Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki
Vrije Universiteit Amsterdam
Faculteit der Exacte Wetenschappen

Bartosz Biskupski
Nr albumu: 181030
Paweł Garbacki
Nr albumu: 181084

Transparent Fault Tolerance for Parallel Java Applications

Praca magisterska
na kierunku INFORMATYKA

Opiekun pracy (Warszawa):
Dr Janina Mincer-Daszkiewicz
Opiekun pracy (Amsterdam):
Prof. Henri E. Bal

Lipiec 2003

Pracę przedkładam do oceny

Data	Podpis autora pracy:

Praca jest gotowa do oceny przez recenzenta

Data	Podpis kierującego pracą:

Streszczenie

In this paper we describe a system that makes parallel Java applications fault tolerant. Our goal was to develop a system which is both efficient and transparent to the programmer. However, we did not aim at providing a solution for all classes of applications. We have concentrated on applications that use the Java Remote Method Invocation mechanism, which is the most commonly used communication mechanism for parallel programming in Java. Similarly, we considered only the class of parallel programs that take some input, compute for a long time, and then yield the result. Such programs, for example, do not interact with users or continuously update files.

The failures that we consider in our system are crash failures such as server halts, broken network connections and for some extent also changes in the network relationships. Our motivation was to avoid having to restart long running parallel computations from scratch after each crash. Therefore, our system makes globally consistent checkpoints of the program's state from time to time. If a processor crashes, the whole parallel program is restarted from the latest checkpoint rather than from the beginning. Since we implemented special replication algorithms for storage of significant data, like checkpoints, our system can resist failures of multiple processors.

Słowa kluczowe

Fault Tolerance, Parallel Programming, Distributed Systems, Checkpoints, Replication, Thread Migration, Java.

Klasyfikacja tematyczna

Computer Systems Organization / Performance of Systems (C.4)

Contents

Chapter 1

Introduction

The Java programming language is becoming a more and more popular platform for parallel computing. Although it is not yet as fast as its C and C++ counterparts, it has several advantages, like portability, flexibility and simplicity. It already dominates in heterogeneous environments, like computational grids, where it has to cope with different operating systems and different network protocols.

Designers of parallel applications frequently ignore fault tolerance, because it requires much additional effort. With the growth in the number of processors in clusters and rapid development of computational grids, which provide even larger numbers of processors, fault tolerance is becoming very important. Consider an application that runs on a thousand CPUs for 12 hours. Even if the mean-time-between-failure of a CPU is 5 years, the chances of one of them crashing can no longer be neglected [1].

Since the Java programming language, despite its wide capabilities, does not provide any mechanisms for fault tolerance, there are two options. One is to have the programmer to explicitly deal with fault tolerance. However, parallel programming is hard enough as it is, and fault tolerance will certainly add even more complexity. The alternative is to develop some mechanisms that will make programs fault tolerant automatically, in a way transparent to programmers. Clearly, the latter option is preferable, although it is in general difficult to implement efficiently.

A failure in non-distributed systems usually results in bringing down the entire application. In contrast, the failure in a distributed system is often a *partial failure*. It happens when one component in a distributed system fails. The failure may affect the proper operation of some components, while at the same time leaving other components unaffected. Well designed distributed systems can automatically detect and recover from a failure without seriously affecting the performance.

In this chapter, we take a closer look at techniques for making distributed applications fault tolerant. We provide some general background on fault tolerance techniques focusing on the checkpointing scheme.

1.1. Replication

The first technique for fault tolerance we address is *replication*. Replication tries to hide the occurrence of failures from other processes. There are several types of replication, but from our point of view the most important are information, time and physical redundancies

[1]In this case the mean-time-between-failure of one of the CPUs is around 44 hours!

(see [14]). With information redundancy, data used by the application are replicated on different nodes. When one of them crashes, data can be restored from backup.

With time redundancy, an action is performed, and then, if needed, it is performed again. This type of replication is especially helpful when the faults are transient or intermittent.

With physical redundancy, extra nodes or processes are added to make it possible for the system to tolerate the loss or malfunctioning of some components. Physical replication can be added on the software or hardware level. We may, for example, have redundant counterpart processes for services that are sensitive for failures. Physical redundancy is an expensive approach, but may provide a high degree of fault tolerance. Process groups are part of the solution for building fault tolerant systems. Having a group of identical processes allows us to mask one or more faulty processes in that group. A group replaces one single process. The replication may be organized by means of either primary-based or replicated-write protocols.

The problem with replication is that having multiple copies may lead to consistency problems. Whenever a copy is modified, that copy becomes different from the rest. Consequently, modifications have to be carried out on all copies to ensure consistency. Exactly when and how those modifications need to be carried out determines the price of replication.

1.2. Checkpointing

Recording a consistent global state, also called a *distributed snapshot*, is a basic technique used for fault tolerance. The checkpointing mechanism is widely used in backward error recovery systems. Each process saves its state from time to time to a stable storage. To recover after a failure it is essential that all the local checkpoints are *globally consistent*. It means that recovery shouldn't bring up the system to an incorrect state. It is also best to recover to the most recent distributed snapshot, also referred to as a *recovery line*.

1.2.1. Independent Checkpointing

The straightforward solution is to allow all processes to record their local state in an uncoordinated fashion. This may however make it difficult to find the recovery line. Discovering a recovery line requires that each process is rolled back to its most recently saved state. If these local states jointly do not form a distributed snapshot, further rolling back is necessary.

The method of taking local checkpoints independent of each other is referred to as *independent checkpointing*. The main disadvantage of this scheme is connected with computing the recovery line that requires an analysis of the interval dependencies recorded by each process when the checkpoint was taken. Such calculations are fairly complex and do not justify the need for independent checkpointing. Coordinated approach described in section 1.2.2 is much less complicated. Additionally, it often appears that the synchronization is not the dominating phase of the checkpointing process.

