An Application-Assisted Checkpoint-Restart Mechanism for Java Applications

Diana Andreea Popescu, Eliana-Dina Tirsa, Mugurel Ionut Andreica, Valentin Cristea

An Application-Assisted Checkpoint-Restart Mechanism for Java Applications

Diana Andreea Popescu[1]
École Polytechnique Fédérale de Lausanne
Lausanne, Switzerland
diana-andreea.popescu@epfl.ch

Eliana-Dina Tîrşa, Mugurel Ionuţ Andreica, Valentin Cristea
Politehnica University of Bucharest
Bucharest, Romania
eliana.tirsa@cs.pub.ro, mugurel.andreica@cs.pub.ro, valentin.cristea@cs.pub.ro

***Abstract*—In this paper we present a novel application-assisted checkpoint-restart mechanism for Java applications. The checkpoint-restart API provides the application developers with full control over what data needs to be check-pointed. The novelty of our system is that it allows different checkpoint periods for different data items. Our implementation makes full use of the Java Reflection API.**

Keywords— checkpoint; restart; Java; fault tolerance

I. INTRODUCTION

Computer systems are prone to hardware and software failures, thus making checkpointing an increasingly important tool for them. Checkpointing is an important mechanism for fault tolerant systems, migration systems, load-balancing systems, play-back debuggers and many others. Currently, the major concern regarding checkpointing is the overhead, defined either as the amount of time added to a program due to checkpointing or as the extra space needed for storing the checkpoint files (or as a combination of both).

In this paper we introduce an application-assisted checkpoint-restart mechanism for Java (multi-threaded) applications. The checkpoint mechanism may be backed by a reliable distributed storage system for the checkpoint files, thus ensuring data availability in case of hardware failures. The proposed solution is a non-transparent checkpoint-restart system which uses the Java Reflection Application Programming Interface. We base our design on the idea that not all the data is updated with the same frequency; in consequence, only part of the data has to be saved in a certain checkpoint file. We also provide an Application Programming Interface which can be used by the programmer in order to attain fault tolerance for his application.

The rest of this paper is structured as follows. In Section II we present the motivation for the work presented in this paper, as well as some of the main characteristics of our proposed mechanism. In Section III we discuss related work. In Section IV we mention several realistic applications which would benefit from the approach presented in this paper. In Section V we present the software architecture of the checkpoint-restart mechanism. In Section VI we present experimental results. Finally, in Section VII we conclude and discuss future work.

[1] The work presented in this paper was performed while the author was a student in the Computer Science Department of the Politehnica University of Bucharest.

II. MOTIVATION AND OVERVIEW OF THE PROPOSED CHECKPOINT-RESTART MECHANISM

We chose to explore the area of non-transparent checkpoint-restart mechanisms which, although they are more intrusive from an application development perspective, provide the application developer with full control over the relevant data, allowing him to decide which data needs to be checkpointed. Our goal was to design a checkpoint-restart mechanism and to propose an associated Application Programming Interface for it, which would be easy to use in a Java multi-threaded application. We will present below the main advantages of non-transparent checkpointing that our proposed mechanism also shares, and we will present the specific details of our approach.

The major advantages of application-level checkpoints are the reduced size of the checkpoint data and the high portability of the checkpoint files, thus allowing an application to be restarted on different machines. Another convenient feature is that we can save only the most important data necessary for the restart of the application. Regarding portability, we use binary checkpoints, which are only portable in the Java world, but the advantage over the far more portable XML files lies in the size of the checkpoint.

In our design, the useful data that has to be checkpointed consists of selected objects of the Java application. These objects will not be saved all in the same checkpoint and the checkpoints will not be performed at fixed time intervals. Because not all objects are saved in the same checkpoint, checkpoint files will be smaller and the duration of their writing will also be lower. In order to be able to recover all the objects, at restart time, we will need a series of checkpoint files, not only the most recent one. This is also a bonus point for applications that need history. If the application cannot make use of the obsolete information contained in the checkpoints, we also provide a cleanup mechanism which

erases checkpoint files that contain exclusively outdated values.

The automation support is another strong point of non-transparent checkpoint restart. It may seem intrusive to have explicit checkpointing function calls in the code, but the effect is smoothed if they are integrated in the development phase of the application. Besides, there is a better understanding of the specific application needs. Also, due to the fact that the non-transparent checkpoint is at user level, the implementation is simpler, as one doesn't have to program inside the kernel.

