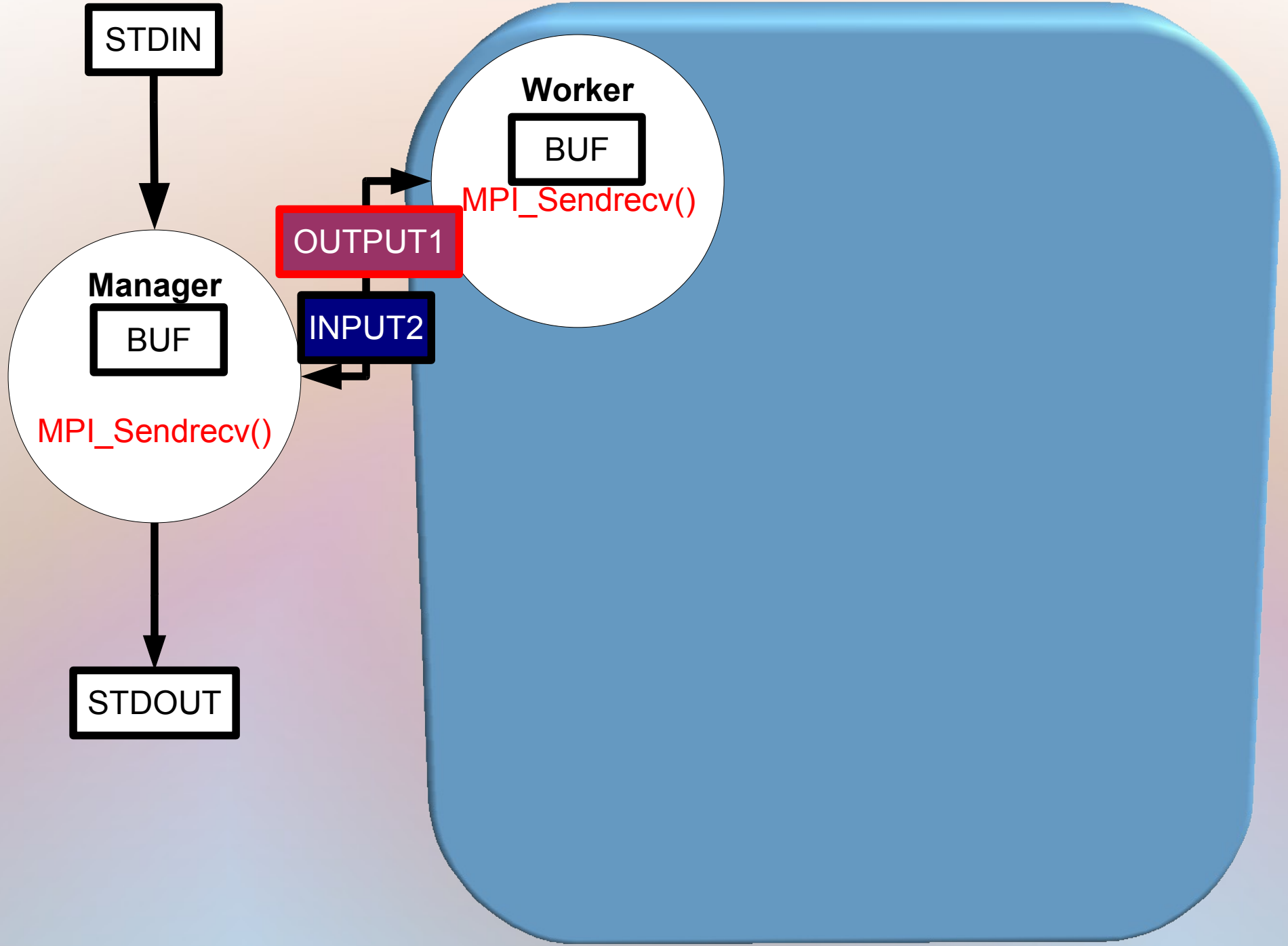

STDIN
Manager
INPUT2
MPI_Sendrecv()
STDOUT
Worker
OUTPUT1
read(pipe[READ])
Worker(child)
INBUF
pclose(cmdpipe,INPUT1)
STDIN
STDOUT
OUTPUT1
COMMAND
INPUT1
OUTPUT1
STDOUT