1.2.2. Coordinated Checkpointing

According to its name, in *coordinated checkpointing* all processes synchronize before writing their state to a local stable storage. The saved state is automatically globally consistent. There are many different distributed coordination algorithms like the two-phase blocking protocol (for details we refer to [14]). Note that to take a consistent checkpoint of a particular process we do not have to synchronize all processes, but only the processes that depend on its recovery. This leads to a notion of an *incremental snapshot*.

As stated before, coordinated checkpoints gain on popularity mainly because of their simplicity. Independent checkpointing and message logging (see section 1.3) schemes are often too complicated and replication too resource consuming.

1.3. Message Logging

Message logging is a technique that is considered to be less expensive than checkpointing but still enables recovery. The idea of message logging is based on resending messages sent after the most recent checkpointed state.

This approach works under an assumption of a *piecewise deterministic model*. In such a model, the execution of each process is assumed to take place as a series of intervals in which events take place. An event is, for example, executing an instruction or sending a message. Each interval in the piecewise deterministic model starts with an nondeterministic event as receiving of a message. From that moment the execution of a process is completely deterministic. An interval can be reconstructed in a completely deterministic way when we know the starting nondeterministic event.

1.4. Summary

Fault tolerance is an important issue for long running distributed applications. A system is considered to be fault tolerant if it can continue to operate in the presence of errors.

There are different ways of solving the problem of faults. The most straightforward solution relies on the replication of resources such as processes. The cost of recovery is in this case limited to the necessary minimum, but the overhead of updating replicas during normal execution may be expensive.

The checkpointing scheme moves the costs connected with system failure to the recovery phase. In general, there are two types of checkpoints. In the independent approach every process captures its local state without communicating with the others. Problems start before the recovery. The consistent global image should be composed of the local checkpoints. This is a nontrivial task, especially when we want to find the most recent consistent global checkpoint (referred to as a recovery line). Because of these limitations a much simpler coordinated scheme seems to be a better choice.

Message logging is yet another way for providing a distributed system with fault tolerance extensions. This approach requires however special assumptions according to the system behavior and is at least as complicated as independent checkpoints. Message logging may be a good idea when we have limited space for storing checkpoints.

1.5. Contributions and Overview of the Thesis

Our task were the design and implementation of a system that makes distributed Java applications fault tolerant without serious loss of performance.

The main contributions of this work are:

- design of a Java byte code transformer which uses advanced compilation time analysis techniques such as method call graph analysis,

- global coordination algorithm that takes into account the specificity of the checkpointing process and network topology,
- fault tolerant Remote Method Invocation (RMI) [12] which can recover from a system crash in a completely transparent way,
- mechanisms which allow to initiate checkpoints also inside remote calls,
- implementation and efficiency tests of all these mechanisms.

The rest of this paper is organized as follows (in brackets we specify the author of particular chapter). Chapter 2 (Bartosz Biskupski) describes the mechanisms used for making the process of taking local checkpoints transparent to a user and the execution environment. Distributed aspects of our framework are presented in chapter 3 (Paweł Garbacki). Fault tolerant RMI implementation is described in chapter 4 (Bartosz Biskupski). The following chapter 5 (Paweł Garbacki) reveals the arcana of the subsystem responsible for handling checkpoints initiated inside remote method calls. Similar projects are presented in chapter 6 (Bartosz Biskupski, Paweł Garbacki). Chapter 7 (Bartosz Biskupski, Paweł Garbacki) analyzes the results of efficiency tests. Chapter 8 (Bartosz Biskupski, Paweł Garbacki) concludes this paper.

The implementation of particular subsystem was the responsibility of the author of the chapter that describes it.

Chapter 2

Local Checkpoints

2.1. Problem statement

A local checkpoint is a state of a user application running on one node, which allows to restore the application at the point at which the checkpoint was taken. Transparent checkpointing ideally means that no explicit code has to be provided to save or restore an application's state. However, there is a difficulty in making transparent checkpoints. A local checkpoint has to preserve three states:

- Program state: which is the bytecode of the object's classes.
- Object state: the contents of the bytecode of the object's class.
- Execution state: a Java object executes in one or more JVM threads. Each thread has its own private *Java stack* (see figure 2.1). A Java stack stores *frames*. The JVM creates a separate frame for each invoked method and destroys it when a method completes. Each frame contains its own local variables, operand stack on which JVM saves partial results of computations and *program counter* (in short: PC), which indicates the next instruction to be executed.

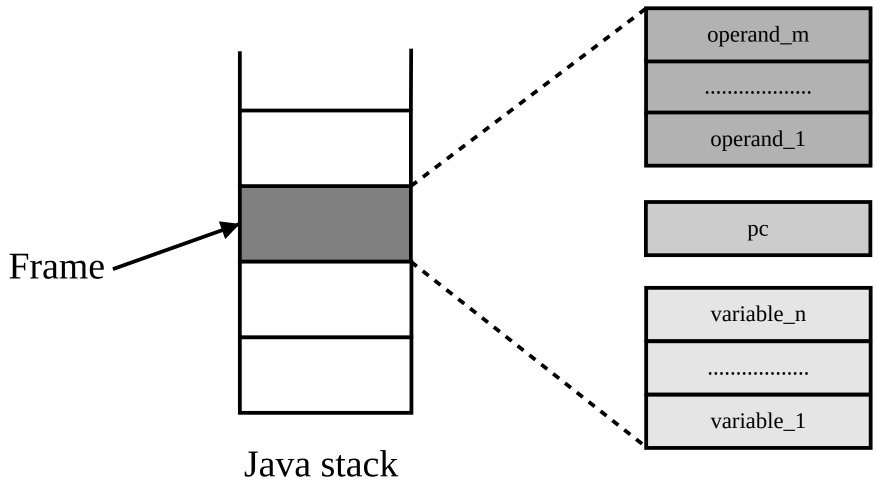


Figure 2.1: Java stack

The Java language supports preserving of the first two states. By defining a custom class loader, necessary classes can be downloaded. Object state is preserved using serialization mechanisms. The only problem concerns execution state capturing and reestablishing, due to the lack of built-in mechanisms in Java. An application, on a bytecode level, has access only to the current method frame and therefore there is no straightforward way to preserve the whole Java stack.