Incremental checkpointing, saving only the data that has been updated from the last checkpoint, is another useful feature available to non-transparent approaches. This issue is even more important when, for fault tolerance purposes, the checkpoint files are saved remotely, on a distributed storage, accessible by all the system entities, such that if one service fails, a new one can be deployed on another machine, using the last checkpoint in order to restore the application data.

Another advantage of non-transparent checkpoint is that the application doesn't need to be interrupted while checkpointing. Moreover, having the checkpointing mechanism at application level, the timing of the associated checkpoint, reported to the application progress and data updates, can be more closely controlled. In our implementation, checkpointing is performed by a separate thread that only takes action when the update period for a group of objects has been exceeded. However, having multiple threads access the same data introduces the need for synchronization.

III. Related Work

Checkpointing is a method which provides an application with fault tolerance by recording the application's state and using the saved data to restart the application in case of failure. Checkpointing can also be used for process hibernation (conserving the machine state during power cuts) or suspension (to save memory space or to allow rolling back to different states). The checkpoint file can be saved either locally, to stable storage, or in a remote node's memory.

One of the checkpoint/restart methods [1] implemented in operating systems is based on creating a file that describes a process currently running. The process can then be "reconstructed" along with the saved state, based on the data from the checkpoint file. Running applications can be "saved" periodically (e.g. based on notifications from monitoring applications). Once the application status has reached a stable storage, the application can be restarted and reconfigured if necessary. Checkpoint/restart techniques can also be used to reduce the time the applications are stopped for maintenance (hardware or operating system) by migrating applications on another machine. In Grid and Cloud systems, checkpoint/restart is used to suspend or migrate jobs [2].

As presented in [3], checkpoint-restart mechanisms can be classified with respect to the context, the agent that provides the checkpoint-restart functionality and the implementation. Using the first criterion of classification, the checkpoint-restart implementations may be user-level or system-level. The user-level implementations can require the programmer to use an existing API for checkpoint-restart when writing the application. Alternatively, this can be inserted automatically by a compiler. If we do not want to modify the source code of the application, the checkpoint-restart primitives can be invoked by signal handlers defined at user level. Another option is to use an environment variable, the checkpoint library being loaded without recompilation or relinking of the application. System-level implementations can be deployed in the operating system or in hardware. For the operating system implementations there are many alternatives: as a kernel-mode signal handler, system call or kernel thread.

System level checkpoints enable a high level of transparency and flexibility. The checkpoints are taken automatically and restarting the application can be performed without the user's intervention. The major disadvantage is that system level checkpoints cannot use the semantic information available at the application level. As a result, the size of the checkpoint is larger.

Implementation at the application level is often the most effective, given that it is known which data structures need to be saved and which not. But this approach presents a number of disadvantages. It is often not possible to change the source code. Another disadvantage consists of checkpoint time restrictions; there may be a long delay between the time when a checkpoint is requested and actual writing of that file on the disk. The major advantages of the application level checkpoints are the reduced size of the checkpoint data and the high portability of the checkpoint files, thus allowing an application to be restarted on a different machine. In the next section we mention several advantages of our application-level approach.

Library-level implementations solve some of these drawbacks [4, 5]. Libraries do not require changes to application source code and often use a signal "handler" to perform the checkpoint, so the time restrictions are eliminated. Implementations of this type have a common procedure for restart. There is however a major obstacle for the implementation of the checkpoint/restart mechanism as a library: to impose restrictions on system calls that can be used by the application (all forms of inter-process communication are generally prohibited). As a result, in the case of shell scripts and the majority of parallel applications, a library level implemented checkpoint mechanism cannot be applied.

Checkpoint/restart implementations at the operating system kernel level promise to eliminate much of these restrictions [6, 7]. At this level most of the core data structures are available and applications can usually be saved at any time. The implementation of checkpoint/restart in the kernel is much more difficult than at application-level or as a separate library.