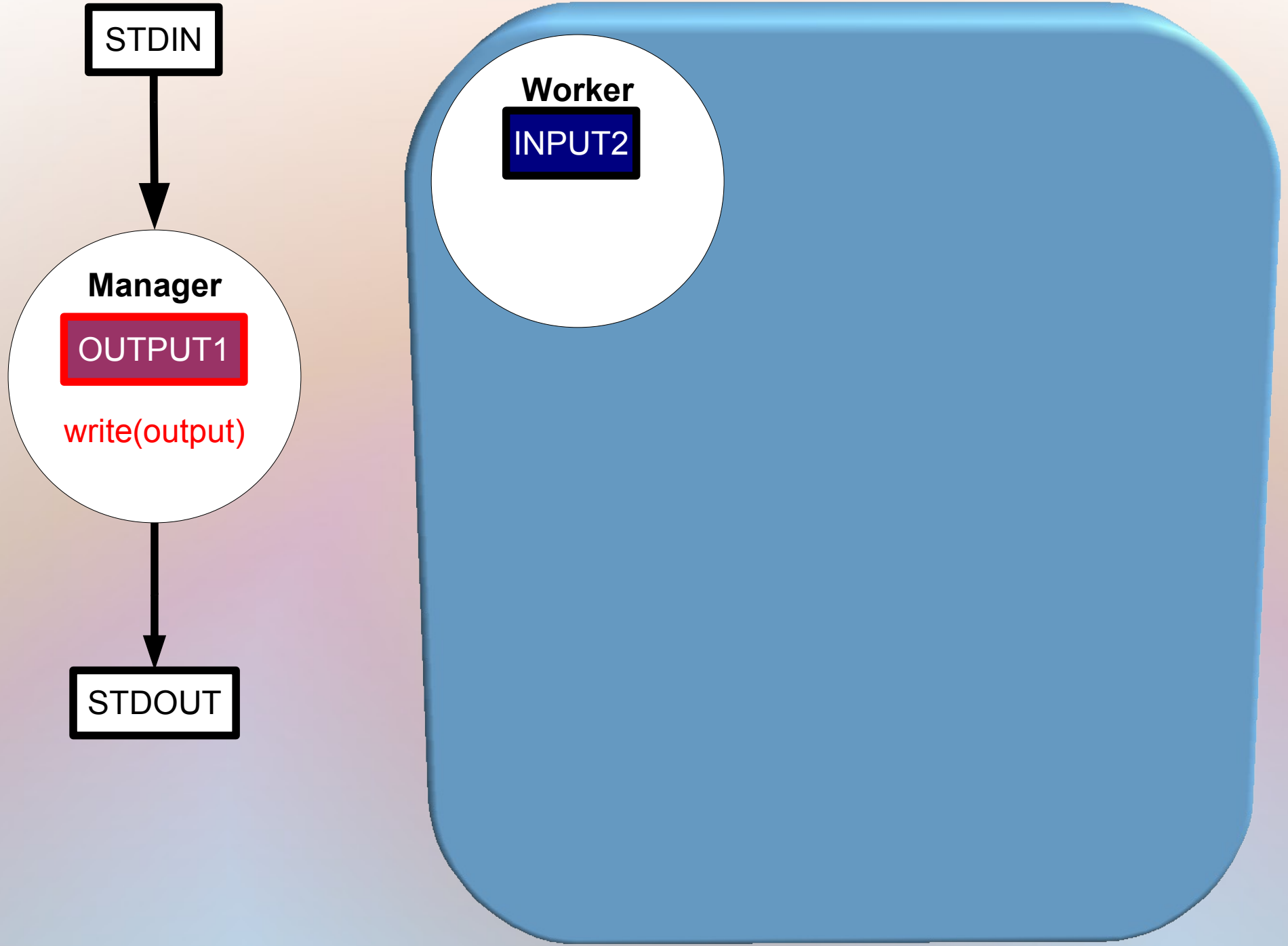

STDIN
Manager
INPUT2
MPI_Sendrecv()
STDOUT
Worker
OUTPUT1
read(pipe[READ])
Worker(child)
INBUF
pclose(cmdpipe,INPUT1)
STDIN
STDOUT
OUTPUT1
COMMAND
OUTPUT1
STDOUT