To deal with this issue several techniques have been proposed (see chapter 6.1). We have chosen the technique used in the Brakes [4] project with some new improvements and modifications. This algorithm uses a Java post-compiler, which transforms bytecode by inserting additional code blocks that do the actual capturing and reestablishing of the current thread's state. The basic idea of this state capturing mechanism is to save the current method frame (local variables and program counter), then immediately return from that method and repeat this process for all methods on the method's stack. State reestablishing is accomplished by reconstructing the user application from scratch. Each method's frame is restored (one by one) and the already executed code is skipped by the additional code inserted by the code transformation algorithm. After the reconstruction user application continues its execution at the point where it was interrupted.

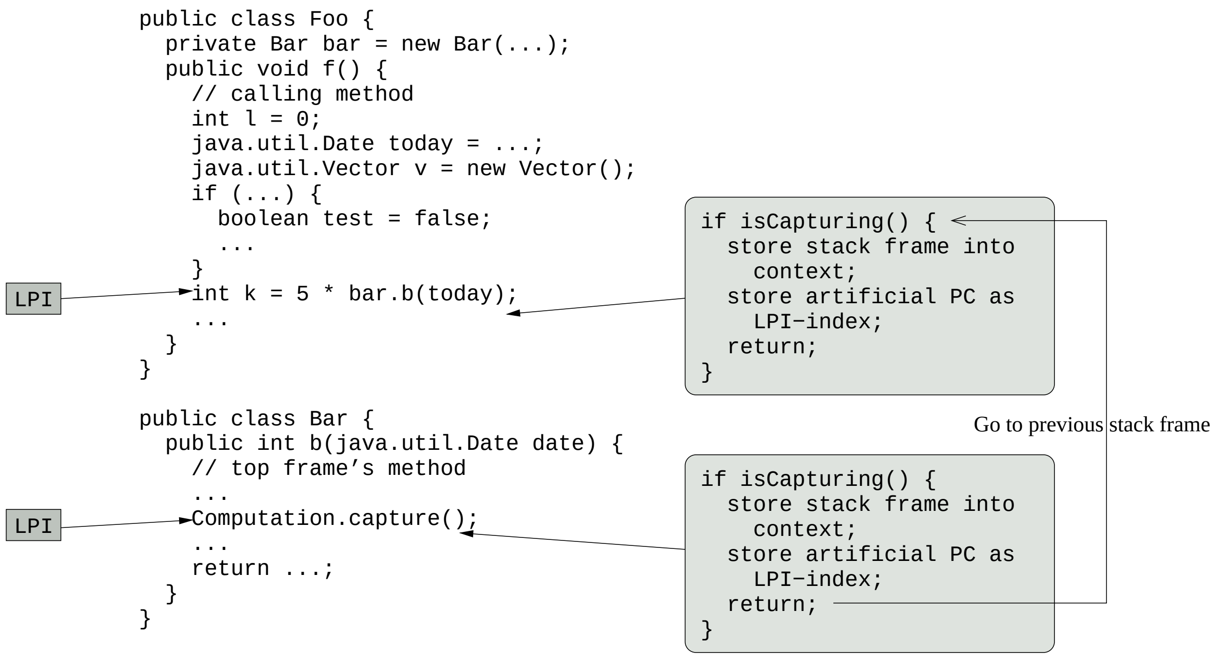
2.2. Code Transformation Algorithm

This section describes the code transformation algorithm. Each computation is associated with a separate `Context` object into which its execution state is captured and from which its execution state is later reestablished. A computation is also associated with a number of boolean flags, which indicate the current execution mode. A computation can be in one of three different modes of execution: running (normal execution), capturing (before serialization) and restoring (after deserialization). These flags can be influenced by the programmer. The programmer has to use the internal `capture` (or `make checkpoint`) method in order to set the capturing flag and initiate state capturing.

Inside every method additional code that saves the method's frame is inserted (see figure 2.2)[1]. Every method invocation is followed by a code block that checks if the computation is in a capturing state (its capturing flag is set) and if so, it saves in the context its operand stack, local variables, last performed invoke-instruction (LPI), and then immediately returns to the previous method on the stack (`return` instruction removes the current frame from the stack). This process is repeated until the thread is back where it started (on the bottom of its method's stack). At that time, the thread's context contains the whole trace the thread followed.

In the example from figure 2.2, `f()` method in the `Foo` class invokes `b()` method from the `Bar` class which in turn invokes our system's internal `capture` method. The `capture` method sets the capturing flag and initiates state capturing. When the application leaves the `capture` method, the inserted code block is executed. It stores `b()`'s local variables and `capture` method's index in the context and returns from that method. The application continues from the next instruction in the `f()` method which is the inserted code block. Again, it saves `f()`'s method frame and returns to the previous method. Finally, the user application ends when it returns from the main method.

