A Javaspace-based Framework for Efficient Fault-Tolerant Master-Worker Distributed Applications

Virginie Galtier
SUPELEC - UMI-2958
France
Email: Virginie.Galtier@supelec.fr

Constantinos Makassikis
SUPELEC - UMI-2958 &
AlGorille INRIA Project Team
France
Email: Constantinos.Makassikis@supelec.fr

Stéphane Vialle
SUPELEC - UMI-2958 &
AlGorille INRIA Project Team
France
Email: Stephane.Vialle@supelec.fr

***Abstract*—We propose a framework built around a JavaSpace to ease the development of bag-of-tasks applications. The framework may optionally and automatically tolerate transient crash failures occurring on any of the distributed elements. It relies on checkpointing and underlying middleware mechanisms to do so. To further improve checkpointing efficiency, both in size and frequency, the programmer can introduce intermediate user-defined checkpoint data and code within the task processing program. The framework used without fault tolerance accelerates application development, does not introduce runtime overhead and yields to expected speedup. When enabling fault tolerance, our framework allows, despite failures, correct completion of applications with limited runtime and data storage overheads. Experiments run with up to 128 workers study the impact of some user-related and implementation-related on overall performance, and reveal good performances for classical JavaSpace-based master-worker application profiles.**

***Keywords*-master-worker; framework; distributed fault tolerance; checkpointing; user-framework-middleware cooperation;**

I. Motivation and objectives

PC Clusters are becoming common facility in many organizations (science research centres, financial companies...). Yet, most users unfamiliar with distributed computing do not exploit them, mainly because they do not know how to write parallel programs. The first contribution of our work is to provide a framework which considerably eases the development of master-worker (MW) distributed applications for the layman. The purpose of such a distribution is usually to speedup or size-up using cheap parallel machines. Yet, faults are more likely to happen in that context. Hence, fault tolerance (FT) becomes mandatory. Moreover, in an industrial context, applications are often subject to time constraints. For example, some long financial applications are run overnight and their results are expected by morning to decide on the strategy to follow for the day. Therefore, FT must also be efficient by enabling the application to (1) deliver correct results despite failures, (2) run with limited runtime and data storage overheads when no failures occur, and (3) waste as little work and time as possible on failure recovery. Automatic and frequent checkpointing at system level avoids restarting the application from the beginning after a failure by resuming execution on recovery from a recent checkpoint. However, it often leads to extensive data transfer and backup as the whole context of the application is saved. Application-level checkpointing allows a finer selection of data which must be saved but makes application development and testing more complex. Such an approach clearly does not fit laymen. Our second contribution is to propose an intermediate solution, with optional user-tunable framework-level checkpointing to ensure efficient FT by exploiting the programmer's knowledge of the application's semantics.

We target applications that can be parallelized according to the MW distribution pattern. FT mechanisms in our framework are interesting mainly for applications with long runtimes and medium to coarse grain task sizes. Moreover, we deal with *transient crash failures* resulting from hardware/software weaknesses or maintenance operations rather than application development errors or malicious faults.

Thereafter, we present our framework for MW applications (Section II) and delve into the mechanisms supporting its automatic and user-defined FT (Sections III and IV). Next we present related works (Section V) and preliminary evaluation results (Section VI). Finally, we summarize our work and propose future extensions (Section VII).

II. A framework for MW applications (without fault tolerance)

Our framework is designed above a variant of the MW pattern where the master assigns each worker a new task as soon as the latter has returned the result of its previous task. This leads to a natural load balancing as fastest workers get assigned new tasks more frequently. The master may choose different termination conditions: it could be that all initial tasks have been processed. Alternatively, the master can provide tasks to workers till reaching the desired result accuracy.

