

# Exercises 7

When experimenting with MozartSpaces, remember that abnormal program shutdown (initiated by pressing `ctrl+c`, for example) may not stop certain background network processes. If your program does not work properly when it should, try to restart the console and run the program again.

## Task 7.1. Code Breakers

The project in [ex07\\_codeBreakers.zip](#) implements a “code breaking service” implemented with a producer-consumer architecture. It consists of three modules that should be compiled and run separately:

```
javac -cp .;mozartspaces-2.3.jar ex07_cbServer.java
javac -cp .;mozartspaces-2.3.jar ex07_cbWorker.java
javac -cp .;mozartspaces-2.3.jar ex07_cbClient.java

# "Server": run one instance in a separate console
java -cp .;mozartspaces-2.3.jar ex07_cbServer

# "Worker": run any number of instances in separate consoles
java -cp .;mozartspaces-2.3.jar ex07_cbWorker

# "Client": run any number of instances in separate consoles
java -cp .;mozartspaces-2.3.jar ex07_cbClient
```

A client connects to the server and places a “code-breaking job”, which consists of a password-encrypted string. A worker connects to the server and takes the first available job to complete. Once the job is done, the worker publishes the result, so that the client can retrieve it.

Compile and test this project with several clients and workers.

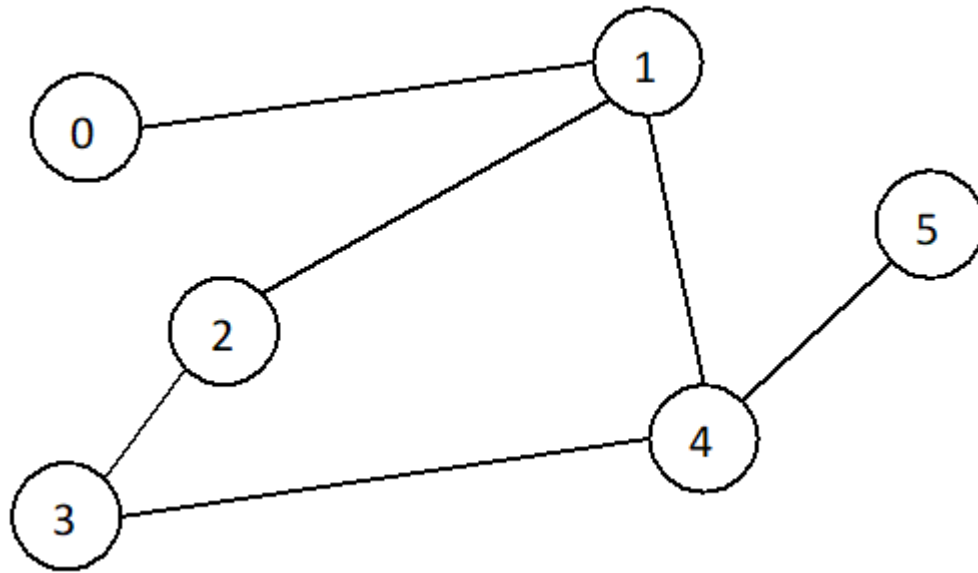
In real systems, it is common to run multiple workers on the same physical machine, which might cause uneven load: some machines work hard while others are almost idle. This issue is solved by load balancers that distribute work to least busy machines.

Your task is to introduce very simple load balancing into “Code Breakers”. Instead of one job container, create two. When starting a worker process, randomly associate it with only one of these two job containers (so the given worker will only takes jobs from one container).

When starting a client process, publish a job randomly in one of the two available job containers.

## Task 7.2. Define Topology v2

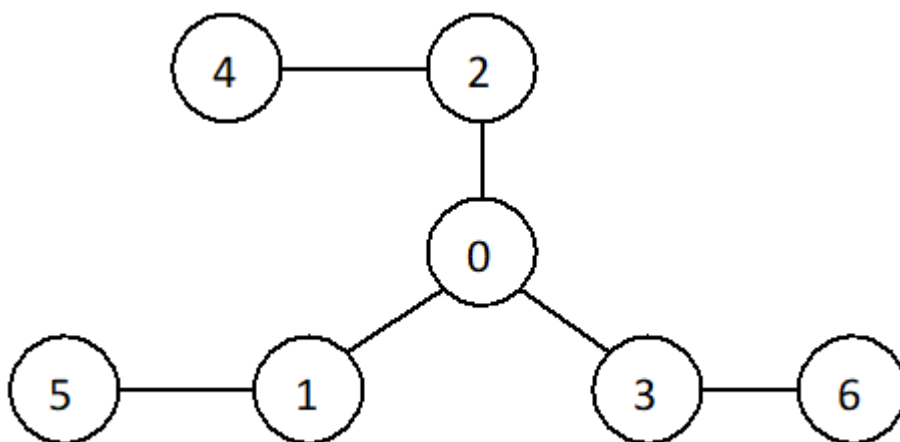
Write the MPI definition (`index` and `edges` arrays) for the following graph topology:



## Task 7.3. Gaussian Prime Star

Write an MPI program that acts as a prime numbers-finding farm. The task of this program is to find the **amount** of all Gaussian prime numbers in the interval  $[2 \cdots M]$ . (So it is enough to count the numbers found, there is no need to keep the numbers themselves.) The value of  $M$  is specified by the user as a command-line argument. Gaussian primes are prime numbers (having no divisors except one and the number itself) representable as  $4k + 3$ . For example, the first five Gaussian primes are 3, 7, 11, 19, and 23.

The task should be performed by a network with the following graph topology as outlined below:



- A process with a single neighbor waits for the interval  $[Left, Right)$  from its only neighbor, searches Gaussian primes inside the interval and sends back the results (a number of found primes).

- A process with two neighbors waits for the interval  $[Left, Right)$  from any other process, divides this interval into two halves  $h_1$  and  $h_2$ , and sends  $h_1$  to its neighbor who was *not* the sender of the interval. Let's call the neighbor who sent the job  $N_S$ ; then  $h_1$  should be sent to the other neighbor that we will call  $N_R$ . Next, the process searches Gaussian primes inside  $h_2$ , combines the obtained result with the answer received from  $N_R$ , and sends back the resulting answer to  $N_S$ .
- A process with more than two neighbors divides the original interval  $[2, M)$  into equal parts and sends the parts to all of its neighbors. Then it combines the results obtained by the neighbors and prints the final answer.

**Hints.** 1) Send process ID to the receiver along with other data when the receiver needs to know who is the sender; 2) Don't try to use collective `Reduce()` operation to get the final result: it works for all processes in the system, and unfortunately you can't call it for an arbitrary subset of processes.

## Task 7.4. Mozart Space Bank

Let's model the bank account / money problem in MozartSpaces. As discussed in Talk 3, there are two users and a shared bank account. Both users add  $N$  money units to the account and finish their work. The task of the bank is to make sure that the account will contain exactly  $2N$  money units as a result of this operation.

In MozartSpaces representation, we will have three processes. The bank process [ex07\\_mozartBank.java](#) creates a shared space and publishes `Account` container with a single zero-valued integer entry. Then the bank waits until `signal` container has two entries and prints the resulting account value. Each of two client processes connects to the shared account, adds  $N$  money units and appends one entry to `signal` container.

Your task is to implement the client process. Use  $N = 2000$  in your tests: MozartSpaces is not very fast, so it might take much time to perform a large number of operations.