[1]Examples come from [4]

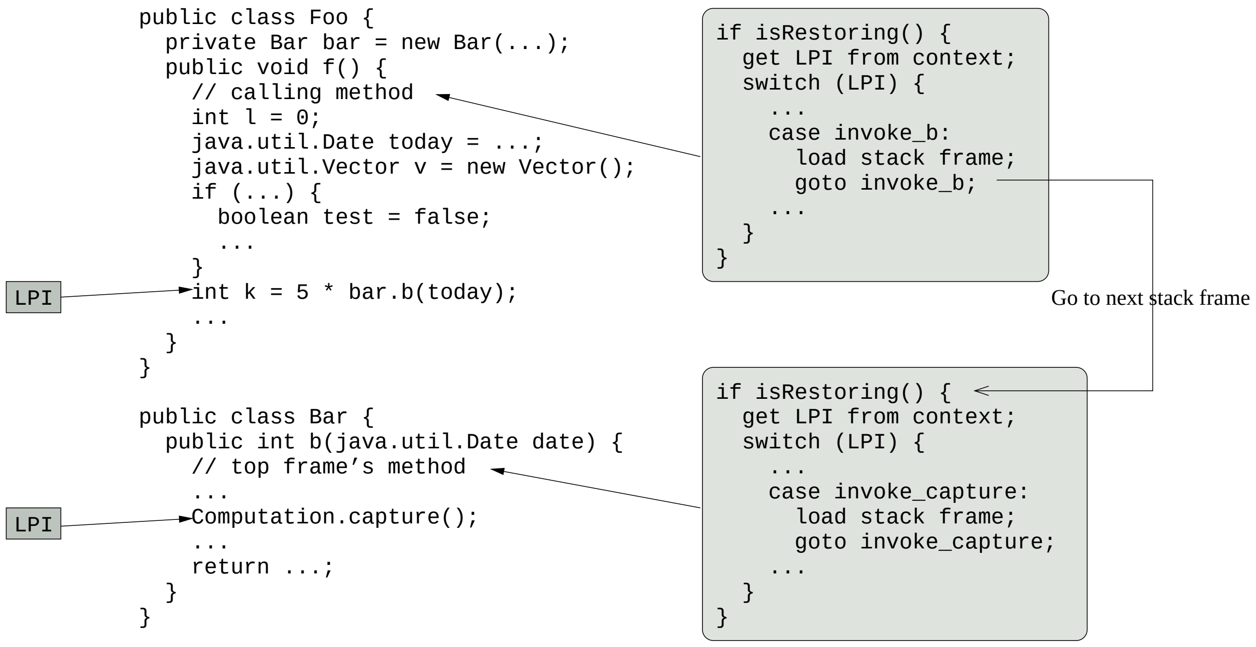

```
public class Foo {
  private Bar bar = new Bar(...);
  public void f() {
    // calling method
    int l = 0;
    java.util.Date today = ...;
    java.util.Vector v = new Vector();
    if (...) {
      boolean test = false;
      ...
    }
    int k = 5 * bar.b(today);
    ...
  }
}

public class Bar {
  public int b(java.util.Date date) {
    // top frame's method
    ...
    Computation.capture();
    ...
    return ...;
  }
}
```

```
if isCapturing() {
  store stack frame into
    context;
  store artificial PC as
    LPI-index;
  return;
}
```

```
if isCapturing() {
  store stack frame into
    context;
  store artificial PC as
    LPI-index;
  return;
}
```

Figure 2.2: State Capturing


```
public class Foo {
  private Bar bar = new Bar(...);
  public void f() {
    // calling method
    int l = 0;
    java.util.Date today = ...;
    java.util.Vector v = new Vector();
    if (...) {
      boolean test = false;
      ...
    }
    int k = 5 * bar.b(today);
    ...
  }
}

public class Bar {
  public int b(java.util.Date date) {
    // top frame's method
    ...
    Computation.capture();
    ...
    return ...;
  }
}
```

```
if isRestoring() {
  get LPI from context;
  switch (LPI) {
    ...
    case invoke_b:
      load stack frame;
      goto invoke_b;
    ...
  }
}
```

```
if isRestoring() {
  get LPI from context;
  switch (LPI) {
    ...
    case invoke_capture:
      load stack frame;
      goto invoke_capture;
    ...
  }
}
```

Figure 2.3: State Reestablishment

Accordingly, inside every method additional code that restores the method's frame is inserted (see figure 2.3). When the thread is to be restored, a new thread is started and the restoring flag is set. The first stack frame is restored (local variables and operand stack are reestablished) and the thread jumps to this method's LPI. This causes invoking the next method on the method's stack. The situation is repeated until the thread reaches the method that started capturing. The capturing flag is then cleared and the thread continues its execution like nothing happened.

In the example from figure 2.3, `f()`'s method frame is restored (local variables like `l`, `today` and `v` are restored) and the application jumps to the invocation of `b()` method. The `b()` method is invoked and similarly its method frame is restored and the execution is continued at the invocation of `capture` method. Since the internal `capture` method changes the execution state to the normal execution (clears the restoring flag), the inserted code block that follows `capture`'s invocation is not executed.

Figure 2.4 presents a flow diagram for example source code, where the order of taken actions is given in circles and squares respectively for state capturing and reestablishing.

