STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **XLVI**, Number 2, 2001

DESIGNING A FAULT–TOLERANT JINI COMPUTE SERVER

IOAN LAZĂR

Abstract. Java-based tuplespaces provide a simple infrastructure for scientific distributed computing. There are several classes of problems that are not efficiently solvable in JavaSpaces model of computation while efficiently solvable in other tuplespace implementation. JavaSpaces can be used for high performance computing if viewed less strictly in the heritage of Linda and more as a platform-neutral code delivery mechanism.

This paper presents an early design of JavaSpaces compute server. We describe how we use mobile co-ordination and agent wills to provide fault-tolerance in Jini based compute servers. Preliminary experimental results show performance gains made when mobile co-ordination is used.

In this paper we apply the mobile co-ordination and agent wills [12] to provide fault-tolerance in compute servers based on tuple spaces.

The first section the tuple space paradigm and the Linda model for parallel computation. The next section provides an overview of Jini and JavaSpaces [14]. The third section describes in detail our design and then we present our conclusions.

1. The Tuple Space Paradigm

Linda is a well known co-ordination model [3]. The fundamental concepts of Linda are tuples, templates and tuple spaces.

A *tuple* is an ordered collection of fields. Each field has a type and a value associated to it. A field with both a value and a type is known as an *actual*. The tuple $< 1989_{int},\ \text{“Linda”}_{string},\ 1.0_{real} >$ is a tuple containing three fields, with the type of the field shown as a subscript of the value.

Tuples are placed into tuple spaces and are removed from tuple spaces using an associative matching process.

A *template* is similar to a tuple except the fields do not need to have values associated to them, but all fields must have a type. A field that has only a type and no value is known as a *formal*. A template is a tuple which can have formals. A template *matches* a tuple if they have the same number of fields, and

2000 *Mathematics Subject Classification.* 65Y05, 68Q85.

1998 *CR Categories and Descriptors.* F.1.2 [**Computation by Abstract Devices**]: Modes of Computation – *Parallelism and concurrency*; G.4 [**Mathematics of Computing**]: Mathematical Software – D.1.3 [**Programming Techniques**]: Concurrent Programming – E.1 [**Data**]: Data Structures – Distributed data structures.

all the actuals in the template match the actual in the tuple and the formals in the template matches the type of the corresponding actual in the tuple. The templates $< |\Box_{int}, \text{“}Linda\text{”}_{string}, 1.0_{real}| >$, and $< |1989_{int}, \text{“}Linda\text{”}_{string}, 1.0_{real}| >$ will match the tuple $< 1989_{int}, \text{“}Linda\text{”}_{string}, 1.0_{real} >$. ($\Box$ in a template is used to indicate formal fields.)

A *tuple space* is a logical shared associative memory that is used to store tuples. A tuple space implements a *bag* or a *multi-set* (the same tuple may be present more than once and there is no ordering of the tuples in a tuple space).

1.1. **Basic Tuple Space Operations and Matching.** Tuples are inserted into tuple spaces. In order to retrieve a tuple an associative match is performed between a template and the tuples in the tuple spaces. The main primitives (operations) on tuple spaces are:

***out(tuple)*:** Place the *tuple* into the tuple space.

***in(template): tuple*:** Removes a *tuple* from the tuple space. The *tuple* removed is associatively matched using the template and the *tuple* is returned to the calling process. If not tuple that matches exists then the calling process is blocked until one becomes available.

***rd(template): tuple*:** This primitive is identical to *in* except that the matched tuple is not removed from the tuple space, and a copy is returned to the calling process.

***eval(active tuple)*:** The *active tuple* contains one or more functions, evaluated in parallel with each other and the calling process. When all the functions have terminated, a tuple is placed into the tuple space with the results of the functions as its elements.

1.2. **Multiple rd problem.** The *rd* operation returns one arbitrary matching tuple from all tuples matching a given template. If a process wants to iterate over the list of all tuples matching a given template, no atomic Linda operation is provided in order to do this.

One solution presented in [10] is adding another operation *copy-collect* to the Linda operations. This operation reads all matching tuples in the tuple space and copies them into another tuple space. Therefore this extension of the tuple space model needs multiple addressable tuple spaces [3].

A similar operation *collect* is added to solve the equivalent multiple *in* problem, which moves all matching tuples from one tuple space to another tuple space.

Another solution is to change the application using the tuple space instead. One way is to include an index field in the tuples and iterate over that field when *rd* operations.

