The Supervisor-Worker Pattern

Sebastian Fischmeister and Wolfgang Lugmayr
Distributed Systems Group
Argentinierstr. 8/184-1, A-1040 Vienna
Technical University of Vienna, Austria
{fischmeister,lugmayr}@infosys.tuwien.ac.at

Aug. 1999
revised June 2000

Abstract

Mobile agents and mobile computing have grown in importance recently. The Supervisor-Worker pattern is an architectural pattern that helps architects solving the problem of protecting the mobile agent from leakage and tampering. The fundamental Master-Slave Pattern is widespread and heavily used in traditional applications. The Supervisor-Worker pattern inherits many of the Master-Slave pattern's benefits. It also solves several of the security issues of mobile agents.

The Supervisor-Worker Pattern

Intent

The intent of this architectural pattern is to protect a mobile agent from a local attack. More specifically it protects the mobile agent from leakage and tampering[2]. While protecting the mobile agent the advantages of closely related patterns and the mobile agent paradigm are not lost.

Motivation

Scenario Consider this travel planning example: You live in Europe and want to stay one week in New York and you need a flight to New York, a room for one week and a flight back. An additional point is that the flight and the stay should be cheap though meet your needs. To book this trip the following components are needed: flight databases containing flights, airlines, reservations, and prices and hotel databases containing hotels, prices, and reservations.

To use the mobile agent paradigm two more software components are required. First, mobile agents that have the ability to move from host to host on their own volition. Second, the agent system which is executed on top of an operating system and provides all the infrastructure for the mobile agents. The agent system also executes the mobile agents. It is obvious that there are more than one agent systems needed to run an application.

In the mobile agent paradigm the solution to the scenario would look like this: you create an mobile agent, then define all the constraints (e.g. a range for the flight costs, duration of the stay, preferred airlines, etc.; for a detailed description see section), and initiate the agent; now the agent moves around in the net to find a flight and a room which suits you. It decides locally at the current hosts what to do next and only keeps the data which will be used in future (so, for example, expensive flights will not be carried).

The main drawback of this paradigm are the lack of security and also its technical feasibility. A lot of research has been done in this area but the remaining problems are untrusted and malicious hosts, which can attack mobile agents. Travel agencies could try to cheat, so their offers will be taken as the one which suits best. There are several ways this could be done, but all are done via tampering or leakage.

Leakage: the acquisition of information by unauthorized recipients.

Tampering: the unauthorized alteration of information (including programs).

For example the local agent system of the travel agency may modify the offers the agent has collected so far (changing the prices) so that then its offer will be the cheapest one. Or the local agent system changes the list of travel agencies the agent is going to visit too, so the chances that its offer will be the best one increases. Both attacks are tampering - the local agent system modifies the agent's data.

Another example is that the local agent system scans the agent for the lowest price found so far and then makes a better offer, so that at least at this moment its offer is the best one. Or the local agent system could try to get the list of travel agencies that will be visited too, then ask the prices there before it makes its final offer. Both attacks are leakage - the local agent system spies out data and misuses it by changing itself.

The solution for this problem is that the agent does not take such information with it. Because the less confidential information the mobile agent takes with it the less it has to trust the hosts. A agent without confidential information is less likely to be attacked, especially for leakage and tampering.

The underlying pattern is the Master-Slave pattern [1]. Buschmann described that for reasons of fault-tolerance, efficiency, "separation of concerns", exchangeability and the possibility of concurrent execution you can divide a whole task into several smaller ones which are carried out by several slaves in parallel and controlled by one master. The key point from the Master-Slave pattern, for this paper, is the separation of concerns - separating the master from the slaves. The master will hold all information and the slaves visit the travel agencies. So the core design of the Master-Slave pattern will be reused.

Although the Master-Slave pattern has many benefits, it does not introduce mechanisms for collecting information and controlling workers. The solution given above that the agent does not take confidential information with it needs these mechanisms. Based on the Master-Slave pattern and the separation of concerns, one part will just collect information and the other part will control it. So you need a different design from the Master-Slave pattern. To ensure that all the information is processed in the right way and actions are done at the right time a central knowledge-base and a central management unit are needed. This central knowledge-base and the central management unit will be protected from malicious hosts. The Supervisor-Worker pattern, which is built on top of the Master-Slave pattern, provides this infrastructure.

