Class Project – Task Bag using Java RMI

Note:
1. Submission must be made in electronic format to the assistant.
2. You need to work in groups, choose your partners and inform the instructor.

Objective

The objective of the class project is to build a simple generic toolkit that enables processes running in different computers to carry out parts of a large computation in parallel. The general idea is that a master process places sub-tasks of the computation in a Task Bag and worker processes select and take tasks from the Task Bag and carry them out, returning the results to the Task Bag. The master then collects the results and combines them to produce the final result.

The concept of Task Bag comes from the Linda system, which was designed by Carriero and Gelertner at Yale University, [Ahuja, Carriero and Gelertner 1986], [Carriero and Gelertner 1989]. In Linda the task bag is implemented as distributed shared memory that acts as a persistent object store and provides a small number of simple operations such as take, place etc. In Linda terminology, the object store is called space and it is used for different purposes from reliable exchanging messages to perform parallel computations. It is a different programming style, which is useful in solving many distributed problems. The JavaSpaces implements the same idea and is an integral part of Jini [Eric Freeman and Susan Hupfer 1999], [Eric Freeman 2000].

In this project you are asked to implement the Task Bag as a Remote Object (using Java RMI) and to use it as a basis for performing a parallel computation of a big problem on several computers. You can choose any big problem that can be performed in parallel to demonstrate the results. However, your implementation must be generic and it must be easy to adapt it to any type parallel computation.

Three entities are involved:

1. The Task Bag object,
2. The master process
3. The worker processes.

The master and worker processes are clients of the Task Bag object.

The Task Bag Object

The Task Bag is an object whose functionality is to provide a repository for Pairs. Each Pair may be regarded as a task description. A Pair consists of two parts - a key and a value. The value contains the actual description of a task and the key is anything that can be used to reference the Pair. A typical key might be a task name or number. A task description may be used by the master to describe tasks and by workers to describe results. Clients may: i) add task descriptions to the Task Bag, ii) remove them from the Task Bag iii) retrieve them from the Task Bag. To access a task description, the client must specify a key.
The Task Bag object will offer three operations `placePair`, `takePair` and `readPair` in its interface. They are defined as follows:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>placePair (key, value)</code></td>
<td>causes a Pair (key, value) to be added to the Task Bag. The client process continues immediately (non-blocking);</td>
</tr>
<tr>
<td><code>takePair (key)</code></td>
<td>causes some Pair in the Task Bag that matches key to be withdrawn from the Task Bag; the value part of the Pair is returned and the client process continues. If no matching Pair is available, the client waits (blocks) until one is available and then proceeds as before. If several matching Pairs are available, one of them is chosen arbitrarily;</td>
</tr>
<tr>
<td><code>readPair (key)</code></td>
<td>is the same as <code>takePair(key)</code> except that the Pair remains in the Task Bag.</td>
</tr>
</tbody>
</table>

**The Application**

You should choose one application that requires fairly intensive computation to carry it out and which is easily divided into a number of identical subtasks. You could consider tasks such as:

- (i) Breaking an encryption algorithm by exhaustive search
- (ii) finding prime numbers
- (iii) parallel compilation of the modules of a program (parallel make);
- (iv) searching for files containing a particular text string;
- (v) matrix multiplication or
- (vi) fractal images.

**Parallel Programming with the Task Bag**

We consider the sort of parallel program that involves a transformation or series of transformations to be applied to all the elements of some set in parallel. This type of parallelism is suitable for modeling with the master-worker paradigm.

A master process provides a set of tasks to be done by a collection of identical worker processes. Each worker is capable of performing any one of the steps in a particular computation. In the simplest cases, there is only one step. A worker repeatedly gets a task, carries it out and then puts the result in the Task Bag. The results are collected by the master. The program executes in the same way whether there are 1, 10 or 1000 workers.

We refer to two examples throughout this explanation. In the first example, the joint task is to generate all the prime numbers less than some limit, `MAX`. We use one master process together with one or more worker processes. The master process sets up the first task and then waits to collect the prime numbers calculated by the workers. Each worker process repeatedly gets a range of numbers within which to search for prime numbers. Each worker places the sets of primes it finds in the Task Bag, from whence the master may collect them.