BLCR [6] and Zap [7] are checkpoint/restart implementations in the Linux kernel, using dynamically loadable kernel modules that do not need the kernel source and do not require recompiling it. There are operating systems, such as IRIX produced by SGI or Unicos produced by Cray, where the checkpoint mechanism is implemented directly in the OS kernel. BLCR saves and restores the state of a process using an existing module, previously used to "fork" processes running on Beowulf distributed clusters. BLCR extended this

module with support for processes with multiple threads, files, pipes. As weaknesses, BLCR can not checkpoint at socket level and can not preserve the IP addresses of the checkpointed applications. Solution developers have left the applications or messaging libraries to address these issues. Also, BLCR can not successfully restart an application if one of the component process IDs are assigned to another process during the application restart.

Application-assisted checkpoint-restart APIs and mechanisms were also developed for the IBM Blue Gene Architecture [8].

An increased level of interest for application-level checkpoint restart mechanisms also appeared in the exascale computing and high performance computing fields [9, 10]. Checkpointing in distributed environments (e.g. Grids) has been considered in [11].

Specific application-level checkpoint-restart mechanisms for Java applications have also been considered. In [12] the authors consider the problem of efficiently checkpointing user-defined objects in Java applications. Their approach is similar to ours up to a point. First, they do no consider the possibility of having different checkpoint periods for the objects; instead, they consider incremental checkpoints, in which it is possible to avoid checkpointing an object if no modifications occurred. Second, they employ program specialization techniques in order to achieve efficient checkpoint implementations for various types of objects and modification patterns. Our approach is different: we define multiple types of checkpointable objects, e.g. checkpointable collections. In [13] code instrumentation techniques are used for developing a checkpoint-restart mechanism for Java applications. No support in the application source code is required (and, thus, no API is provided). However, a language for specifying checkpoint regions is defined, which needs to be used in order to guide the checkpointing process. Moreover, it seems that the developed techniques are intended for single-threaded Java applications only.

IV. Realistic Applications that would benefit from our Approach

Our proposed mechanism is suited for checkpointing data-intensive Java applications that have the following properties:

- the managed data is represented as a graph of objects
- various fields of the objects are updated (periodically) with different periods
- the application can benefit from maintaining the history of the updates

One example that meets the above criteria is a service similar to the Google Maps support for real time traffic. It recommends best routes, taking intro account the monitored traffic in a given city.

Let's consider that the graph of objects of such an application consists of a network of *Street* objects. The *Street* objects are *Vectors* of *StreetSegment* objects (the portion of a street between two intersections). Among the object fields of the *StreetSegment* object we can find: *Intersection*, *TrafficLight*, *Station*, *LegalSpeedLimit*, *EstimatedTraffic*, *NavigationTime*, *Priority*, *UpdatePeriod*, *Accident*. For a given *StreetSegment* object, the fields *TrafficLight*, *EstimatedTraffic* and *NavigationTime* are updated more often than *Priority*, *UpdatePeriod*, *LegalSpeedLimit*.

We may safely assume that the field traffic information is collected and transmitted by wireless sensors powered by solar energy. We know that sensors communication is an energy-wise expensive operation; we want to retrieve data at different periods from different locations. The *Priority* field quantifies this period for a given street segment. Depending on the time of day, or day of the week, we may assign higher priorities to very circulated streets or where traffic statistically fluctuates more.

An *Intersection* object may contain the fields: *StreetVector*, *StreetSegmentVector*, *TrafficLightVector*, *TraversalTime* (this will be updated often). Finally, a *TrafficLight* object contains the fields: *RedPeriod*, *GreenPeriod*, *State*, *PedestrianCtrlButton*. *RedPeriod* and *GreenPeriod* are subject to frequent updates.

Another example, this time from the networking field, is a data transfer scheduling service. The *DataTranfer* object may contain the following fields: *NLinkVector*, *Rnode* (controllable devices at the two ends of the link), *StartTime*, *EndTime*, *ReservedParams* (for example bandwidth). An *NLink* object is a collection of *LinkSegment* objects (a part of a link between routing devices). A *LinkSegment* may have the following fields: *TotalBW*, *AvailBW*, *LossRate*, *ScheduledTransfers*.

The information about links is retrieved from monitoring services, at different frequencies for different links. Within a *LinkSegment*, *AvailBW* is the field that is updated more often, *ScheduledTransfers* is updated at longer periods, and *TotalBW* is updated very rarely (only in case of hardware infrastructure changes).