1.3. **Predicate Operations.** Linda includes predicate variants of the two operations *in* and *rd*, named *inp* and *rdp*. These are non-blocking versions of the

operations, meaning that if no tuple matches the template provided these operations return the boolean value *false* (or some other value) indicating that no matching tuple was found, and do not block.

In general, the non-blocking Linda operations *rdp* and *inp* cause a problem because they inspect the "present" state of a tuple space, and one could argue that they are inappropriate in a distributed environment, where "the most recent operation" is not defined [7].

1.4. **Fault-tolerant tuple spaces.** Early tuple space based languages suffered from poor fault agent (program, process) fault tolerance. Later, many systems have used transactions to provide fault-tolerant Linda primitives: PLinda [6], JavaSpaces [14] and TSpaces [16]. More recently Rowstron [12] proposed a new technique *mobile coordination* in order to provide fault tolerant tuple-space based co-ordination.

Transactions. Most implementations which use transactions add two new primitives which are *start* and *commit* [12]. The *start* primitive causes the server managing the tuple spaces to retain all copies being removed and to hold all copies being inserted by the agent (program, process) which performed the *start*. When the agent execute the *commit* operation, the tuples inserted under the transaction are actually placed in the tuple space and the tuples deleted are actually discarded. There are many problems with this approach [12]: altering the semantics of the co-ordination language and how can decide the server if the agent that performed a *start* operation is alive?

As an alternative to this traditional approach using transactions the mobile co-ordination approach solves the problems mentioned above.

Mobile Co-ordination. In this approach the co-ordination primitives are moved on the server which store the tuple space. If all the coordination *primitives* reach the server before any are executed then the entire operation of the agent will be executed.

An overview of this framework from an application developer perspective is as follows (see Figure 1). The agent who whish to execute an operation composed by many primitives *in* and *out*, encapsulate these primitives into the function *coordination*. This function will be executed by the tuple-space server. In order to migrate this code to the tuple-server, the application developer creates a class that implements the *MobileCoordination* interface (extended from the interface *Serializable*). After that, the agent calls the tuple-space operation *executeSafe* which returns a tuple.

An *agent will* is a set of tuple space access primitives that are executed when the tuple space server managing the tuple spaces decides that an agent owning the will has failed.

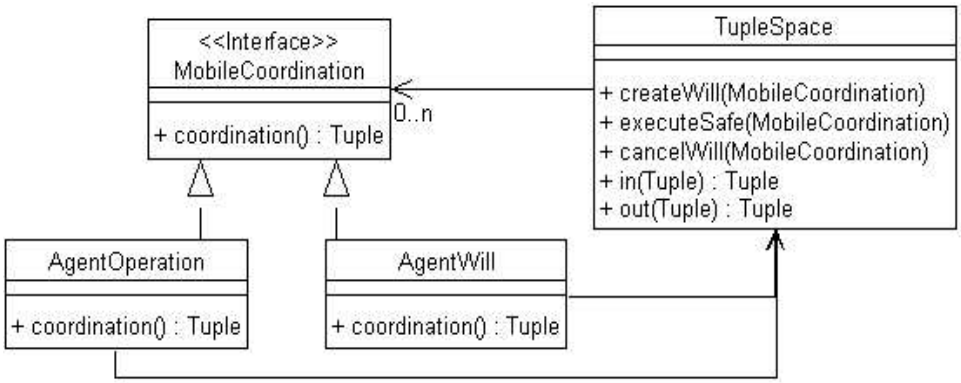

FIGURE 1. Mobile coordination class diagram

The following lines of code shows how an agent can take safely a tuple from a tuple space:

```
AgentOperation inOperation = new AgentOperation();
AgentWill inWill = new AgentWill(inOperation);
tuplespace.createWill(inWill);
tuple = tuplespace.executeSafe(opIn);
tuplespace.cancelWill(inWill);
```

where

```
class AgentOperation implements MobileCoordination {
  Tuple t;
  Tuple coordination() { //operations executed on server
    t = tuplespace.in(template);
    return t;
  }
}
class AgentWill implements MobileCoordination {
  Tuple t = null;
  AgentWill(AgentOperation inOperation) {
    this.tuple = inOperation.tuple;
  }
  Tuple coordination() { //operations executed on server
    tuplespace.out(tuple);
    return null;
  }
}
```

```
class TupleSpace {
  Tuple executeSafe(MobileCoordination agent) {
    return agent.coordination();
  }
}
```

This fault-tolerant mechanism is working if the `AgentOperation` and `AgentWill` objects are migrated to the tuple space server and their `coordination` methods are executed there.