Utilizing Supervisor-Worker Pattern

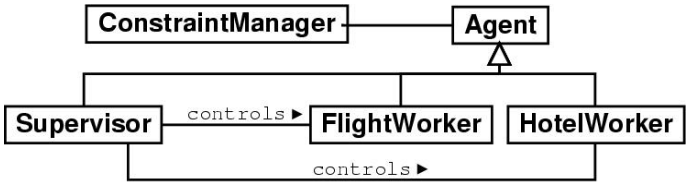

Figure 1: Participants in the Supervisor-Worker Pattern (Scenario)

In our travel planning example utilizing the Supervisor-Worker pattern the structure would be designed like Figure 1. You create the *supervisor*, who controls the *flightworker* and the *hotelworker*. So the *supervisor* is the managing component and is decoupled from the two workers. These workers are independent and move around in the net. The *flightworker* tries to find a flight to New York and back which meets the specification and the *hotelworker* tries to find a hotel in New York. Their tasks are defined by the *supervisor* which coordinates them; it keeps track of all dependencies and constraints. For example that the date of the arrival of the flight is the same as the beginning of the reservation of the room. In order not to lose advantages of the mobile agent paradigm the

supervisor is a mobile agent itself. Now tampering and leakage are prevented because the workers do not have any interesting information. After each hop they send back their data to the *supervisor* - the *flightworker* sends back the flights it has found and the *hotelworker* sends back the hotels found in the local database. The data will be sent back as a message and to protect it simple cryptographic algorithms can be used to protect it from leakage and tampering. The *supervisor* itself only resides on trusted hosts so it will not be attacked.

This design solves the problems of leakage and tampering. Neither can a malicious agent system spy out any useful information and misuse it, nor does it make sense to modify the mobile agent, because the supervisor coordinates everything and if necessary disposes the workers and creates new ones. Besides this many benefits of the Master-Slave pattern can still be used, for example the tasks can be split and executed concurrently.

Applicability

- The Supervisor-Worker pattern should be used when you want to protect a mobile agent from a malicious host concerning tampering and leakage.
- The Supervisor-Worker pattern should be used when a task that can be broken down and serviced by multiple agents.

Structure

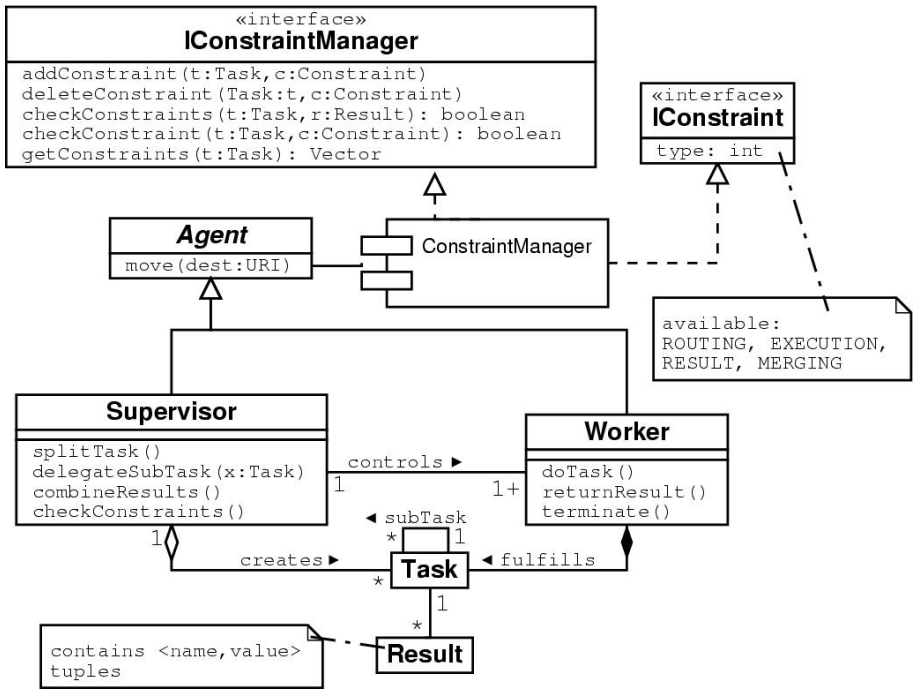

Figure 2: Structure of the Supervisor-Worker Pattern

Participants

The whole class diagram consists of two parts: the mobile part and the constraint managing part. The mobile agent part is described in the next paragraphs. The detailed description of the constraint managing part is beyond the scope of this paper. Only the interfaces are defined, so the method calls are still transparent.

Mobile Part

The mobile part contains the classes and methods (according to Figure 2) needed for performing an action, fulfilling the whole task and returning a result.