V. Software Architecture

A. The Checkpointing Mechanism

An overview of our system is presented in Fig. 1. As stated before, our checkpoint solution is suited to (multi-threaded) Java applications, but it can also be applied to distributed applications, provided there is no (checkpointable) data dependency between the components of the distributed system. The checkpointing architecture is composed of two threads, the checkpoint thread and the cleanup thread, that are distinct from the main application threads. In this way, the checkpoint is performed in parallel with main application execution, without the need for interruption.

The checkpoint thread takes care of all the computations necessary for determining which objects have to be checkpointed in the current checkpoint. After the data is written to a file, the thread suspends its execution for a specified period, more exactly until the next checkpoint time arrives for at least one of the registered objects. This is an efficient way of making processor time available to the other threads of the application or other applications that might be

running on the computer system. When an object is registered from one of the application threads, the checkpoint thread resumes its execution and recalculates the next checkpoint time, taking into account the new object's period of update. The checkpoints are not taken at fixed intervals of time and the checkpoint times are not determined by the application execution. These are two advantages to our design.

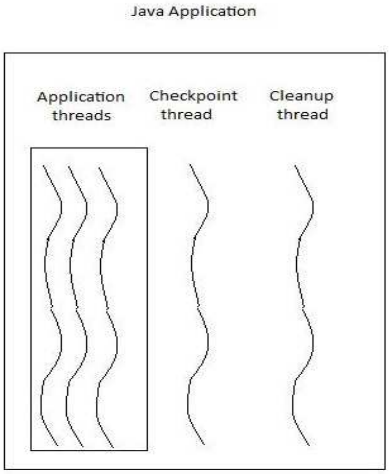

Fig. 1. Application, checkpoint and cleanup threads.

Because at first there are no registered objects, the checkpoint thread suspends its execution for a long period of time (*Long.MAX_VALUE* seconds). The thread runs in a continuous loop. The programmer can use the *stopCheckpoint()* method for stopping the thread. As a first action (repeated after each checkpoint), the checkpoint thread retrieves the next checkpoint time from a priority queue. After this, the thread suspends its execution for a specified period of time - until the next checkpoint time arrives for at least one of the registered objects. If an object is registered from one of the application threads, the checkpoint thread resumes its execution before the next checkpoint time arrives (it receives an interrupt from the application thread) and gets the next checkpoint time, taking into account the new object's period of update.

We define the period of update of an object as the period between two checkpoints for this object. The period of update can be seen as a few (usually 5) time intervals in which the object is updated or simply as a period of time that the programmer considers appropriate between checkpoints for that object. The programmer registers an object with the checkpoint thread, providing a period of update for the respective object and an object identifier. If an object has a field which implements the *Checkpointable* interface, it will automatically be registered with a default period of update during the first checkpoint of the object.

The new object's period of update is inserted in the priority queue in the registration function. If there are no new registered objects, the thread resumes its execution after the specified period of time and checkpoints the objects. Afterwards, it computes the next checkpoint time for the most recently checkpointed objects and it leaves the next checkpoint time unmodified for the rest of the objects. The last step of the loop is the traversal of the unregistered objects list and the removal from the data structures used by the checkpointing mechanism (a hash table and a priority queue) of the unregistered objects' entries. This step takes place after each checkpoint in order to be able to ensure a consistent state of the hash table entries.

Because we have many threads accessing the same data, we need synchronization mechanisms for accessing the objects. In the checkpoint method *checkpointObject(Object obj)*, the object *obj* is read using synchronized statements. Note that the application threads must also access (read/write) the object in the same manner.

If the object is checkpointed for the first time, the *checkpointObject* method will run recursively on the object's fields which are also *Checkpointable* objects. In order to avoid a deadlock, we keep a list of the *Checkpointable* fields and we checkpoint them after releasing the object's lock. Because the objects are registered from another thread than the checkpoint thread, the priority queue used for computing the next checkpoint time is a *PriorityBlockingQueue*, which is thread-safe. The list of unregistered objects is a *CopyOnWriteArrayList*, which is also thread-safe.

The cleanup thread's role consists of deleting the unnecessary checkpoint files. It runs at fixed intervals of time. The cleanup thread is started only if the programmer specifies this argument at the initialization of the *Checkpoint* object. Its role consists of deleting the unnecessary checkpoint files. As described above, in order to be able to recover all the objects we need a series of checkpoint files, not only the last one. The cleanup thread determines which files are not part of this series and deletes them.