A. Virtual shared memory architectured solution

Besides master and workers entities, we can emerge the concept of a pool of tasks and results to further struc-

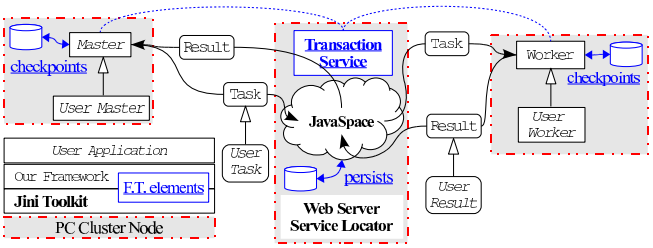

Fig. 1. Overall architecture and data exchanges

ture the pattern between data-oriented components (the pool) and processing-oriented components (master and workers). Considering no tight-coupling between processing entities is mandatory, information exchanges can be made through a third-party representing the pool. An advantage of that solution is to migrate part of the master and worker complexity towards the pool. Besides the software design gain it represents, this solution allows to use existing middleware to implement the pool and to benefit from its operative FT mechanisms.

JavaSpace originated from Linda programming model [1] and enables programs to exchange Java objects through a virtual shared memory. It comes as a Jini service and we use it as the substratal layer for the pool implementation in our solution. Fig. 1 illustrates the resulting framework architecture and the relationships between components: the master posts to the JavaSpace the tasks it was provided with by the user, and starts observing the JavaSpace for the arrival of results. Whenever it takes a result from the JavaSpace, the master re-evaluates the distributed computation termination condition. Results post-processing may lead to the creation of new tasks which are posted to the JavaSpace. When the termination condition is reached, the master posts a special message into the JavaSpace to signal the computation end to all workers. On its end, a worker retrieves a task from the JavaSpace, processes it and posts the yielded result to the JavaSpace. A worker quits after reading the master's stop message in the JavaSpace.

B. Developing with the framework

Restricting the form of the application code to the structures described above allows us to propose generic classes for the Master, Worker, Task and Result objects. These classes take care of discovering the appropriate Jini resources, and they execute the flow of operations. The programmer inherits his classes from ours and only needs to write domain-specific code (cf. italic portions in fig. 1).

We illustrate this process by using our framework to distribute the computation of an approximation of π through a Monte-Carlo method. To distribute that algorithm according to the MW pattern, the master generates N random points and posts them into the JavaSpace. Each worker takes a point, computes whether it lies within the unit circle and returns a boolean result. The master counts the positive results and divides that number by N to compute an approximation of π.

Using the *mawork* ("master - worker") package of our framework, the programmer starts by defining what constitutes a task (*PiTask*) and a result (*PiResult*):

```
public class PiTask extends mawork.Task {
  public Double x, y;
  public PiTask() {}
  public PiTask(Double x_, Double y_)
  {  x = x_, y = y_;  }
}

public class PiResult extends mawork.Result {
  public Boolean isInside;
  public PiResult() {}
}
```

Next, he implements the worker's *compute* method (cf. class *PiNonFTWorker* below) which processes a *PiTask* to produce a *PiResult*. The programmer still has to name the Jini resources he wants to use since multiple instances may be available.

```
public class PiNonFTWorker
  extends mawork.Worker<PiTask, PiResult> {
  public PiResult compute(PiTask t) {
    PiResult result = new PiResult();
    result.isInside = ((Math.sqrt((t.x*t.x)
          +(t.y*t.y)))<=1);
    return result;
  }
  public PiNonFTWorker(String lookupAddress,
                       String spaceName)
      throws java.rmi.RemoteException {
    super(lookupAddress, spaceName, PiTask.class);
  }
}
```

Lastly, the programmer writes the master's *checkTermination* method (cf. code below) which the base class calls upon each result retrieval. If this method returns true, the base class writes a stop order and executes the programmer-provided *postProcess* method; otherwise it waits for a new result.

```
public class PiNonFTstatMaster
  extends mawork.Master<PiTask, PiResult> {
  private int nbPostedTasks;

  public PiNonFTstatMaster(String lookupAddress,
                           String spaceName,
                    Vector<PiTask> initialTasks) {
    super(lookupAddress, spaceName,
          initialTasks, PiResult.class);
    nbPostedTasks = initialTasks.size();
  }

  public boolean checkTermination
                    (Vector<PiResult> results) {
    return results.size() == nbPostedTasks;
  }

  public void postProcess
                    (Vector<PiResult> results) {
    int nbIn = 0, nbOut = 0;
    for (int i=0; i<results.size(); i++)
      (results.get(i).isInside) ? nbIn++ : nbOut++;

    System.out.println("pi=" +
            (double)(4*(double)nbIn/(nbIn+nbOut)));
  }
}
```

In this example, the Master does not post new tasks atop the ones provided at the beginning. However, our API provides an *addTask(Task)* method to do so when needed.