As a second example, we consider a program that exhaustively searches the key space to break a block cipher algorithm such as AES or Triple DES given a ciphertext and plaintext pair. Master process has the following responsibilities: (i) generate the candidate keys...
randomly (ii) place the first task in the Task Bag (iii) retrieve the results and checks whether
the key is found (iv) stop the computation if the key is found. The worker processes (i) take
next keys to be tried from the Task Bag and (ii) place next keys in the Task Bag for other
workers (iii) encrypt the given plaintext with these keys, (iv) check if the ciphertext is the
same as the given ciphertext and (v) place the result (FOUND or NOTFOUND) in the Task
Bag.

How the workers know which task to perform next

In many computations, there is a collection of tasks, numbered from First to Last. Each
worker repeatedly carries out one (or a group) of tasks. Before a worker starts, it needs to
know which task to do next. A Pair with the key “Next Task” can be used for this purpose.
The master process places the first task:

\[
\text{placePair}(“Next\ Task”, \text{First});
\]

and each worker, in turn, takes the Pair out, increments its value and puts it back. e.g.

\[
\text{nextElement} = \text{takePair}(“Next\ Task”);
\]

\[
\text{placePair}(“Next\ Task”, \text{nextElement + GRANULARITY});
\]

The number of tasks done together is a per application constant (GRANULARITY). When
there are no more tasks to be done, the worker does not replace Next Task in the Task Bag.
When other workers attempt to remove it, they will have to wait. No more work will be done
until the master supplies another collection of tasks to calculate.

In the prime numbers example, the worker calculates primes within the range nextElement to
nextElement + GRANULARITY -1.

In the key search example, the master process generates random keys (without repeating the
same key) and groups them into sets that can be handed over to a worker that can process all
the keys in the set at one time. The master starts the computation by placing the number of
one of the key sets in the Task Bag as follows

\[
\text{placePair}(“Next\ Task”, \text{FirstKeySet});
\]

A free worker removes the task by issuing the following call:

\[
\text{NextKeySet} = \text{takePair}(“Next\ Task”);
\]

(the worker uses NextKeySet which essentially a number to access the actual set than contains
the keys). The same worker has the responsibility of placing another key set in the Task Bag.
It does this as follows:

\[
\text{placePair}(“Next\ Task”, \text{NextKeySet + 1});
\]
Other than Task pairs, there are two other types of pairs in the task bag: **Data and Results Pairs.** Task pairs indicate the next task to be done, Data pairs indicate sets of key data needed by the Workers to perform encryption operation, and Result pairs are used to indicate whether the key is found or not.

In order for the Worker to begin its work, there must be a **Task Pair** available in the Task Bag. The Master is responsible for placing the first Task Pair in the Task Bag. When the Task Bag receives a Task Pair, it will immediately send a notification to all free Workers that a new task is available. The first free Worker to respond will remove the Task Pair and replace it with the next Task Pair indicating the next task that needs to be done. If another free Worker also requests the Task Pair before the first Worker replaces it, it will simply fail to read the Task Pair and wait for another Task Pair notification from the Task Bag. In this way there will never be more than one Task Pair in the Task Bag.

The key for every Task Pair will be “Next Task”. The master process that creates key sets also numbers them starting from 1. The Task Pair value will consist of a value indicating this number of the key set.

Here is an example of typical placePair/takePair calls:

```java
placePair("Next Task", "1");  // call by Task Master to start computing
// the first set
NextKeySet = takePair("Next Task");  // call by some free Worker
placePair("Next Task", NextKeySet+1);  // call by same Worker to replace Task
// Pair
NextKeySet = takePair("Next Task");  // call by another free Worker
placePair("Next Task", NextKeySet+1);  // call by same Worker to replace Task
// Pair
...
NextKeySet = takePair("Next Task");  // call by another free Worker
// If it is the last set, it will not replace
// Task Pair
```

**Data for the workers**

In some computations, the workers need data in order to perform their task. For example in key search program, a worker needs a set of keys in order to check if any key in the set is the one yielding the given ciphertext. This data is put in the Task Bag by the master and may be accessed by workers that need it. The master can put the key sets as follows:

```java
placePair("KS1", <first key set>);
placePair("KS2", <second key set>);
...
```

In this example, since only one worker will use a key set, the worker can use `takePair` to remove it from the Task Bag indefinitely.

```java
String key = "KS"+ NextKeySet;
```
keySet = takePair(key) ;
...