B. Proposed Application Programming Interface

First of all, a class has to implement the *Checkpointable* interface in order to be registered with the checkpoint mechanism. Secondly, the class needs to have a no-argument constructor. Also, the access to the registered objects has to be synchronized when modifying the object.

The Application Programming Interface consists of the following Java classes:

Checkpoint

- public *Checkpoint* (String *fileName*, String *directoryName*, boolean *performCleanup*) - Constructs a *Checkpoint* object. The checkpointed data will be saved in the specified directory and in files which have the given name and an identifier appended at the end of the name. If the boolean flag is set to true, the cleanup thread will delete the unnecessary checkpoint files, otherwise all checkpoint files will be kept.

- public boolean *registerObject*(Object *obj*, String *identifier*, int *periodOfUpdate*) - Registers an object with the identifier *identifier* and the period of update *periodOfUpdate*.

The method returns true if the registration is successful and false otherwise.

- public void *unregisterObject*(Object *obj*) - Unregisters the specified object. This object will not be checkpointed anymore.

- public void *stopCheckpoint*() - Stops the checkpoint thread from running. No more checkpoints will be performed.

- public *HashMap*<String, Object> *restore*(String *filename*, String *directoryname*) - The method returns a HashMap with the checkpointed objects, constructed from the most recent series of checkpoints. The key of the *HashMap* represents the identifier given to the object at registration.

We also provide support for **checkpointable collections**. The idea behind the checkpointable collections is that the elements of a collection or a map could have different periods of update, thus allowing the elements to be saved during different checkpoints. A normal collection or map defined by the Java JDK cannot be registered. A collection or map which represents a field of a *Checkpointable* object implicitly inherits the object's period of update and it will have all its elements saved at the same time. The collections from our API have a period of update which is used for updating the changes made on the collections themselves (i.e. adding or removal of an element). Our API offers support for two collections: *CheckpointableVector* and *CheckpointableHashTable*.

CheckpointableVector<T>

- public *CheckpointableVector*(Checkpoint *ckp*, String *identifier*, int *periodOfUpdate*) - Constructs a new *CheckpointableVector* and registers it with the identifier *identifier* and the period of update *periodOfUpdate*.

- public void *add*(T *o*, Checkpoint *ckp*, String *identifier*, int *periodOfUpdate*) - Appends the specified element to the end of this list and registers it with the identifier *identifier* and the period of update *periodOfUpdate*.

- public void *add*(int *index*, T *o*, Checkpoint *ckp*, String *identifier*, int *periodOfUpdate*) **-** Inserts the specified element at the specified position in this list and registers it with the identifier *identifier* and the period of update *periodOfUpdate*.

- public T r*emove*(T *o*, Checkpoint *ckp*) - Removes the element from this list and unregisters it.

- public T *remove*(int *index*, Checkpoint *ckp*) - Removes the element at the specified position in this list and unregisters it.

CheckpointableHashTable<K,V>

- public *CheckpointableHashTable*(Checkpoint *ckp*, String *identifier*, int *periodOfUpdate*) **-** Constructs a new *CheckpointableHashTable* and registers it with the identifier *identifier* and the period of update *periodOfUpdate*.

- public void *put*(K *key*, V *value*, String *identifie*r, Checkpoint *ckp*, int *periodOfUpdate*) - Associates the specified value with the specified key in this map and registers it with the identifier *identifier* and the period of update.

public V *remove*(K *key*, Checkpoint *ckp*) - Removes the mapping for this key from this map if present and unregisters the object associated with the key.

C. Data Structures

When an object is registered with the checkpoint mechanism, the programmer provides the period of update through a parameter of the registration's function. If the period of update given as a parameter is below the *CHECKPOINT_LIMIT*, the period of update of the object will be automatically considered to be the *CHECKPOINT_LIMIT*. The objects' periods of update are stored in a hash-table. We use a HashMap structure in order to have a *O(1)* (constant time) access to the periods of update.