III. Generic fault tolerance

When FT is enabled, the framework automatically performs some operations to guarantee no task or result is lost should some entity fail. These operations preserve application consistency, and ensure only the faulty element needs to be restarted.

A. Description of the fault tolerance protocol principles

Basically, a task or result does not entirely leave a node till it is safely checkpointed at its new location.

When a worker retrieves a task (*i.e.*: *take* operation), it immediately saves it locally and the JavaSpace "remembers"

that task till it gets the confirmation the worker has saved it (cf. Section III-B). Thus, if the worker fails before having successfully saved the task, the latter remains available on the JavaSpace for another worker. Otherwise, the task permanently exits the JavaSpace. Afterwards, the worker keeps the initial task checkpoint till that task is processed and the corresponding result is successfully posted on the JavaSpace (*i.e.*: *write* operation). Hence, if the worker fails during task processing or a problem occurs when posting the result, the task is still available and can be re-processed by the worker.

The master keeps three items: a list of tasks to be posted, the number of tasks currently written in the JavaSpace and a list of retrieved results. Whenever a list changes, the master checkpoints itself by saving these three items. In the general case, the master does not need any other information to recover from a failure. The "take and save" operation is atomic, just as when the worker retrieves a task from the JavaSpace.

Concerning network FT, Jini (and thus, the framework) relies on TCP to deal with potential message loss.

B. Implementation with middleware fault tolerance

The framework is written in Java and built on top of Jini services, which allows us to take advantage of existing FT mechanisms. Firstly, checkpoint and restore operations are eased by Java **serialization**/deserialization process. This constitutes an interesting advantage specially for the neophyte user. Other languages (*e.g.*: C/C++) do not provide such facilities and programmers have to resort to third-party libraries which do not reach the same level of automation.

Secondly, the JavaSpace service comes in two flavors: transient (in-memory) or **persistent**. We use the latter version, where the JavaSpace keeps track on disk of any modification of its content so as to have the exact same entries upon recovery than right before the crash.

Thirdly, the framework uses Jini's **transactions** service to ensure the "take from JavaSpace and save locally" operation is atomic (same for "write to JavaSpace and erase locally"): the *take* method is executed within a transaction which gets committed only after the associated checkpoint operation succeeds. If the master (or worker) fails between the retrieval of an entry and the end of its disk-write operation, the transaction times out and the entry remains on the JavaSpace. The transaction lease duration is an important parameter: if chosen too short, the lease will expire before the end of the checkpoint operation; and if chosen too long, it will take time before a failure is detected. The adopted strategy chooses a reasonably small duration and mandates a local process to regularly renew the lease (till the operations under transaction finish): if the machine hosting the master (or a worker) fails, the local process does not renew the lease which will expire and cancel the operations under transaction.

C. Enabling fault tolerance and restarting after failures

To use the generic FT, the programmer needs only to make his classes derive from *tomawork* ("tolerant mawork") package instead of *mawork*. No other change is needed to benefit from the FT mechanisms described in section III-B.

Upon recovery, the JavaSpace looks for and loads its last persisted state. Master and workers need to be specified the "rego" command line parameter to do the same. Since it relies on RMI [2], the JavaSpace can be made **activatable**. This feature automatically starts a service whenever it is requested. Thus, it can be used to automatically restart a failed JavaSpace. As for master and workers, full implementation of failure detection mechanisms will help automating their recovery.

IV. User-defined checkpoints

Section III-A presented a FT solution which may fall short when (1) each task lasts a considerable amount of time, (2) failures are not so rare, and (3) the application is submitted to soft time constraints. Therefore, our framework enables some collaborations with the programmer which help improving the initial FT based on the programmer's knowledge of the execution environment and the application semantics.