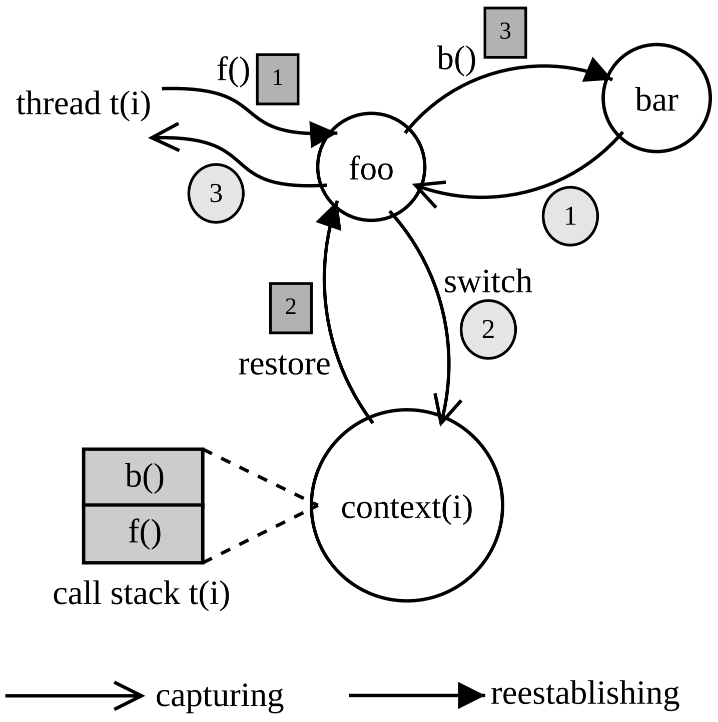


Figure 2.4: Flow diagram for state capturing (numbers in circles) and reestablishing (numbers in squares)

2.3. Methods Call Graph

One of the improvements we have made to this algorithm over the original one from the Brakes project is building the method's call graph to modify only methods and method invocations that can lead to execution state capturing. Algorithm 1 presents the process of building the methods call graph.

Algorithm 1 Building methods call graph

```
1: procedure buildCallGraph(Class rootClass, Method rootMethod)
2: begin
3: if captureSet.contains(rootClass, rootMethod) then
4:   return
5: end if
6: captureSet.add(rootClass, rootMethod)
7: for each (C, M) which can invoke (rootClass, rootMethod) do
8:   buildCallGraph(C, M)
9: end for
10: for each superclass C which rootClass extends do
11:   if C contains rootMethod then
12:     buildCallGraph(C, rootMethod)
13:   end if
14: end for
15: for each interface I which is implemented by rootClass do
16:   if I contains rootMethod then
17:     buildCallGraph(I, rootMethod)
18:   end if
19: end for
20: end
```

The algorithm builds the set of all <class, method> pairs which invocation can lead to state capturing. It begins from the method which starts the actual state capturing (in our case it is `capture` or `make checkpoint`). Then, it finds all methods in all classes which can invoke that method and recursively starts from those methods. However, during compile-time the real object's class is not know — it can be either the declared one or one of its subclasses. Similarly, if a method is invoked on the interface, the real class of the object is not known — we only know that the requested class implements (directly or not) that interface. Therefore, all super classes and implemented interfaces which contain processed method have to be added to the set. Finally, the transformer rewrites only invocations of methods which exist in the generated set. The detailed comparison of our improved transformation algorithm with the original one from the Brakes project is the subject of section 7.5.

2.4. Static Fields Serialization

The Java object serialization mechanism does not support serialization of static fields in classes, because such fields do not correspond to any specific object. However, our checkpointing mechanism requires that all objects have to be preserved. In our case it would be convenient to serialize static fields as well. Therefore, we developed a Java bytecode post-compiler, which adds this feature to our system.

Our transformation algorithm searches for static fields in each user class and adds two methods: one for saving all static fields into the context and one for restoring them from the context. Finally, it adds to the user's main class a method that calls these methods in all user classes that contain static fields. After this transformations it is possible to preserve both object state and all static fields.

2.5. Limitations

Unfortunately, it is not possible to achieve full transparency in state capturing. Firstly, not all objects can be preserved. There are many non-serializable Java classes, like classes responsible for accessing sockets and files. There are also some classes, which are location dependent, like RMI stubs, which have hard coded IP address and port number of remote object. We addressed this problem in our system by introducing *Persistent RMI* which is a location independent RMI and is described in details in chapter 4.

Secondly, transformation algorithms cannot handle native methods, since they extract thread execution state at the bytecode level.

Thirdly, although possible, our algorithm can not deal with state capturing during the execution of an exception handler. The major difficulty here is dealing with the `finally` statement of a `try` clause because the return address from a subroutine is not known during compile time.

Finally, should Java API libraries (i.e. from `java.*` packages) be modified or not? In most cases it is not necessary, since library calls do not initiate state saving by themselves. The exception to this are library calls that result in a callback to the application code. For example when using graphical packages, like `Swing` or `AWT`, events cause callbacks in user application code. We decided not to transform these classes what means that the programmer should not initiate state capturing inside any event handler used for the standard Java API libraries (i.e. handler for a mouse click).

Chapter 3

Distributed Checkpoints

3.1. Problem statement

In the previous chapter we addressed the issue of taking *local checkpoints*. *Local* means that all the threads we want to serialize are running within the same address space. We extended the applicability of our solution to the class of distributed applications by using the *coordinated checkpointing scheme* (see section 1.2.2). As its name suggests, in coordinated checkpointing all the threads running on different machines have to synchronize before writing their state to the stable storage. The advantage of global coordination is that the saved state is automatically *globally consistent* (see chapter 1). However, global coordination in the nondeterministic distributed environment is not a simple problem, especially when we are thinking in terms of efficiency and reliability. Let us point some of the problems connected with coordination of Java processes:

- portability as main design goal limits us to standard Java communication mechanisms. Remote Method Invocation (RMI) — the most popular way of remote process communication does not offer group communication facilities such as *broadcasting* or *multicasting*;

- we have to deal with a situation when either multiple threads initiate the state capturing exactly at the same time or the time interval between the first and last request is non-negligible;

- our goal is to provide extensions for fault tolerance. It also concerns (of course to some extent) the coordination phase. We should take also into account a possibility of a crash during coordination.

In the further part of this chapter we describe the components of our global checkpointing subsystem starting from a distributed barrier.

