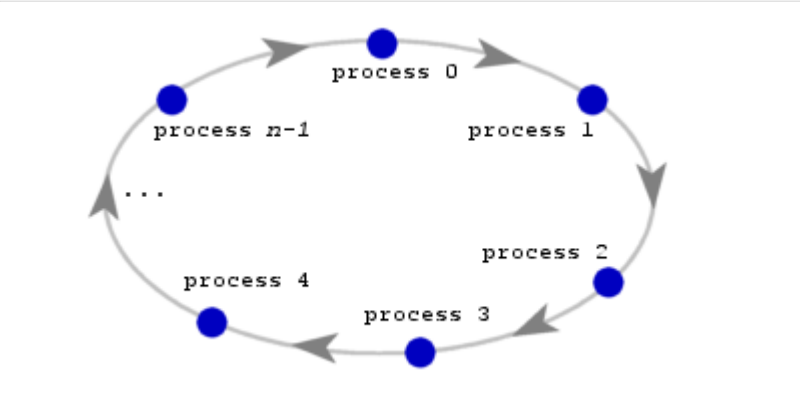
MPI Duck, Duck... Goose!

From Education



The MPI processes on LittleFe have decided that enough is enough. They've been slaving away with hello worlds, trapezoids, and maximums, and it's time for recess.

"Let's play duck, duck, goose," one little process suggests.

What is "Duck, Duck, Goose?" It's a game for kids.

A group of children sit in a circle, facing the center. One person is chosen to be the "goose." She walks around the circle, person by person, tapping each person on the head and saying "Duck."

She "ducks" player after player, until she comes to a person that she chooses to be the new "goose."

When she taps that person on the head, she says "Goose."

Then the person she called "goose" has to get up and chase her once around the circle.

Whoever wins that race, by sitting down in the new "goose's" abandoned seat, wins the race.

Then the loser -- whether the old goose or the new goose -- is now the goose, has to start walking around the circle tapping ducks.

Okay, so the MPI processes want to play "Duck, Duck, Goose."

Except... how are they supposed to play when they can't move around? With messages, of course! (This *is* MPI, after all.)

Imagine a ring of processes from `0` to *`n - 1`* where each process can only pass messages to its immediate "right" (that is, each rank can pass messages to the next higher rank, except rank `n - 1` can pass messages only to rank `0`).

This means that communication can go around the circle only "clockwise."

Write a program that will cause an MPI run of *`n`* processes to play "Duck, Duck, Goose." The server node should start the program off by selecting the first target rank who is "goose" and sending a message off clockwise to let the "goose" process know.

A few hints...

To generate random numbers, first seed the random number generator with **`srandom(time(NULL))`** and then generate a random integer with the expression

```
(random() % number_of_processes)
```

where `number_of_processes` is the number of processes in the world.

To slow down the output, you can add a call to

```
sleep(seconds);
```

(replacing `seconds` with an integer representing the number of seconds to pause before continuing).

Should this be done before or after doing an MPI_Send?

Ready, set...

Here's one potential way of approaching the problem:

- First, write an MPI program that will pass a message around in a ring. Rank 0 should send a message to rank 1, which passes it along to rank 2, which passes it to rank 3... around the ring until the message has reached the server rank again. You can do this by setting the destination equal to `(rank + 1) % size`. As each rank receives the message, have it print something indicating that.
- Once you've gotten the first part working, modify your code so that the server rank chooses an initial random target rank to start. Send the message around the ring with this value until it reaches the target rank, then end with a message.
- Once a rank has been "goosed", have that rank choose another random rank to target. Have it continue passing the message around the ring with the new target goose. At this point, your output might look something like this:

```
Rank 0: I'm going to goose rank 2...
Rank 1: Passing along a message from rank 0...
Rank 2: I was goosed and now I'm going to goose rank 1.
Rank 3: Passing along a message from rank 2...
Rank 4: Passing along a message from rank 3...
Rank 0: Passing along a message from rank 4...
Rank 1: I was goosed and now I'm going to goose rank 2.
Rank 2: I was goosed and now I'm going to goose rank 3.
Rank 3: I was goosed and now I'm going to goose rank 1.
Rank 4: Passing along a message from rank 3...
...
```

- If you've made it this far, congratulations! Time to start experimenting with tags. So far we've only had the processes continue passing around the circle clockwise. Now, use the tag to indicate which direction the message is being passed. (**Hint:** Define constants for clockwise and counterclockwise. Make sure you define them where all of the processes will get a copy. Use these for tags to indicate which direction the message is flowing.) When a rank is goosed from the right, have them respond in the same direction. In other words, each time someone is goosed, reverse the direction the message is traversing the circle. Your output might look like this:

```
Rank 0: I'm going to goose rank 2...
Rank 1: Passing along a message from rank 0...
Rank 2: I was goosed and now I'm going to get rank 2.
Rank 1: Passing along a message from rank 2...
Rank 0: Passing along a message from rank 1...
Rank 4: Passing along a message from rank 0...
Rank 3: Passing along a message from rank 4...
Rank 2: I was goosed and now I'm going to get rank 0.
Rank 3: Passing along a message from rank 2...
Rank 4: Passing along a message from rank 3...
Rank 0: I was goosed and now I'm going to get rank 0.
Rank 4: Passing along a message from rank 0...
Rank 3: Passing along a message from rank 4...
```

You may want to grab some information from MPI_Status. An MPI_Status struct contains three values: the source of the message (`MPI_SOURCE`), the tag from the message (`MPI_TAG`) and the return value (`MPI_ERROR`). Assuming you have declared an **`MPI_Status status`** then you can access the source of the message, for example, with **`status.MPI_SOURCE`**.