A. Programmer's contribution and framework internals

The programmer may **choose to enable or disable FT mechanisms independently on the Master, Worker or JavaSpace entities**. For example, if the JavaSpace service runs on a very reliable machine, the user can choose the transient version of the service. We thus exploit the user's knowledge of the target platform to save costly FT related operations. We also exploit his application knowledge: for example, if a Monte-Carlo application can tolerate some irregularity in the generated random numbers sequence, the loss of a task may not be very serious and FT may be relaxed on workers.

The programmer can segment the task processing code into blocks whose runtime is not negligible. A checkpoint is taken at the end of each block. The programmer is thus **in charge of defining the checkpoint frequency**.

Once a block is completed, most of its internal variables are not useful anymore and remembering the intermediate result of the block is enough to pursue with next block. Instead of saving the whole process context, we once again put the programmer's knowledge to **contribution by asking the programmer to specify the minimum required data**. Right before calling the checkpointing method, the programmer constructs a serializable object containing all the information required to restart the execution from that point.

Upon restart, the framework is responsible for retrieving the checkpoint and passing the information along to the computation method. After adding intermediate checkpoints to his worker computation code, the programmer **needs to specify how to resume computation from the saved information**. To facilitate checkpoint management, intermediate checkpoint locations are numbered according to a user-supplied index starting at 1 (0 is reserved for the framework initial and automatic checkpoint).

B. Example

Hereafter we consider the code of a mock application:

```
public MockResult compute(MockTask t) {
  int i = Aux.method1(t);
  int sum = 0;
  for (int j=0; j<5; j++)
```

```
    sum += Aux.method2(i);
  boolean b = Aux.block2(sum);
  MockResult br = Aux.operation3(b);
  return br;
}
```

Let us say *method1* is quite time-consuming; it is worth checkpointing the value of *i*. If the programmer also wants to checkpoint after each call to *method2*, he needs to create a serializable object containing the values of *i*, *j* and *sum*. Supposing *block2* result is also worth saving we have:

```
public MockResult compute(MockTask t) {
  java.io.Serializable s;

  int i = Aux.method1(t);
  s = i; checkpoint(t, 1, s);
  // t: task, 1: checkpoint id,
  // s: intermediate result

  int sum = 0;
  for (int j=0; j<5; j++) {
    sum += Aux.method2(i);
    s = new MySave(sum,i,j);
    checkpoint(t, j+2, s);
  }

  boolean b = Aux.block2(sum);
  s = b; checkpoint(t, 7, s);

  MockResult br = Aux.operation3(b);
  return br;
}

class MySave implements Serializable {
  int sum, i, j;
  MySave(int sum_, int i_, int j_)
  {  sum = sum_, i = i_, j = j_;  }
}
```

Since the framework does not enforce a structure on the computational code, it is up to the programmer to specify how to resume from a checkpoint situation:

```
public MockResult compute
    (tomawork.WorkerCheckpoint c) {
  MockTask t = (MockTask)c.getTask();
  java.io.Serializable s;

  boolean b;
  if (c.getCkpId() < 7) {
    int i, sum = 0, j = 0;
    if (c.getCkpId() == 1)
      i = (Integer)c.getIntermediateResult();
    else {
      MySave ms = (MySave)c.getIntermediateResult();
      sum = ms.sum; i = ms.i, j = ms.j;
    }
    for (; j<5; j++) {
      sum += Aux.method2(i);
      s = new MySave(sum,i,j); checkpoint(t, j+2, s);
    }
    b = Aux.block2(sum);
    s = b; checkpoint(t, 7, s);
  } else { // chPt id = 7
    b = (Boolean)c.getIntermediateResult();
  }
  MockResult br = Aux.operation3(b);
  return br;
}
```