We also need an efficient structure for calculating the checkpoint time for each registered object. We use a heap structure ordered by the time in the future when the objects have to be checkpointed. In Java, we have the PriorityQueue class, which is a priority queue based on a priority heap. The elements of the priority queue are pairs which consist of the key of the object obtained using the *System.identityHashCode()* function and its next time of checkpoint. The next time of checkpoint is calculated as the current time returned by the *System.currentTimeMillis()* plus the period of update: Using a priority queue for keeping the next checkpoint times for the objects has several advantages. Firstly, we retrieve the next checkpoint time in *O(1)* time. Secondly, having the objects sorted by the time of checkpoint allows us to checkpoint only the elements from the priority queue which have the next checkpoint time lower than the current time plus an interval of time defined by a private field of the Checkpoint class. We use this interval of time, *EPS_TIME*, in order to include in the current checkpoint those objects with next checkpoint times very close in the future to the actual time. As a result, we eliminate the possibility of taking two checkpoints too close in time to one another. Our goal is not to take checkpoints below the *CHECKPOINT_LIMIT* time, so as to avoid the unnecessary load on the system.

We designed a **checkpoint hash-table** structure that holds the actual objects along with metadata used to construct the objects, since not all programmer registered objects and automatically registered objects are saved in the same checkpoint. We also store the necessary information for determining the times at which the objects have to be checkpointed. An object may be constructed solely from entries of the hash-table from the most recent checkpoint or it may have fields which are stored in prior checkpoints. Moreover, in the hash-table checkpoint we could have only the identifier of a prior checkpoint where the object is actually stored. As a result, in order to restore the checkpointed data, we do not need only the most recent checkpoint file, but instead we need a series of checkpoint files. In our implementation we actually store in memory two checkpoint

hash-tables, the one constructed during the previous checkpoint and the one constructed during the current checkpoint, in order to avoid iterating recursively through the list of registered objects and their fields, at every checkpoint.

The key of the hash-table is an integer obtained using the *System.identityHashCode()* function, which returns a unique hash code value for the object. The values stored in the hash-table for each entry depend on the given object's type. We store different information for a *Checkpointable* object, a *CheckpointableVector* and a *CheckpointableHashTable*. Also, if the object is not to be saved in the current checkpoint, the value holds only the identifier of the previous checkpoint where the object can be found. There are four types of values stored in the hash-table: *ValueId*, *ValueObject*, *ValueVector* and *ValueHash*.

For a *Checkpointable* object we use the *ValueObject* which has:

- a string representing the object's class name
- the identifier (string) of the object as given at registration or null if the object is automatically registered
- a list of keys and objects, depending on the object's fields (obtained with the aid of the Java Reflection Application Programming Interface)
- the object's period of update

If an object's field implements the *Checkpointable* interface, we save in the list the key of the entry in the hash-table for this field. Otherwise, if the field is one of the following: a primitive data type, a *String*, a primitive wrapper class type, we add in the list directly the value of the field.

The final and static fields are not saved, since these types of fields are considered to represent the state of the class, not that of a particular object. In addition to this, the fields of a *Checkpointable* object which do not implement the *Checkpointable* interface will not be registered in the hash-table and, consequently, will not be saved in checkpoints.

For a *CheckpointableVector* we use the *ValueVector,* which stores:

- a hash-table: the key is the position of the element and the value is represented by the key calculated using the *System.identityHashCode()* function for the element
- the identifier (string) of the vector as given at registration
- the vector's period of update.

For a *CheckpointableHashTable*, the *ValueHash* contains:

- a hash-table: the key is represented by the key from the *CheckpointableHashTable* hash-table for the element and the value is represented by the key calculated using the *System.identityHashCode()* function for the element
- the identifier (string) of the hash table as given at registration
- the hash-table's period of update.

D. Saving Data to Checkpoint Files and Restoring Data

Saving data to files. The hash-table structure is serialized and saved in a file which has the name given by the user and the checkpoint identifier appended to this name. In order to reduce the size of the checkpoint file and to reduce the serialization time, we implemented the *Externalizable* interface for the *Value* objects. The *Value* interface extends the *Externalizable* interface. The *ValueId, ValueHash, ValueVector* and *ValueObject* classes implement two additional methods:

- public void *writeExternal* (ObjectOutput *out*) throws IOException;
- public void *readExternal* (ObjectInput *in*) throws IOException, ClassNotFoundException;

In this way we defined our own protocol for writing to and reading from the checkpoint file. For *ValueId* we use only *writeInt()* and *readInt()*. For *ValueObject* we use *writeInt()* and *readInt()* for the period of update, *writeUTF()* and *readUTF()* for the name of the class, and for the list we first write its size and then write every object with *writeObject()* and read the object with *readObject()*. We proceed in the same manner for *ValueVector* and *ValueHash* and for the hash-table structure. There is no difference in how a class that implements *Externalizable* is used. When we call *writeObject()* or *readObject()*, the two methods will be called automatically.