In the calculation of prime numbers, a worker calculates whether a number, \( n \) is prime by dividing it by all the prime numbers up to \( \sqrt{n} \). Therefore, the worker needs to know the previously calculated primes up to \( \sqrt{n} \). As the master collects the primes calculated by the workers it can put them in order and then place copies of sets of them in the Task Bag for use by the workers.

The workers' results

It is important to note that many workers perform similar tasks and generally return values with identical keys to the Task Bag. The Task Bag must be implemented so that many Pairs with the same key may be held at the same time.

In the prime numbers example, all the results calculated by the workers may bear the same key: "Result". A worker can place a collection of prime numbers in the Task Bag as follows:

\[
\text{placePair("Primes", \langle a\ collection\ of\ primes\rangle);}
\]

The master just collects all the Pairs with the key Primes e.g., by:

\[
\langle a\ collection\ of\ primes\rangle = \text{takePair("Primes")}
\]

In key search application, a worker may produce two types of result: NOTFOUND and FOUND + ENCRYPTION_KEY. It uses the same numbers to place the result in the Task Bag:

\[
\text{placePair("Result"+NextKeySet, \langle Result\rangle);}
\]

When a Result Pair is received by the Task Bag, it notifies the Master of the new result. The Master will retrieve the result using a takePair call to remove the answer from the Task Bag. When the encryption key is found, the master should display it and notify all workers of the completion of the task.

The Remote Interface of the Task Bag

To carry out this exercise, you need to define the Task Bag interface in Java. You should consider whether to use exceptions or return values to indicate when the operations cannot be carried out. You should also consider the use of exceptions for genuine errors in the arguments.

Monitoring
The above arrangement is not fault-tolerant. If a worker fails before completing a task, the master will have to wait when it attempts to read the corresponding result. In our example, if a worker fails between removing the value of Next and replacing the next value, all the workers will have to wait for an indefinite time.

The user who starts the parallel computation should be able to monitor its progress. The monitor should report on the state of the computation and provide the ability to recover from incomplete computations. This may require you to add some operations in the interface of the Task Bag object.

**Synchronization of Client Operations**

There are several approaches to the case where no matching pair is available for a client performing a `takePair` or a `readPair`.

There are two approaches that could have been used in implementing this project: polling and callbacks. Polling would require the Master to constantly poll the Task Bag for Result Pairs and Workers to poll for Task Pairs. Both the Master and Workers would continually poll after some specified amount of time, say 2 seconds, before polling again. Polling has several disadvantages. Not only does it waste network resources by making repeated calls to the Task Bag, it also slows down the Task Bag by forcing it to respond to unnecessary calls. Furthermore, there may be new Result Pairs or Task Pairs that arrive in the Task Bag between delays in the polling. This means there are tasks needing to be performed but no process aware of it. Decreasing the delay time reduces this problem but increases the previously mentioned problem of wasting network resources.

Callbacks allow the Master and Workers to be notified of new Result Pairs and Task Pairs when they arrive. This not only reduces network resources but ensures that Task Bag data will be retrieved as soon as computationally possible. The drawback with this approach lies in the implementation. The Task Bag must maintain information about which Workers are available for work; and in order not to bother busy Workers; it also needs to maintain “busy” state information for each Worker. When Workers are made available or cease processing, the Task Bag also needs to be notified. This increases the overhead necessary for the Task Bag.

When comparing the advantages and disadvantages to polling and callbacks, it can be argued that although callbacks require more overhead, the savings on network resources (which are typically in much shorter supply) make callbacks the preferred implementation.

Callbacks can be implemented in RMI as follows:

- The client creates a remote object that implements an interface that contains a method for the Task Bag to call.
- The Task Bag allows clients to register their remote interfaces that the Task Bag record in a list. (For example, a worker may just join the worker group and lets the Task Bag know it is ready to participate in the computation).
- Whenever there is task in the Task Bag, the Task Bag object notifies the clients using these remote interfaces.
Implementation Design

These are the Java classes that make up the Master, Task Bag, and Workers.