V. Related Work

MW style of computation is very popular and widespread. The Internet-based BOINC platform [3] uses the MW paradigm to harness the computing power of internet users for various scientific applications. Given the nature of their computing environment, BOINC applications are more concerned by malicious than crash failures, and hence rely on task replication and rescheduling. In more controlled environments, such as PC clusters where computing resources are less but more secure, existing FT solutions focus on crash failures and rely on checkpointing. When implemented "from scratch", MW applications may easily be made fault-tolerant using system-level checkpointing: MPICH-V [4] provides such support for arbitrary MPI applications using the BLCR sequential checkpointer [5]. However, such approach does not simplify the writing of MW applications and its FT is agnostic to the MW pattern resulting in sub-optimal efficiency. Therefore, several efforts have been made to relieve the programmer from low-level implementation aspects and let him concentrate on the application. MW-Condor is a MW API easing MW applications development on top of Condor high-throughput computation environment [6]. Likewise, ProActive grid middleware provides a MW API on top of its *active objects* programming model [7]. GridRPC and MapReduce programming models resulted in middleware such as DIET [8] that inherently follow a MW pattern. Most of these approaches deal with workers failures by making the master reschedule corresponding failed tasks. The master's FT, when present, consists in periodic checkpointing of the master. When the latter fails, the whole application is restarted from such a checkpoint. As a result, (1) FT burden is entirely placed on the master, and (2) all computation done by worker(s) before they (or the master) fail(s) is lost. Using Java and JavaSpace technology, our framework proposes a very simple MW API to ease fault-tolerant MW application development. Unlike existing approaches, our framework provides a generic FT where only failed components undergo recovery. Moreover, our framework enables several collaborations with the programmer to improve overall efficiency by relaxing the generic FT and enabling him to introduce intermediate application-level checkpoints during tasks processing on the workers.

VI. Framework evaluation

Extensive testing attests that an application developed with our framework yields correct results, even in the presence of potentially concomitant failures. We evaluate our framework according to: (1) the ease of development, (2) the framework's efficiency without any FT, (3) the FT incurred overhead, and (4) the recovery quickness.

Tests were lead on a Linux 256-PC cluster with Gigabit Ethernet. The framework used Jini 2.1 and SUN's JVM 1.6.0_11. JVMs of the master and of each worker run on separate nodes. An additional node runs the Jini services. We use as benchmark a mock application where we can vary the tasks number as well as their duration and size. All tasks share the same settings and result size equals task size.

1) Ease of developing MW applications: Writing the mock application without our framework needed 107 logical lines of source code (LLSC): it required knowledge of the JavaSpace API and understanding of the Jini discovery mechanism. With our framework, the mock application needed only basic Java programming skills and resulted in just 62 LLSC. Moreover, the resulting application is fault-tolerant. We expect it will prove **easier to develop using the framework than "from scratch"**.

2) Framework overhead without fault tolerance: Our framework introduces a thin layer between the application

Entries (tasks and results)		Overheads	
Total Number	Individual Size	Absolute (s)	Relative (%)
256	1000 doubles	4	3.3
1024	1000 doubles	5	1.0
2048	1000 doubles	6	0.6
4096	1000 doubles	7	0.4
8192	1000 doubles	8	0.2

TABLE I
ENTRIES NUMBER INFLUENCE ON JAVASPACE PERSISTENCE OVERHEAD.

Workers Number	Runtimes (s)		
	Theoretical	Experimental no FT	Experimental with FT
32	9390	9392	9411
64	4710	4712	4725
128	2370	2372	2389

TABLE II
OVERALL PERFORMANCES.

and the Javaspace API which may incur some overhead. A comparison of runtimes achieved by the framework-based implementation of the mock application and the framework-less one, show that **without any FT, our framework does not introduce any overhead**.

3) Fault tolerance overhead without failures: Persistence, transactions and checkpoints are sources of overheads. To begin with, we did not notice any negative influence of concurrent accesses: executing 64 tasks on 64 workers takes the same time as executing 128 tasks on 128 workers.