Agent implements all basic features needed for mobility. Furthermore every agent has its own *ConstraintManager*; when the task is divided, only some constraints remain important for each subtask (these constraints will be carried with the worker), but the division also carries new constraints (these constraints will be stored by the supervisor).

Supervisor implements all features concerning task division, worker-control, and report merging. It develops strategies for completing the whole task, creates sub-tasks and stores additional information about them (sub-task worker relation, merge dependent constraints, etc.)

Worker implements all methods needed for accepting and fulfilling tasks and sending reports to the supervisor. It also provides the execution environment for tasks, in which they can be executed on each host.

Task is the basic abstraction of a job that has to be done. The results are stored within the task-object, and when sub-tasks are merged, the results for all sub-tasks can be freed. The dependencies between the sub-tasks are stored by the supervisor (eg. it is useless to book a flight, when you do not have a hotel).

Result is a class which stores tuples. These tuples have the following structure: < *name: String, value: Object* >. For each task there can be several results (several workers could carry out the same subtask and return different results).

Constraint Managing Part

The constraint managing part is an encapsulated component which implements the following interfaces:

IConstraintManager defines all methods needed to add, modify, delete and check constraints.

IConstraint , simplified for this paper, contains a variable to determine which type of constraint it is. This is one possible solution and it is adapted for this paper.

Constraints

As mentioned several times there are different kinds of constraints to be defined (also listed in Figure 2). The four constraints concern: routing, execution, results and tasks.

The first type (*routing constraints)* defines variables which are important for the moving phase of the mobile agents. These could be a limitation like a domain restriction. You could limit the movement to certain domains, eg. you want the agents just to move in your intranet; you could limit the number of hosts the agents (more important for the workers) are allowed to visit; you could limit the number of retries so your agents do not get stuck trying to move to a host which is always down. These constraints have to be checked before anything concerning mobility is done. So if the worker found a new host where new information could be found or where the task has to be executed too, it would have to check if the constraints allow it to move there, otherwise it will have to choose another one.

The second type (*execution constraints*) defines the environment of the agent system to which the workers move. These could be a limitation of the agent system, so your workers check if it is the correct one, or one which is up-to-date (in combination with the version number) and provides the special APIs the workers need. The constraints could define which hardware resources should be available and furthermore which minimal resources the worker needs (eg. the amount of memory storage). Also software constraints could be specified, if a specific version of the database-access software is required to fulfill the task or an LDAP-service. In contrast to the first type these constraints are variable and a lot more specific for each task. This type has to be checked before every execution of the task because if the local agent system does not provide all the needed APIs or does not have enough memory, the task could not be finished either.

The third type (*result constraints*) is even more variable than the second one. Constraints of this type define task specific conditions like: how much money should the trip cost, how long do I want to stay, which are my

preferred airlines, etc. They are coded by the user himself and often are the same for all sub-tasks. If the result of the action the worker did does not meet them, the actions should be undone. So when the worker found flights but they are too expensive or they do not fit into the time-frame then it does not make any sense to report them to the supervisor or carry the results further.

The last type (*task constraints*) defines the relations between the several sub-tasks. These are not influenced by the user because they are generated by the supervisor itself. So, for example, when a task is divided into two sub-tasks and one sub-task cannot be fulfilled, then a constraint could say the other sub-task is useless and should be canceled. These are the only constraints which are stored by the supervisor. All other constraints are stored in the workers and they check the constraints before each move, before each execution, and before they send back any result. The task constraints have to be checked before the results of the sub-tasks are merged and eventually a new worker has to be created for undoing a sub-task which has been fulfilled, because another sub-task could not be fulfilled.

Collaborations

Figure 3 shows the Supervisor-Worker pattern in a use-case diagram. The user has a task which has to be fulfilled, so he defines the task and the constraints for it. In order to fulfill a task you have to set actions, fulfill constraints and return the result. Actions are done by the workers at each host they visit. A ConstraintManager is responsible that the defined criteria are met and finally the Supervisor takes the responsibility for the whole task and only it decides when the whole task is finished and the results are returned.