3.2. Centralized Approach

In this section we present our solution for the global coordination problem. We need a kind of a *distributed barrier* that will assure consistency of the captured state. The Java programming language offers thread synchronization by means of monitors. The `synchronized` keyword in the signature of a method guarantees that it will be executed by not more than one thread at the same time. It is much harder to synchronize processes in the distributed environment. The concerns usually center around performance and how to re-coordinate if something breaks.

The straightforward solution for thread coordination problem in the distributed environment is to implement the remote object with a synchronized `barrier` method. Such an object can then be exposed through the RMI interface. Threads (running on different machines) that invoke the `barrier` method can be suspended until all of them declare readiness. The node where the coordination service is installed is called a *coordinator node*.

This simple algorithm has some drawbacks. First of all, one node (the coordinator) is outstanding. It has to gather synchronization requests from all other processes. Because of this fact the coordinator node is often referred as a *bottleneck node*.

Another, more subtle problem may be considered when talking about specific types of applications. The distance between nodes in the distributed system is usually not the same. The well designed system may make use of network topology to increase efficiency and reliability. For instance, if some of the components are located on the other end of the world, it would be nice if we could group them together and then, using one or two method calls, synchronize this group with the rest.

Last but not least, the centralized scheme does not make use of specificity of our task. Checkpoints are very rarely initiated at the same time at different nodes. This observation can be used for optimizing the synchronization process.

Till now we concentrated on the disadvantages of the centralized scheme. Of course there are advantages as well. This algorithm is optimal when considering the number of messages sent or remote method calls. Another, often ignored, but quite important issue is the simplicity. Simple usually means easy to implement.

3.3. Neighborhood

To improve efficiency and reduce synchronization delay we introduced the configurable component that allows the user to tune up our system according to the network topology. *Neighborhood* function defines the rules of control messages routing inside our system. As we mentioned before in this chapter, Java RMI does not support a group communication. In the situation when broadcasting has to be done at the software level, it is reasonable to map the physical network interconnection scheme to the software communication model.

Figure 3.1 shows some basic network topologies and exemplary neighborhood functions. Broadcast is always initiated at the node 0. Messages are sent along the arrows, first to the node with the smallest identifier.

The diagram 3.1(a) shows the simplest situation — physical network topology is mapped directly onto the logical scheme. It takes 4 time units to complete the broadcast under assumption that sending a message takes one time unit. The sending phases look as follows:

1. 0 sends message to 1
2. 0 sends message to 2, 1 sends message to 3
3. 2 sends message to 5, 1 sends message to 4
4. 2 sends message to 6

In the diagram 3.1(b) every two nodes are connected to each other. The message sending phases defined by the neighborhood are:

1. 0 sends message to 1

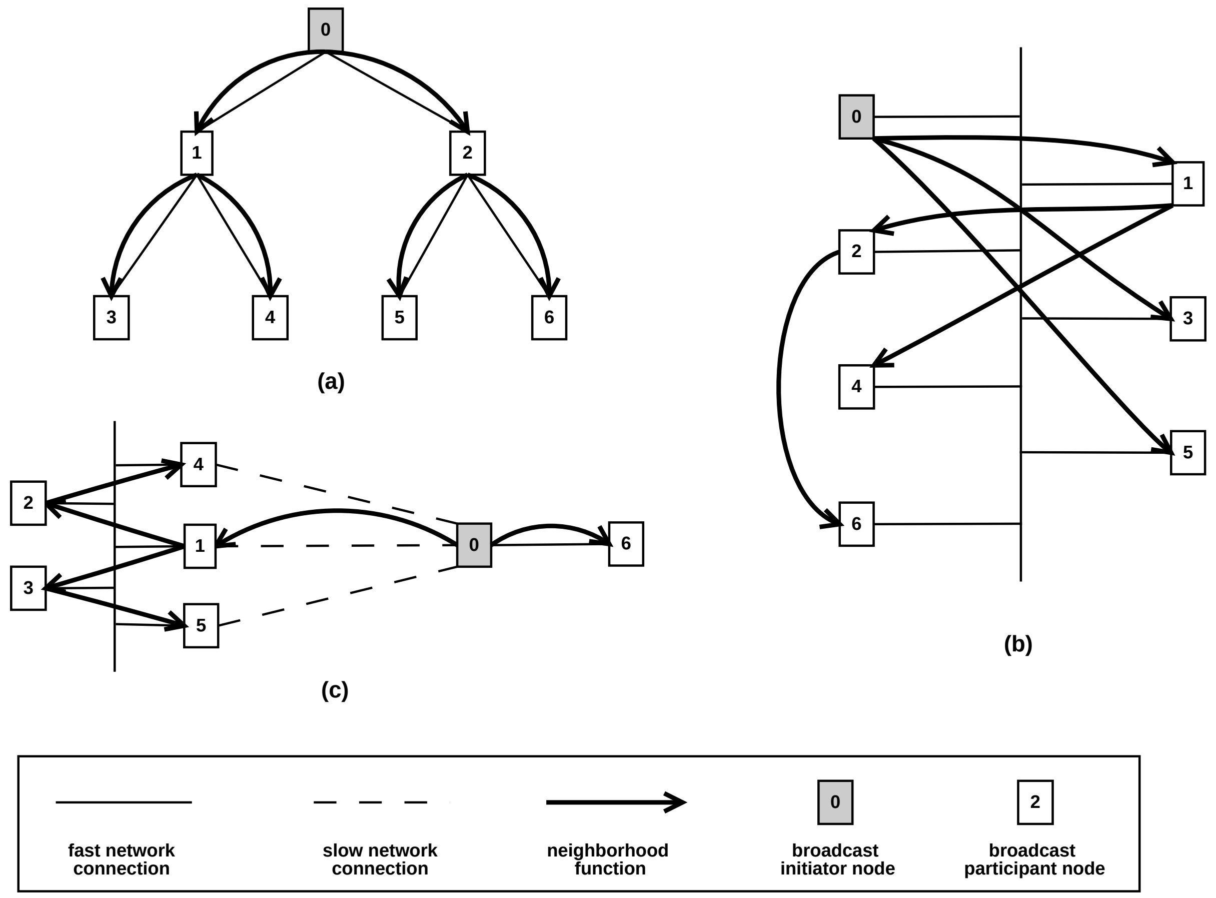


Figure 3.1: Network topologies and neighborhood. Straightforward network interconnection to neighborhood mapping (a), Ethernet-like bus (b) and multi-cluster scheme (c).