Using **a persistent JavaSpace is more costly than a transient one** as it involves disk I/O operations. Given a disk subsystem, that cost should depend on the JavaSpace entries size (tasks and results) and their number. To assess the cost of JavaSpace persistence, we measured the runtime of the mock application without any FT, and then, with only the JavaSpace persistence. We used 128 nodes and 2048 tasks of 120s each. Overhead remains almost constant when varying entries size (up to 10000 doubles) and does not exceed 7s in all cases. With entries size fixed at 1000 doubles, absolute overhead increases linearly as entries number increases exponentially (cf. Table I). These experiments verify that persistence slightly decreases the JavaSpace performance; but that considered data volumes do not seem to matter in our case. We do not consider bulkier data for it is far more efficient to indicate their remote location (using URLs) than to make them transit through the JavaSpace. Concerning **transactions overhead**, our experiments reveal an almost constant overhead when increasing the entries number (till 8192) with 128 nodes and tasks of 120s and 1000 doubles.

Checkpoint operations overhead depends on checkpoint size and frequency. As a result, observed overhead on workers was low given the relatively low task sizes and checkpoint frequency. However, systematic checkpoint of the master (*i.e.*: after each modification of its lists) amounted to a huge overhead. We believe that keeping track of the modifications only will greatly mitigate that overhead.

Table II reports the mock application runtimes when varying the workers number. Parameters are similar to the ones which could be encountered in a real-life application: 10000 tasks of 1000 doubles and lasting 30s. Experimental runtimes achieved without any FT (3^{rd} column) are really close to theoretical runtimes (2^{nd} column). Experimental runtimes (4^{th} column) achieved with full FT on workers and the Javaspace but only transactions on the master (*i.e.*: no checkpointing) testify to negligible overhead ($< 0.8\%$). This setting fits a configuration where the master runs on a very reliable host. We expect similar results with the improved checkpointing on the master. Achieved performances should satisfy most users, especially in view of the low effort they have to provide.

4) Recovery delay: To measure the recovery time, we compare the failure-free runtime of a run with one worker executing a single task of 120s, to the total runtime of the application when the worker fails after 55s. This yields a runtime of 176s (*i.e.*: 120+55+1). By introducing an intermediate checkpoint after 40s, the worker finishes in 136s (*i.e.*: 120+(55-40)+1). In both cases, the additional 1s-overhead accounts for the recovery time and results from small checkpoint size of 1000 doubles. Similar results were obtained with other failure times which shows the interest of intermediate checkpoints.

VII. CONCLUSION AND PERSPECTIVES

We have presented a framework to facilitate the development of bag-of-tasks applications. Non-specialist programmers have fewer and easier code to write, and the resulting application achieves good performances. Moreover, the application can tolerate concomitant transient crash failures of any of its elements. Given not too small application grain, FT mechanisms introduce acceptable overhead. For long-running tasks, the programmer can add intermediate checkpoints at strategic locations, and include only the necessary data to further improve FT performance. Performance is directly related to the underlying middleware and is expected to improve using a performance-oriented JavaSpace (*e.g.*: GigaSpaces).

Future works are envisioned along three lines. Firstly, we must improve the master's checkpoint strategy. Secondly, the framework can easily be extended to funnel checkpoints to a remote location so as to guard against permanent node failures. Lastly, it would be useful for non-specialists users to augment the framework with automatic resources discovery, deployment and fault detection mechanisms; we could again rely on Jini mechanisms for that.

ACKNOWLEDGMENT

This research is partially supported by *Region Lorraine*.

REFERENCES

[1] D. Gelernter, "Generative communication in Linda," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 1, pp. 80–112, 1985.
[2] *Java RMI Specification*, Oracle, 2010, v. 1.6, ch. 7.
[3] D. P. Anderson, "BOINC: A System for Public-Resource Computing and Storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. IEEE Computer Society, 2004.
[4] A. Bouteiller, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello, "MPICH-V: a Multiprotocol Fault Tolerant MPI," *International Journal of High Performance Computing and Applications*, 2006.
[5] P. H. Hargrove and J. C. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters," in *Proceedings of SciDAC*, 2006.
[6] J.-P. Goux, S. Kulkarni, J. Linderoth, and M. Yoder, "An Enabling Framework for Master-Worker Applications on the Computational Grid," in *HPDC*, 2000.
[7] *ProActive Programming: Reference Manual*, 2010, v4.3.0.
[8] *DIET User's Manual*, http://graal.ens-lyon.fr/ diet/, 2008, v2.3.