BSE 3202 – Distributed Systems Development

Project 2: Using RPC in C (or C++) to implement a Task Bag

Introduction

This project consolidates the basic knowledge about RPC implementation in Chapters 4 and 5 and gives students experience in constructing programs that use Unix sockets with UDP. The attached notes on Sockets in Unix may be also be useful (socketnotes.pdf). You should use either C++ or ANSI standard C function prototypes. This exercise should be done in groups of four or six. You are required to demonstrate working programs and to submit well-commented code.

This project is loosely based on a model for parallel computation and consolidates the knowledge from Chapter 12 about synchronization in servers. The objective of this exercise is to build a simple toolkit that enables processes running in several workstations to carry out parts of a computation in parallel. The general idea is that a master process places sub-tasks of a computation in a 'Task Bag' and worker processes select 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 this exercise we shall implement the Task Bag as a Remote "Object" (using RPC in C or C++) and to use it as a basis for performing a parallel computation on several workstations.

Three roles are involved: The Task Bag, the master process, and the worker processes. The master and worker processes are clients of the Task Bag "object".

Preamble

Before working on the coursework itself, you should perform the following exercise, which is not examined. It involves compiling and running an existing sockets program in *cliserv_rpc.c*. Study it carefully. Like the subsequent programs you are asked to write for this exercise, it uses UDP (not TCP) sockets. You may borrow any code you feel necessary from this program, such as `MakeLocalSA`, `MakeDestSA`, `MakeReceiverSA`, `printSA` and use them as utilities in the following exercises. Also note the 'include' files needed and the function prototype for `gethostbyname`.

Warm-up exercise. The program *cliserv_rpc.c*, when given "client" as an argument repeatedly calls `sendto` with a dummy data argument of size `datasize` (given as another program argument), followed by `recvfrom` with a 4-byte data argument. When given "server" as an argument it repeatedly calls `recvfrom` with a data argument of size `datasize` (given as the second program argument), then sends a reply using sendto with a 4-byte data argument. Note that this server does not perform any useful processing upon the arriving 'request'. Nor does it do a lookup to identify a (dummy) procedure or 'method' to call. Use some sensible port number for the server, unique on the machine. Run the client and server, and test that they work together.

The Task Bag

The functionality of the Task Bag 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 will offer the operations `pairOut`, `pairIn` and `readPair` in its "interface". They are defined as follows:

`pairOut(key, value)`	causes a Pair (key, value) to be added to the Task Bag. The client process continues immediately
`pairIn(key) -> value`	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 until one is and then proceeds as before. If several matching Pairs are available, one is chosen arbitrarily
`readPair(key) -> value`	is the same as `pairIn(key)` except that the Pair remains in the Task Bag

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) parallel compilation of the modules of a program (parallel make); (ii) searching for files containing a particular text string; (iii) finding prime numbers (iv) matrix multiplication or (v) 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 modelling 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.

For the second example, we consider a program that multiplies two matrices *A* and *B*. In this program one master process sets up the multiplication tasks and collects the results, generated by one or more workers. Each worker repeatedly gets an element to calculate and puts the result in the Task Bag (for later collection by the master).

How the workers know which task to do next

In many computations, there is a collection of tasks, numbered *First* to *Last*. Each worker repeatedly carries out one (or a group) of the 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 puts in the first task and each worker in turn takes the Pair out, increments its value and puts it back.

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 matrix multiplication example, GRANULARITY = 1 and the worker calculates the row and column of the element to calculate from the value retrieved. e.g. the elements may be numbered in order across the rows.

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. (That is, it is not a set!)

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

```
pairOut("Primes", <a collection of primes>);
```

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

```
<a collection of primes> = pairIn("Primes")
```

In some applications, each worker needs to apply a different key to the results of its work. For example, in the matrix multiplication, each worker task consists of calculating one element of the result: the key of the result needs to indicate the row and column numbers of the element calculated. When a worker has calculated an element (row, column), it will specify the number of the row and column in the result e.g.:

```
sprintf(key, "Element%d%d", row, column);
pairOut(key,<the calculated result>);
```

Data for the workers

In some computations, the workers need data in order to perform their task. For example, in the matrix multiplication task, a worker needs row *i* of matrix *A* and column *j* of matrix *B* in order to multiply them together. This data is put in the Task Bag by the master and may be accessed by workers that need it. The master can put in the rows of matrix A and the columns of matrix B as follows:

```
pairOut("A1", <A's first row>);
pairOut("A2", <A's second row>);
// …
pairOut("B1", <B's first column>);
```

```
pairOut("B2", <B's second column>);
```

// …

In this example, many workers will require the same rows and columns, they therefore use the `readPair` operation rather than the `pairIn` operation. A worker may for example access a particular row of *A* and column of *B* as follows (in C!):

```
sprint(key, "A%d", row);
aRow = readPair(key);
sprintf(key, "B%d", column);
bRow = readPair(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.

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.

Synchronization of client operations

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

We suggest that you implement the Task Bag first so that the client 'polls' the Task Bag, repeating the request after a small time out, if the Task Bag tells it no pair is currently available. This is not an ideal approach, but you may describe a better approach and if you have time, implement it.

References

[Ahuja, Carriero and Gelertner 1986] Linda and Friends, Ahuja, S., Carriero, N. and Gelertner. D., IEEE Computer, August 1986. pp 26-34.

[Carriero and Gelertner 1989] How to write parallel programs: A guide to the perplexed, Carriero, N. and Gelertner. D., Computing Surveys, Vol. 21, No 3, Sept. 1989.

This assignment is due 4 weeks from the 20th of February, 2020.