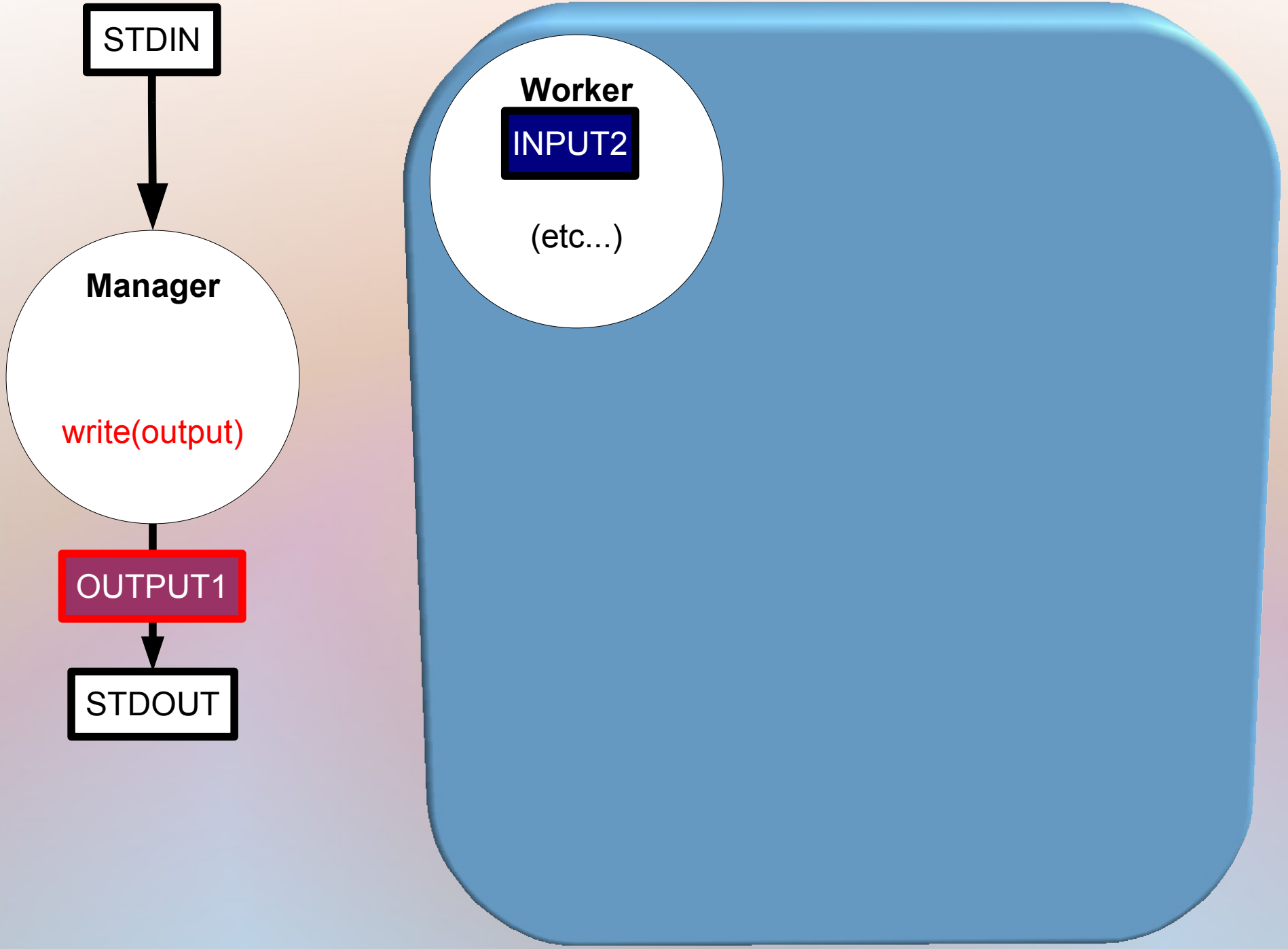

STDIN
Manager
INPUT2
MPI_Sendrecv()
STDOUT
Worker
OUTPUT1
read(pipe[READ])
Worker(child)
INBUF
pclose(cmdpipe,INPUT1)
STDIN
STDOUT
OUTPUT1
COMMAND
STDOUT


STDIN
Manager
INPUT2
MPI_Sendrecv()
STDOUT
Worker
OUTPUT1
close(pipe[READ])
Worker(child)
INBUF
pclose(cmdpipe,INPUT1)
STDIN
STDOUT
COMMAND
STDOUT


STDIN
Manager
INPUT2
MPI_Sendrecv()
STDOUT
Worker
OUTPUT1
MPI_Sendrecv()
Worker(child)
INBUF
_exit()
STDOUT