The *ValueId* class is used for an object which must not be saved in the current checkpoint. We store only the identifier of the most recent checkpoint where the object can be found. The structure of the hash-table is presented in the following figure.

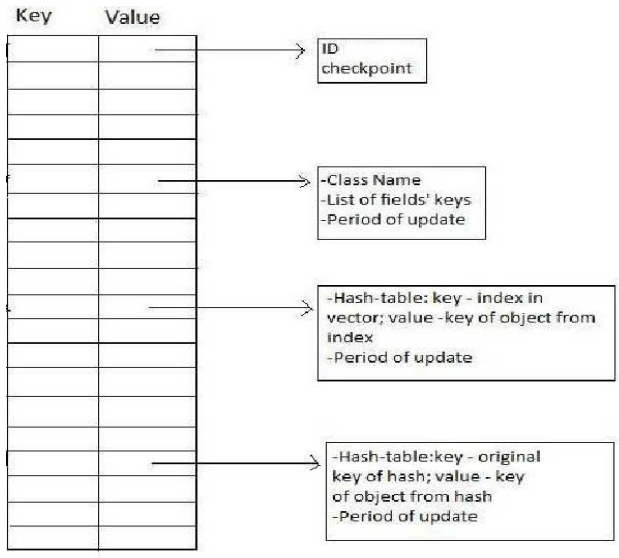

Fig. 2. Checkpoint hash table structure.

Restoring Data. Our proposed Application Programming Interface provides a method called *restore*, which returns a HashMap containing the registered objects reconstructed using the data from the checkpoints. The key of the hash represents

the identifier given as an argument for the registration function.

Firstly, given the name of the checkpoint file and the directory where the files were being saved, the function determines the most recent checkpoint file, using a *FilenameFilter*. Secondly, the hash-table contained in this checkpoint file is deserialized and, after this, we iterate through the keys' set, constructing each object, using Java Reflection. An object is added to the list of restored objects and is automatically re-registered, thus allowing it to be saved at the next checkpoint according to its period of update.

The function recursively constructs an object, using the information from the corresponding entry stored in the hash-table. If the information for constructing the object is found in the current checkpoint, we get the object's class name and, afterwards, we use the static method *Class.forName(String x)*, which returns the *Class* object associated with the class or interface with the given string name. Next, we invoke the no-arguments constructor of the class to create and initialize a new instance of the constructor's declaring class:

If the object's class is *CheckpointableVector*, we construct each element of the vector and add it to the new *CheckpointableVector* instance created. We proceed in the same manner for the *CheckpointableHashTable* objects. For reconstructing a *Checkpointable* object, we obtain the fields of the class. The values of the fields which do not represent *Checkpointable* objects are set using the method *set(obj, currentField)*. Before this, the field was made accessible with the *setAccessible* method.

If the information stored for the object is of type *ValueId*, we restore the checkpoint file whose identifier was found in the entry for the object. After this, we recursively call the construction function with the new restored checkpoint.

VI. Experimental Evaluation

A. Validation Tests

For the validation testing we constructed various scenarios for testing our Application Programming Interface. In our tests we used different types of objects. Each object (and object field) was assigned an identifier and was registered with a certain period of update. We started a thread that updated the fields of the registered objects with the frequency of the assigned period of update. We let the checkpoint thread run for a few (5) maximum update periods.

We checked that each checkpoint contained the objects that had to be checkpointed during a certain period and the correct checkpoint ID reference for the others. We restored the objects from various checkpoint files, obtaining the expected objects with their identifiers. We also verified that the cleanup thread correctly deletes the old checkpoint files and that all the objects can be restored from the remaining files.

B. Performance Tests

We conducted a series of tests in order to establish the performance of the checkpoint mechanism. We chose the following periods of update: 10 seconds, 20 seconds, 50 seconds, 100 seconds and 150 seconds. The checkpoint minimum limit is 10 seconds. The tests were conducted for 100 000 objects and 500 000 objects. Each object has been assigned one of the periods listed above. We formed 5 groups of equal size. We started a thread that updated the fields of the registered objects with the frequency of the assigned period of update and let it run until it updated five times the objects in the category of 150 seconds. We also started the checkpoint thread and we measured every checkpoint (processing time and writing to file).