1.5. **Tuple Space Usage in Parallel Computing.** The tuple space paradigm provides a mechanism for communication between processes in a distributed system. The abilities provided by a tuple space are for processes to *share data* and *coordinate events* in a distributed environment.

The data sharing of a tuple space is simply achieved by having several processes accesing a tuple space as a global shared memory.

The event coordination ability of a tuple space is achieved using a tuple to simulate a semaphore by having clients insert tuples into a tuple space and withdraw tuples from a tuple space. A single-element tuple is functionally equivalent to a semaphore [4]. The coordination feature of the tuple space is illustrated by implementing a counting n semaphore:

begin
  $in(t_{\text{semaphore}})$
    //critical region
  $out(t_{\text{semaphore}})$
end

where the initial value n can be obtained by n repetitions of $out(t_{semaphore})$.

There are two types of coordination (or synchronization): mutual exclusion and conditional synchronization. Mutual exclusion is exemplified in the pseudo-program above. Conditional synchronization – waiting for an event to occur, is also an integral part of Linda. This is achieved by blocking operations – waiting until a tuple is present.

2. Design overview of Jini and JavaSpaces

The Jini technology [15] is a runtime infrastructure that resides on the network and provides mechanisms to enable addition, removal, discovery and access of services. Jini enable building and deploying distributed systems that are organized as a *federation of services*. A *service* is an entity capable of performing some function. Services advertise their capabilities via a look up server. The primary function of lookup servers is to assist Jini enabled clients to discover and access services.

out(...)	`write(Entry entry, ...)`
in(...)	`take(Entry template, ...)`
rd(...)	`read(Entry template, ...)`
eval(...)	not included
inp(...)	`takeIfExists(Entry template, ...)`
rdp(...)	`readIfExists(Entry template, ...)`
not included	`write(Entry, ...)`

TABLE 1. Linda and JavaSpaces operations and terminology

JavaSpaces is a Java implementation of a tuple-based system [14], and is provided as a service based on Jini technology.

The tuple space is represented by the `JavaSpace` interface, and the Linda tuples and templates are represented by the `Entry` marker interface.

The `JavaSpace` operations `read`, `take` and `write` correspond to the *rd*, *in* and *out* Linda primitives. All these operations are synchronous (blocking operations).

There are also two asynchronous (non-blocking) operations `readIfExists` and `takeIfExist`. These types of asynchronous operations correspond to some Linda primitive extensions [11].

In order to provide fault tolerance for tuple space primitives, the `read`, `take` and `write` operations can be performed as part of a transaction or not. The description of the basic primitives include descriptions of how they interact with transactions. This fact increases the complexity of the implementation, and makes it from a simple model into a complex one.

JavaSpaces does not introduce any new concept at the Linda model level. JavaSpaces objects (`Entry` derived objects) can be introduced in a space but there they are not active objects. The `JavaSpace` interface does not have any operation corresponding to Linda *eval* primitive.

3. A Framework for Adaptive Master-Worker Parallelism

Master-worker parallelism is a widely used form of parallel application programming [13, 2]. It is conceptually very simple and involves dividing a problem into a smaller number of independent work units which can be distributed to remote worker processes for computation in parallel. A single master process centrally controls both the distribution of work units to worker processes and the returned of computed results back to the master process. The method of maintaining a collection of work units is referred as work queue or task farm scheduling.

There are many opportunities for running distributed running applications [13]. We choose here the Jini [15] environment for the master and worker processes, and JavaSpaces for task scheduling.

A typical space-based compute server works as described below:

- A task is an entry that both describes the specific of the task and contains methods that performs the necessary computations.
- Worker processes monitor a space, take tasks as they become available, compute them, and then write their results back to the space.
- Results are entries that contain data from computation's output.

Spaced-based computer servers have the following nice properties:

Scalability: the more worker processes there are, the faster the tasks will be computed. Workers can be added or removed at run time and the computation will continue as long as there is at least one worker to compute tasks.

Load balancing: workers running on slower CPU will compute their tasks slowly (and thus complete fewer tasks) than those running on faster CPU.

Low coupling: the master and the workers are uncoupled. The workers do not have to know anything about the master and the specific of the task - they just compute them and return results to the space.

3.1. **Basic abstractions.**

Tasks. Our spaces hold tasks. The task is considered an *active entity* of the space if it is not completed.