STDIN
Worker
BUF
MPI_Sendrecv()
OUTPUT1
Manager
BUF
INPUT2
MPI_Sendrecv()
STDOUT


STDIN
Manager
OUTPUT1
write(output)
STDOUT
Worker
INPUT2

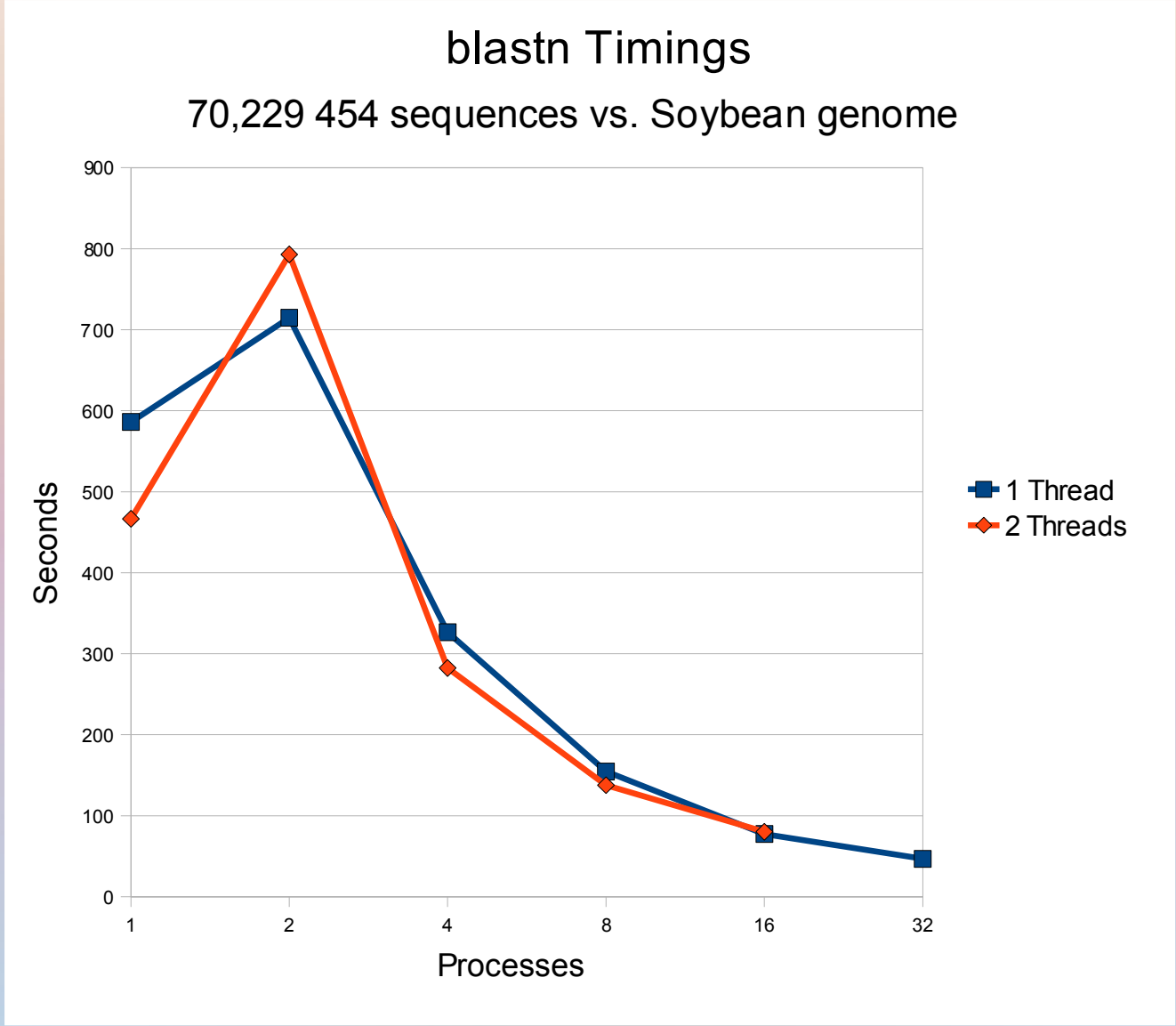

STDIN
Manager
write(output)
OUTPUT1
STDOUT
Worker
INPUT2
(etc...)

Benchmark

- System: hpc-class
- Program: blastn (NCBI blast-2.2.19+)
- Input Data
  - Query
    - 70,229 “454” EST sequences (average length approx. 233 nucleotides)
  - Target
    - Glycine max (soybean) genome

      955,054,837 (known) nucleotides

Benchmark


blastn Timings
70,229 454 sequences vs. Soybean genome
900
800
700
600
500
400
300
200
100
0
Seconds
1
2
4
8
16
32
Processes
1 Thread
2 Threads

Problems

- Open MPI uses polling for blocking MPI calls
  - Excessive CPU usage while waiting for a message
  - Possible workaround: replace “MPI_Recv(...)” with

```
received = false;
MPI_Irecv(...,request,...);
do {
    MPI_Test(request, received,...);
    if (received == true) break;
    else sleep(seconds);
} while (true)
```

Tested Configurations

- Mac OS X 10.5.x (Intel iMac)
  - Open MPI 1.2.3 (included in default installation)
- Solaris 10 (Sun Blade 1000 workstation)
  - Sun HPC Cluster Tools 8.1 (Open MPI 1.3)
- Linux (hpc-class cluster)
  - Open MPI 1.3.1

MPIPIPE

Questions?