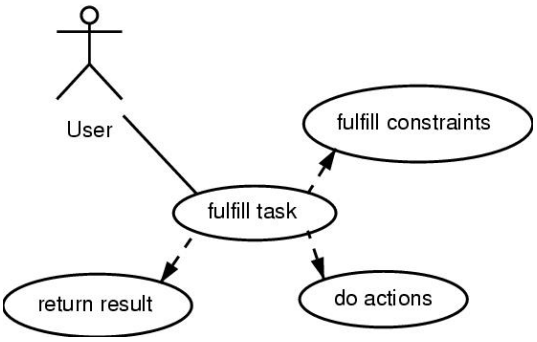

Figure 3: Use-Case diagram of the problem

The collaboration between the participants in the Supervisor-Worker pattern is depicted in Figure 4. Method calls for checking constraints are not shown to avoid clutter (constraints are checked before every method call), but they are given in the description.

1. The user creates the supervisor and delegates the special tasks.
2. The supervisor splits the tasks into different sub-tasks and moves (obeying its routing constraints) to a host near the area where the workers will work.
3. The supervisor creates workers and delegates the sub-tasks (obeying its execution constraints).
4. The workers jump to various hosts (obeying their routing constraints) and while finishing their task (obeying their execution constraints) send reports to the supervisor (obeying their result constraints).
5. The supervisor merges the results (obeying its merging constraints) and will eventually start from point 2, if the task is not finished.
6. The supervisor reports the result to the user.

In the activity diagram (Figure 5) you have the activities described for each component.

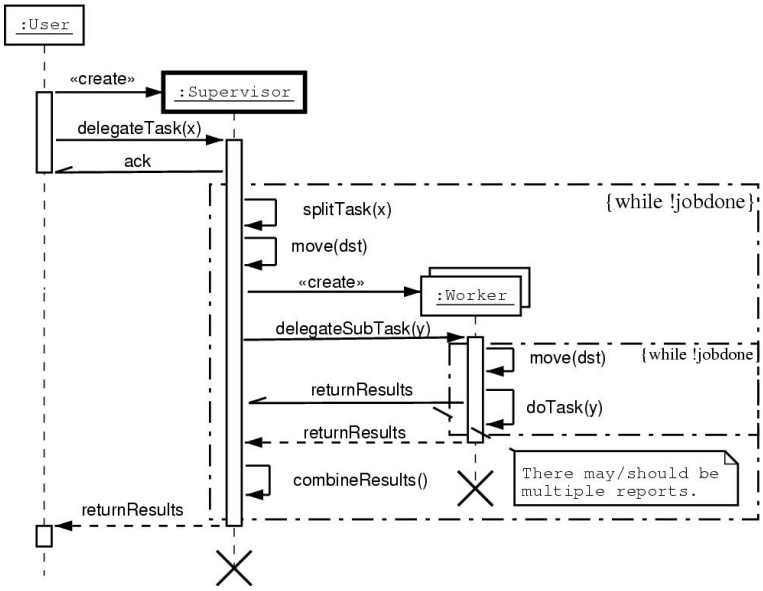

Figure 4: Interaction diagram of the Supervisor-Worker pattern.

Consequences

Benefits The benefits that directly come from the Supervisor-Worker pattern are:

- *Security* - There is no need for the workers to carry all information found so far with them, so the likelihood of a modification attack is reduced and other threats like tampering and leakage are prevented. Furthermore cloning is no longer necessary.
- *Maintainability* - The decoupling of the components - *Supervisor*, *Worker*, and *ConstraintManager* - enhances the maintainability of the whole design. Each component can be exchanged without touching the others' code or interfaces.
- *Simplicity* - The simple and straight forward design of this pattern improves its usability and it can easily incorporated in frameworks or combined with other patterns.

Side-Effects Besides the consequences that directly come from the pattern there are many positive sides-effects that come from closely related patterns. These sides-effects are:

- *Fault tolerance* - If one worker fails to return within a given deadline the supervisor could start another one or redefine the subtasks and then start a new one.
- *Concurrent execution* - There can be many workers working for one supervisor and each of these workers operates independently. This speeds up collection results of the sub tasks and so speeds up the whole process.

Liabilities The are also some liabilities and trade-offs which have to be considered.

- *Cryptography* - The security relies on safe transmission of the messages which are sent back to the supervisor. Nevertheless the local agent system is interested in transmitting its data safely. So there have to be mechanisms that prevent these messages from being read by outsiders.
- *Complexity* - Division of tasks is not always simple and depends very much on the situation. Therefore creating and handling the constraints can get complex.