The `Task` class implements the Jini `Entry` interface in order to be JavaSpaces compatible. Also, the `Task` class defines a method `compute()` that is overridden by user-defined tasks and a method `execute()` called by the `Worker` processes. The `compute()` method should be an abstract method but in order the `Task` to be a template for retrieving tuples from the JavaSpaces spaces, the `Task` must be a concrete class.

The master process writes `Task` instances (`done` = `False`) into spaces and the workers take tasks `execute` them, and then return the completed tasks back into the space (`done` = `True`).

Master and Worker Processes. Master and worker processes use Jini services for JavaSpaces and distributed transactions. In the current implementation of the framework both uses a `TransactionManager` and a `JavaSpace` Jini services. This services are available through some methods of `AbstractProcess` class.

The abstract methods `generateTasks()` and `collectResults()` of the class `GenericMaster` are defined in concrete master processes.

A worker continually looks for tasks, takes it from a space, computes it and writes the result back in a space. The significant running code for the worker process is:

```
work() {
  for ( ; ; ) {  //looks continually for tasks
    Task task = taskReader.takeTask();  //take (safe) a task
```

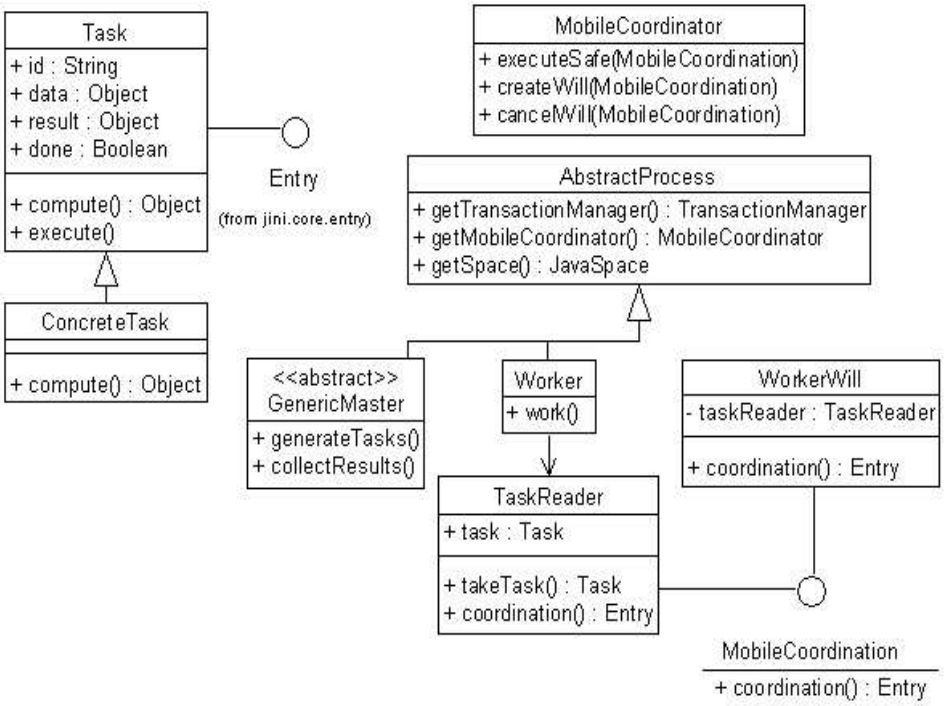

FIGURE 2. A Master-Worker Jini Framework

```
      task.execute();                    //execute, and then
      getSpace().write(task);        //write the task
    }                                    //(completed) back to the space
  }
```

where `taskReader` is an instance of the class `TaskReader`.

The next subsection discusses the rest of the design (our fault–tolerant decisions).

3.2. **Fault Tolerance.** The worker process uses mobile co-ordination and agent wills in order to provide a fault tolerant solution in our compute framework.

The `TaskReader` class encapsulates a safe Linda *in* operation on a JavaSpaces space. The `takeTask` method returns a `Task` from the space and it is a blocking operation:

```
1.  Task takeTask() {
2.    WorkerWill will = new WorkerWill(this);
```

```
3.    mobileCoordinator.createWill(will);
4.    Task task = mobileCoordinator.executeSafe(this);
5.    mobileCoordinator.cancelWill(will);
6.  }
```

The `mobileCoordinator` in the above code is a proxy of a Jini service running in the same Java virtual machine as the JavaSpaces service. (We have made some minor modifications in the `com.sun.jini.outrigger.SpaceProxy` class, but not altering the JavaSpaces services.) Line 4 causes the `TaskReader` object to be migrated to the server, and the result is returned.