For the second round of tests, we started the updating thread and a modified checkpoint thread that serialized the whole graph of objects every 10 seconds (the minimum update period of an object). We wanted to compare our proposed solution with simple serialization of all the objects. Even if we save fewer objects per checkpoint, our solution might take more time to decide which objects it should save.

The measurements were performed on a system with Intel Core 2 Duo, 2.5GHz, 3GB RAM.

For 100000 objects (1000 types of objects, 100 objects of each type), the average time per checkpoint for our implementation was 190ms, whereas for serialization of the whole graph, the mean time per checkpoint was 220ms. Also, the average size of a checkpoint file was 2.8MB in our approach and 5.2MB in the serialization case.

We tested similarly with 500000 objects (1000 types of objects, 500 objects of each type). The average time per checkpoint for our approach was 1450ms, whereas for serialization of the whole graph, the mean time per checkpoint was 1600ms.

As we can see from the results presented above, we have obtained a checkpointing overhead of about 90% of the serialization time of the whole graph. It may not seem a significant improvement, but in long running applications, it can make a difference. Also, the average size of the checkpointing files was visibly smaller in our approach, needing less storage space. This is particularly useful for applications that need to maintain the history of the updates.

VII. Conclusions and Future Work

In this paper we presented a novel application-assisted checkpoint-restart mechanism. The mechanism and its associated API were designed for Java applications (and, thus, our implementation is also Java-based). Experimental evaluation has shown that our system improves upon the standard Java serialization both in terms of running time and (particularly) in terms of disk space.

As future work we intend to improve our checkpoint protocol, in order to obtain better improvements in running time compared to the standard Java serialization mechanism. Moreover, we intend to expand our checkpoint mechanism to distributed applications and distributed (shared) objects.

References

[1] Eric Roman, „A Survey of Checkpoint/Restart Implementations", Berkeley Lab Technical Report (publication LBNL-54942), July 2002.

[2] I. Foster, C. Kesselman, J. Nick and S. Tuecke, „Grid Services for distributed system integration", Computer, vol. 35, no. 6, Jun. 2002, pp. 37 – 46.

[3] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa and S. Jiang, "Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance", Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, 2005.

[4] M. Litzkow, T. Tanenbaum, J. Basney, and M. Livny, "Checkpoint and migration of unix processes in the condor distributed processing system", Computer Sciences Technical Report 1346, University of Wisconsin, Madison, WI., 1997.

[5] E. N. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson ,"A survey of rollback-recovery protocols in message-passing systems", ACM Computing Surveys, vol. 34, no. 3, Sept. 2002, pp. 375–408.

[6] P. H. Hargrove and J.C. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters", Proceedings of SciDAC 2006, publication LBNL-60520, June 2006.

[7] S. Osman, D. Subhraveti, G. Su and J. Nieh, "The design and implementation of Zap: A system for migrating computing environments", Proceesings of the 5th Symposium on Operating System Design and Implementation (OSDI 2002), 2002, pp. 361–376.

[8] B. Chan, et al., "Science at LLNL with IBM Blue Gene/Q", IBM Journal of Research and Development, vol. 57, pp. 11:1-11:18, 2013.

[9] G. Zheng, "A scalable double in-memory checkpoint and restart scheme towards exascale", Proceedings of the 42nd IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, 2012.

[10] K. Kharbas, et al., "Combining Partial Redundancy and Checkpointing for HPC", Proceedings of the 32nd IEEE International Conference on Distributed Computing Systems, 2012, pp. 615-626.

[11] E. Feller, J. Mehnert-Spahn, M. Schoettner, and C. Morin, "Independent checkpointing in a heterogeneous grid environment", Future Generation Computer Systems, vol. 28 (1), 2012, pp. 163-170.

[12] J. L. Lawall, and G. Muller, "Efficient Incremental Checkpointing of Java Programs", Research Report no. 3810, National Institute for Research in Informatics and Automatics (INRIA), 1999.

[13] G. Xu, A. Rountev, Y. Tang, and F. Qin, "Efficient Checkpointing of Java Software Using Context-Sensitive Capture and Replay", Proceedings of the ESEC/FSE International Conference, 2007.