STRATEGY 1 : PRUNING NODES BASED ON COST OF BEST-KNOWN SOLUTION

A tighter lower bound is computed for every node in the search tree inside the *decompose()* method of *TspTask* object. This is achieved summing the cost of the partial tour with all possible minimum-cost edges among the remaining cities left to be traversed. Based on this tighter lower bound, an attempt is made to prune the current node in the search tree. The current lower bound is compared to the upper bound value (cost of best-known solution) encapsulated in the *TspShared* object owned by the computer. This strategy is explained below.

```
Class tasks.tspTask

decompose() {
    if (lower bound >= upper bound) {
        Return an empty Result object with no sub-tasks and values. The ComputerProxy
        that reads the returned Result object understands from the null values that the
        node has been pruned. It also adds a null value to this node's argument slot in
        the Successor's Closure. These null values in the Closure are later ignored by the
        compose() method. Thus the pruning process is effective.
    } else {
        If max. recursion depth is reached, then use Strategy 2 explained below.
        Otherwise, decompose this task and return sub-tasks to the ComputerProxy
        object.
    }
}
```

STRATEGY 2 : SHARING THE COST OF BEST-KNOWN SOLUTION ACROSS COMPUTERS

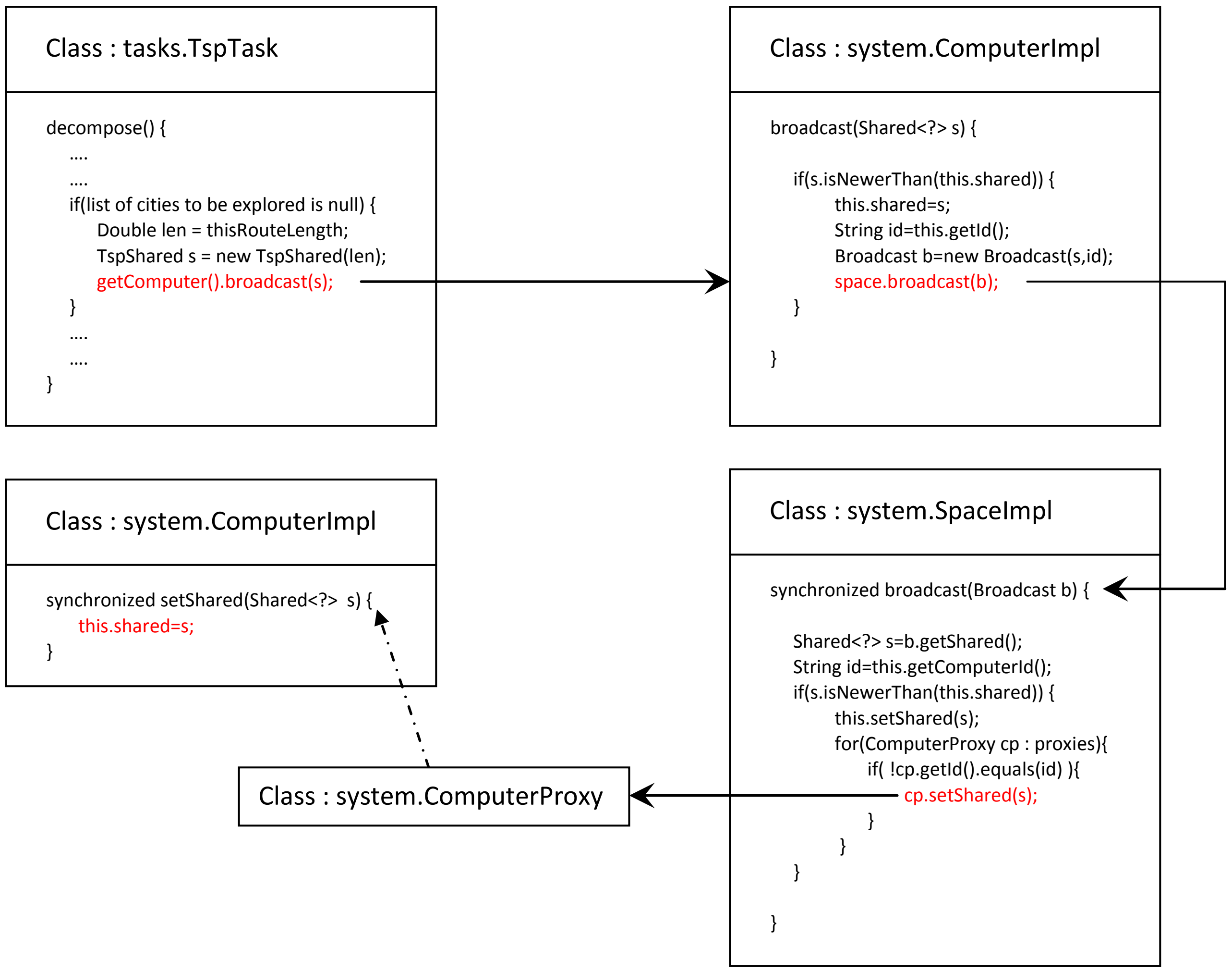
Step 1 : If the maximum permissible recursion depth has been reached in task decomposition, the *TspTask.decompose()* method uses depth-first search to locally traverse all possible routes in the remaining cities. For each leaf node visited during the depth-first search, a new upper bound is produced. At any point of time, the *TspTask* object is unaware if the global upper bound maintained in the compute space is better than this new upper bound. The *decompose()* method of the *TspTask* class passes a shared object containing the upper bound to the computer that is currently executing the task.

Step 2 : The computer compares the new shared object with its own copy to decide if the upper bound is indeed better. If yes, the new shared object and the computer ID are placed in a *Broadcast* object and sent to the compute space.

Step 3: The compute space unpacks the *Broadcast* object and compares the shared object with its own copy. If the new upper bound is indeed better, then the space pushes the updated shared object to each computer proxy except the proxy corresponding to the computer that initiated this broadcast. Note that the broadcast(..) method in class : system.SpaceImpl is synchronized to prevent lost updates when multiple computers broadcast upper-bound values concurrently.

Step 4 : Each proxy delegates the setShared(..) call to the *Computer* object that it encapsulates, thus propagating the new upper bound to the computer.

The figure below depicts steps 1-4 :


Class : tasks.TspTask
Class : system.ComputerImpl
Class : system.SpaceImpl
Class : system.ComputerImpl
Class : system.ComputerProxy

```
decompose() {
    ....
    ....
    if(list of cities to be explored is null) {
        Double len = thisRouteLength;
        TspShared s = new TspShared(len);
        getComputer().broadcast(s);
    }
    ....
    ....
}
```

```
broadcast(Shared<?> s) {

    if(s.isNewerThan(this.shared)) {
        this.shared=s;
        String id=this.getId();
        Broadcast b=new Broadcast(s,id);
        space.broadcast(b);
    }

}
```

```
synchronized broadcast(Broadcast b) {

    Shared<?> s=b.getShared();
    String id=this.getComputerId();
    if(s.isNewerThan(this.shared)) {
        this.setShared(s);
        for(ComputerProxy cp : proxies){
            if( !cp.getId().equals(id) ){
                cp.setShared(s);
            }
        }
    }

}
```

```
synchronized setShared(Shared<?>  s) {
    this.shared=s;
}
```