The `MobileCoordinator` server executes the following:

```
Entry executeSafe(MobileCoordination mc) {
  return mc.coordination();
}
```

which is the `TaskReader`'s code:

```
Entry coordination() {
  this.task = space.take(new Task(), null, Long.MAX_VALUE)
  return task;
}
```

but executed into the server.

Now, if the worker dies while executing the task, then the server can restore the task back into the space since the `Worker` *will*, encapsulated into the class `WrokerWill`, is:

```
Entry coordination() {
  if (taskReader.task != null)
   space.write(taskReader.task, null, Long.MAX_VALUE);
  return null;
}
```

where the `taskReader` is a copy (stored in the server) of the `TaskReader` object.

4. Conclusions

In this paper we have demonstrated how the concepts of mobile co-ordination can be used to provide fault–tolerant Jini compute servers.

This paper describes the use of a single tuple space server. In real implementations, multiple servers must be used. A comparison between the transactions model used in Jini and also a multiple fault–tolerant tuple space servers is an area under consideration.

We also will investigate the proposed design related to some computational problems – linear and nonlinear solvers.

References

[1] N. Carriero, E. Freeman, G. Gelernter, D. Kaminsky, *Adaptive parallelism and Piranha*, IEEE Computer, 28(1):40-49, 1995.
[2] N. Carriero, G. Gelernter, *How to write parallel programs: a first course*, MIT Press, Cambridge, 1990.
[3] N. Carriero, D. Gelernter, Linda in context, *Communication of the ACM*, 32(4):444–458, 1989.
[4] D. Gelernter, Generative Communication in Linda, *ACM Transactions on Programming Languages and Systems*, 7(1), 1985.
[5] E. Freeman, S. Hupfer, K. Arnold, *JavaSpaces Principles, Patterns and Practice*, Addison Wesley, 1999.
[6] K. Jeong, D. Shasha, Persistent Linda 2: a transaction/checkpointing approach to fault-tolerant Linda, in *Proc* 13^{th} *Symposium on Fault-Tolerant Distributed Systems*, 1994.
[7] J.E. Larsen, J.H. Spring, *GLOBE: Global Object Exchange*, Candidatus Scientiarum in Computer Science Thesis, Univ. Copenhagen, 1999.
[8] M.S. Noble, S. Zlateva, *Distributed Scientific Computation with JavaSpaces?*, Boston University, Technical Report CN01-34, 2001.
[9] A. Rowstron, Using Agent Wills to Provide Fault-tollerance in Distributed Shared Memory Systems, 8^{th} *EUROMICRO Workshop on Parallel and Distributed Processing*, Rhodos, Greece, IEEE Press, pp. 317-324, 2000.
[10] A. Rowstron, A.M. Wood, Solving the Linda multiple rd problem, *Coordination Languages and Models, Proc. Coordination '96*, eds. P. Ciancarini, C. Hankin, Springer-Verlag, LNCS 1061, 1996, pp. 357–367.
[11] A. Rowstron, Using asyncronous tuple space access primitives (BONITA primitives) for process co-ordination, *Coordination Languages and Models*, eds. D. Garlan, D. Le Metayer, Springer-Verlag LNCS 1282, pp. 426–429, 1997.
[12] A. Rowstron, Mobile Co-ordination: Providing fault tolerance in tuple space based co- ordination languages, *Coordination Languages and Models, Coordination '99*, eds. P. Ciancarini, P. Wolf, Springer-Verlag, LNCS 1594, 1999, pp. 196–210.
[13] G. Shao, *Adaptive Scheduling of Master/Worker Applications on Distributed Computational Resources*, Phd Thesis, Univ. California, San Diego, 2001.
[14] Sun Microsystems, *Jini Specifications*, Available from Sun Microsystems WWW Site (http://java.sun.com/products/javaspaces/), 1998.
[15] Sun Microsystems, Jini Specifications, Available from Sun Microsystems WWW Site (http://www.sun.com/jini/specs/), 2000.
[16] P. Wyckoff, S. McLaughry, T. Lehman, D. Ford, TSpaces, *IBM System Journal*, 1998. 1994.

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, RO-3400 Cluj-Napoca, Romania
E-mail address: ilazar@cs.ubbcluj.ro