Master

Master – Displays the GUI, registers the Task Bag object with RMI, registers for notification (using callbacks) with the Task Bag, and generates random numbers and group them into sets.

MasterFrame – Actual GUI implementation

ControlPanel – Displays control panel.
PlaintextPanel – Displays the plaintext.
CiphertextPanel – Displays the corresponding ciphertext.
Result – Displays when the encryption key is found.
WorkerPanel – Displays worker table.

MasterControllerCallbacksImpl – Implementation of remote TaskMasterControllerCallbacks object for receiving notifications of Result Pairs and Worker (de)registrations.

Task Bag

TaskBagImpl – Implementation of remote TaskBag object. Maintains table for storing Worker notification objects and for storing Pairs.

Worker

Worker – Displays Worker window, obtains reference to Task Bag remote object using RMI, registers for notification (using callbacks) with the Task Bag.

TaskReadyCallbackImpl – Implementation of remote TaskReadyCallback object for receiving notifications of Task Pairs.

TerminateComputationCallbackImpl – Implementation of remote TerminateComputationCallback object for receiving notifications of end of computation.

The listing below can be used to define the Task Bag interface as well as the interfaces for the remote callback objects to be used by the Task Bag to notify Master and Worker processes of events (Note that these interfaces are mere suggestions and you are by no means strictly restricted to use them in verbatim).

```java
public interface TaskReadyCallback implements Remote {
    void taskReady() throws RemoteException;
};

public interface TerminateComputationCallback implements Remote {
    void terminateTask() throws RemoteException;
};
```
public interface MasterControllerCallbacks implements Remote {
    void taskStarted(string workerName, string task) throws RemoteException;
    void addWorker(string workerName, TerminateComputationCallback callback) throws RemoteException;
    void removeWorker(string workerName) throws RemoteException;
};

public interface TaskBag implements Remote {
    void placePairData(string key, string value) throws RemoteException;
    void placePairTask(string key, string value) throws RemoteException;
    void placePairResult(string key, string value, string workerName) throws RemoteException;
    string takePairData(string key) throws RemoteException;
    string takePairTask(string key, string workerName) throws RemoteException;
    string takePairResult(string key) throws RemoteException;
    void clear();
    void registerMasterCallbacks(MasterControllerCallbacks callback) throws RemoteException;
    void addWorker(string workerName, TaskReadyCallback callback) throws RemoteException;
    void removeWorker(string workerName) throws RemoteException;
};

It is important to make the Task Bag and Worker multithreaded where each thread would handle separate calls. This is necessary in systems where two processes act as both client and server, like in the case of using callbacks. Since the Task Bag must make a taskReady call to the Worker and the Worker must then make a takePair call to the Task Bag. The call to takePair will hang since the Task Bag is still waiting for the taskReady call to complete.

Notice there are three types of placePair and takePair methods each.

The placePairData method is used for Data Pairs. The placePairTask and takePairTask methods are used for Task Pairs. placePairResult is used for Result Pairs. A new Task Pair means that all Workers should be notified of new tasks. New Result Pairs require notification to the Master so the results can be obtained. The reason an extra workerName parameter is added to placePairTask and takePairTask is so that the Master can update its worker table when a worker starts a task and is finished with a task.

The clear method is provided to remove all Pairs from the Task Bag. The registerMasterCallbacks, addWorker, and removeWorker methods are provided for event notification registration purposes.

**Submitting the Work**

You will be required to demonstrate your programs. Please provide us with the following at the time of the demonstration):
• Printouts of your master and worker programs and of the Task Bag interface defined in Java RMI. If you defined other remote interfaces, for example for callbacks, include them too.

• Documentation explaining the following:
  
o the classes and objects representing the Task Bag (please don't design anything too complicated - this is an exercise on RMI, not data structures!)
  o the task that you did
  o how your task maps onto the Task Bag operations in both worker and master
  o a description of an approach to the synchronization problem which would be feasible with the system you used (Java RMI).

References