2. 0 sends message to 3, 1 sends message to 2

3. 0 sends message to 5, 1 sends message to 4, 2 sends message to 6

Finally, figure 3.1(c) presents the most complicated variant of two fast local networks connected with slow links. There are following message sending phases:

1. 0 sends message to 1

2. 0 sends message to 6, 1 sends message to 2

3. 1 sends message to 3, 2 sends message to 4

4. 3 sends message to 5

Note that broadcast could be finished in 3 instead of 4 time units. Node 0, which is idle during third and fourth phase, could be used for sending the message to node 5 in the third phase. However, this variant requires sending one more message over the slow link, that may introduce bigger overhead than the additional phase.

The intuitive meaning of the neighborhood function should be clear now. We introduced the notion of neighborhood for the purpose of the next section where we describe the global barrier synchronization. To assure the correctness of the coordination algorithm, we have to limit the set of all possible neighborhood functions to a class that satisfies the following conditions:

- the neighborhood function defines a graph that is a tree,
- the neighborhood function is a function of three parameters — actual node number, root node number and total number of nodes.

The first point should be clear — we are using our virtual interconnection structure for broadcasting. There is no point in having cycles in the graph. The second sentence states that all children nodes of a particular node should be determined by its own identifier, identifier of the root node and total number of nodes in the graph. Note that this condition requires global ordering of nodes.

3.4. Distributed Coordination Algorithm

Because of the limitations of the centralized algorithm and specificity of checkpointing subsystem we decided to design own distributed coordination algorithm that best fits our needs. As mentioned in the previous section there are some observations we may make to improve the robustness of our solution. These are the design goals of our algorithm:

- make use of different network topologies,
- take into account that time interval between the first and the last checkpoint request may be long and use this time for useful synchronization work,
- tolerate some faults that occur during *coordination phase*. Coordination phase is the period between the first participant declares its readiness for global coordination and the last participant is resumed after coordination.

According to chapter 2 all local threads on every node are synchronized locally before initiating the distributed checkpoint. This situation is presented in figure 3.2.

Algorithm 2 is our proposition for barrier synchronization that coordinates processes involved in the global state capturing process.

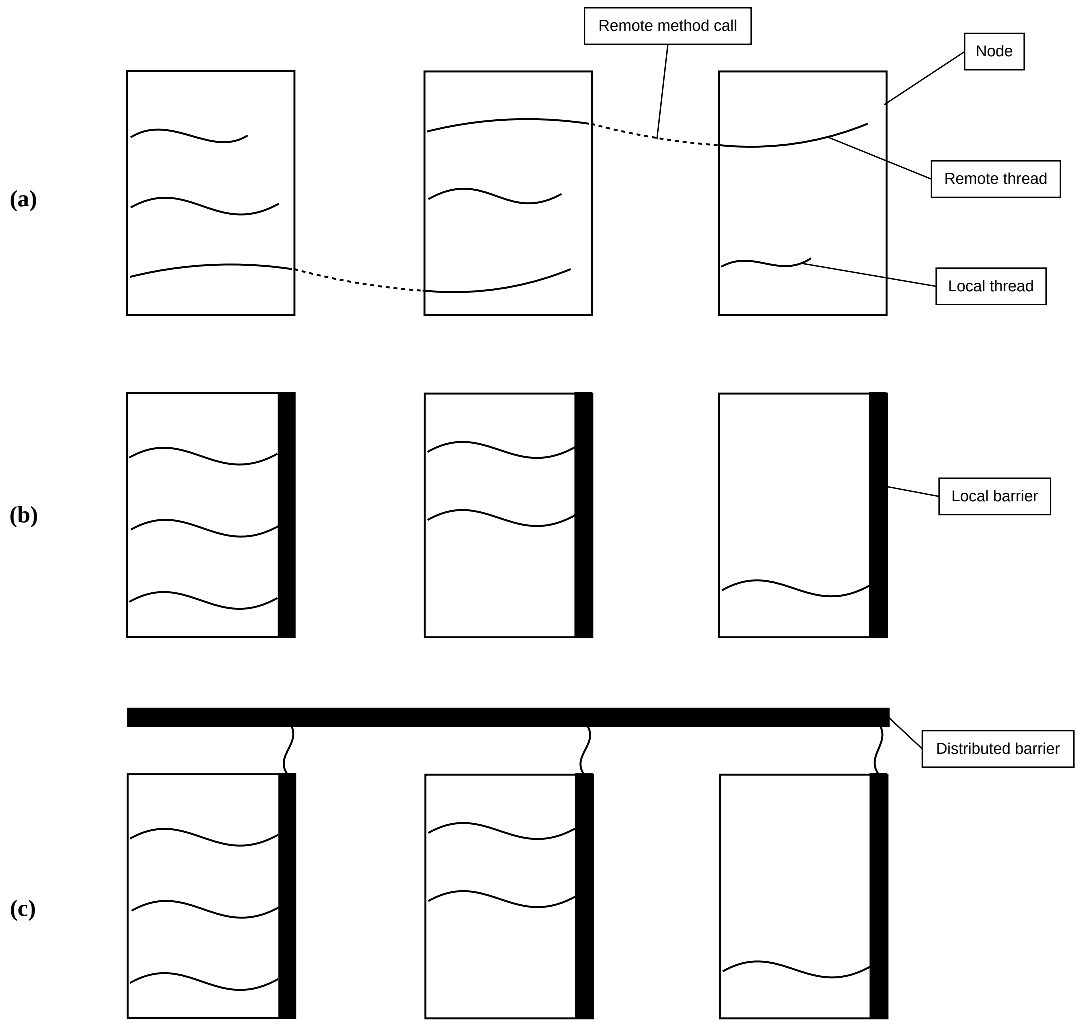


Figure 3.2: Local and distributed barrier. Situation before synchronization (a), local threads synchronized on every node (b) and global barrier (c).