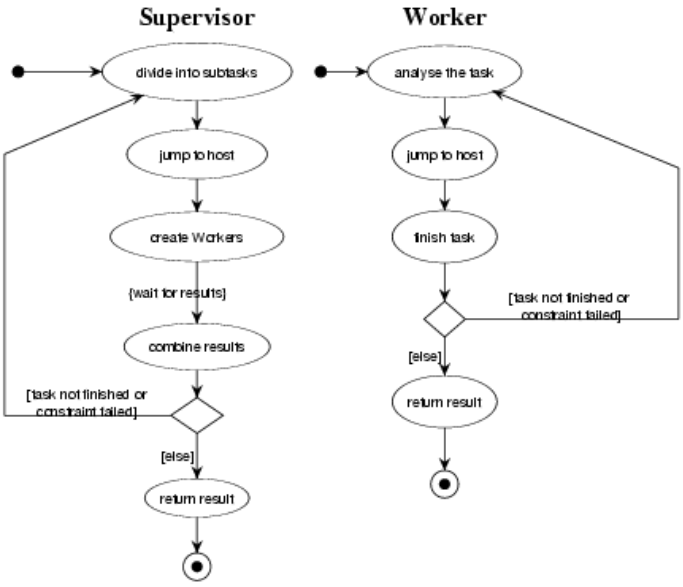

Figure 5: Activity diagram for the supervisor and a worker

- *Overhead* - As there is more than just one agent roaming the net (at least one worker and one supervisor) the overhead of network transmission and managing these multiple agents has grown.
- *Requirements* - Trusted hosts are needed so the supervisor can reside there. In the worst case this will be your local computer and the supervisor never moves to another host.

Known Use

This pattern will be implemented in the Gypsy Project[3, 5]. The Gypsy Project provides the main facilities needed to implement and test this pattern. For the test scenario the agent will collect system information from various hosts and then do additional result based actions. The constraints in this scenario will be routing constraints. Every worker gets a routing constraint which tells him which computer it should check. The results then are sent back to the supervisor and in the end it will create a short report about all computers.

Sample Code

Here is a sample implementation for the supervisor and a worker. Some parts are simplified and the interfaces are also not given.

First of all the *doTask()* method will be called in the supervisor object. Then the supervisor will be transfered to the first host where it starts splitting the whole task and setting up the sub-tasks. While splitting the task it creates several workers which will send back their results. The supervisor waits for these results and finally merges them. If the task is not finished then the whole process will start over again.

```
public class Supervisor extends Agent {
    private ConstraintManager cm;
    private Vector tasks;
    private Vector sentWorkerIds;

    protected void doTask() {
        do {
            getCurrentHost(). transfer ( this );
            splitTasks ();
            waitForResults ();
```

```
            mergeResults();
        } while (! supertask.finished());
        sendResultsHome();
    }

    private void splitTasks() {
        // 1. apply strategy to devide the task
        // 2. refine constraints for the subtasks
        for (int i = 0; i < tasks.size(); i++) {
            Worker w = new Worker(subtask, constraints);
            sentWorkerIds.add(w.getId());
            getCurrentHost(). transfer(w);
        }
    }
}
```

When the supervisor splits the tasks and assigns the worker-task relation, the worker will immediately start planning its trip and will be moved to another host. There the local agent system instantiates the worker and calls its *doTask()* method. In this method the worker runs its task and then returns the results. Finally it will move on to the next host.

```
public class Worker extends Agent {
    private ConstraintManager cm;
    private Task task;

    Worker(Task t) { task = t; }

    protected void doTask() {
        do {
            task.run();
            addResult(task.getResult());
            getCurrentHost(). transfer(this);
        } while (! task.finished());
    }

    private void addResult(Result r) {
        if (cm.checkConstraints(task, r))
            sendResultToSupervisor(r);
    }
}
```

Related Patterns

The patterns which related to this one are the Master-Slave[1, 4]. As stated in section the core design of this pattern is incorporated into the Supervisor-Worker pattern. Both main components of the Master-Slave pattern also exist in this pattern: the master, which is equivalent to the supervisor, and the slave, which is equivalent to the worker.

References

[1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons Ltd, Chichester, UK, 1996.

[2] George Coulouris, Jean Dollimore, and Tim Kingberg. *Distributed Systems: Concepts and Design*. Queen Mary and Westfield College, University of London, 1996.

[3] Mehdi Jazayeri and Wolfgang Lugmary. Gypsy: A Component-based Approach to Mobile Agent Systems. Technical Report TUV-1841-99-09, Distributed Systems Group, Technical University of Vienna, Argentinierstrasse 8, A-1040 Vienna, Austria, 1999.

[4] Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley Longman, 1998.

[5] Wolfgang Lugmary. The Gypsy Environment. World Wide Web Site, 1998. `http://www.infosys.tuwien.ac.at/Gypsy/`.