Algorithm 2 Global barrier synchronization

```
Require:
    n_nodes — number of all nodes
    node_no — actual node number
Ensure:
    processes on all nodes reached the barrier
 1: if got message SYNCHRONIZATION_REQUEST(coordinator) then
 2:   wait for SYNCHRONIZATION_CONFIRMATION(global_coordinator)
 3: else
 4:   coordinator := node_no
 5:   for all neighbor in neighbors(node_no, node_no, n_nodes) do
 6:     local_coordinator := process SYNCHRONIZATION_REQUEST(node_no) at
        neighbor
 7:     if local_coordinator.getPriority() > coordinator.getPriority() then
 8:       coordinator := local_coordinator
 9:     end if
10:   end for
11:   if coordinator == node_no then
12:     global_coordinator := node_no
13:   else
14:     wait for SYNCHRONIZATION_CONFIRMATION(global_coordinator)
15:   end if
16: end if
17: for all neighbor in neighbors(node_no, global_coordinator, n_nodes) do
18:   process SYNCHRONIZATION_CONFIRMATION(global_coordinator) at neighbor
19: end for
```

There are few things that should be explained about algorithm 2. The `neighbors(actual_node, root_node, total_nodes)` function computes the set of `actual_node` node descendants when the root node of the broadcasting structure is `root_node` and there is `total_nodes` nodes in general.

The coordinator of the whole process is elected dynamically. The synchronization phase may be initiated by multiple processes by means of broadcasting the `SYNCHRONIZATION_REQUEST` message, but it is a task of one node to confirm global synchronization by sending the `SYNCHRONIZATION_CONFIRMATION` message. Messages are parametrized with the identity of their creator. This fact brings another observation — broadcast trees may vary, depending where the initial message was created. `global_coordinator` variable contains the identifier of the elected coordinator that will finally send the `SYNCHRONIZATION_CONFIRMATION` message.

The **process** `message` **at** `node` is a complex construct that consists of message sending and processing at destination `node`. The processing may return a value.

To understand why the algorithm guarantees that there is only one coordinator elected let us now look at the message processing phase. Algorithm 3 describes it in detail.

Note that broadcast of message `message` is continued only if receiver at the actual node did not get the message with higher priority yet (checked in line 2). By this condition we are trying to limit the number of redundant messages sent.

If the message we got was sent by a process with a higher priority than any of previously received messages, the coordinator is determined by the neighbors of actual node.

Algorithm 3 Message processing

```
Require:
    message — message to be processed
    n_nodes — number of all nodes
    node_no — actual node number
Ensure:
    returns coordinator of the subtree defined by neighborhood function and rooted in
    actual node
 1: coordinator := message.getCreator()
 2: if got message SYNCHRONIZATION_REQUEST(local_coordinator) and
    local_coordinator.getPriority() > coordinator.getPriority() then
 3:   return local_coordinator
 4: else
 5:   local_coordinator := coordinator
 6:   for all neighbor in neighbors(node_no, coordinator, n_nodes) do
 7:     new_coordinator := process message at neighbor
 8:     if new_coordinator.getPriority() > local_coordinator.getPriority()
        then
 9:       local_coordinator := new_coordinator
10:     end if
11:   end for
12:   if local_coordinator <> coordinator then
13:     return local_coordinator
14:   else
15:     wait for application running on local node to invoke barrier
16:     if in the meantime got message SYNCHRONIZATION_REQUEST(local_coordinator)
        such that local_coordinator.getPriority() > coordinator.getPriority()
        then
17:       return local_coordinator
18:     else
19:       return coordinator
20:     end if
21:   end if
22: end if
```

At last, our algorithm should synchronize all the nodes, so at some point (line 15) we have to wait for the application part, we are responsible for, to invoke global coordination request.

Algorithms 2 and 3 try to implement some kind of heuristic for minimizing the number of messages sent and *synchronization delay.* By synchronization delay we mean the difference between the time when the last process invoked the barrier function and the time when the last process left the barrier.

The efficiency depends not only on the neighborhood function, but also on the barrier notification scheme. For example, when there is one process that is much faster than the others, the SYNCHRONIZATION_REQUEST can be delivered to every node before any of them will even invoke the barrier function. The coordinator will be known already in the initial phase of the algorithm and we will not have any redundant messages at all. When the processes at all nodes run more or less at the same speed we may have race conditions.

The synchronization delay is dependent on the neighborhood function probably even more than redundant messages overhead. When we use a ‘smart’ one that groups together nodes with similar characteristics according to barrier synchronization initiation times, we can achieve good results.

3.5. Communication Scheme

One thing that needs to be specified in order to fully understand the algorithms described in the previous section is the communication scheme. To use the advantages of parallel broadcasting we decided to base our system on *parallel synchronous* messaging. Parallel synchronous means that communication with neighbors is synchronous, but messages are sent to different neighbors in parallel. The initiator is blocked until all its descendants accepted the message. Senders are implemented as separate threads and are reused across different synchronization phases. The sender’s life cycle is showed in figure 3.3.

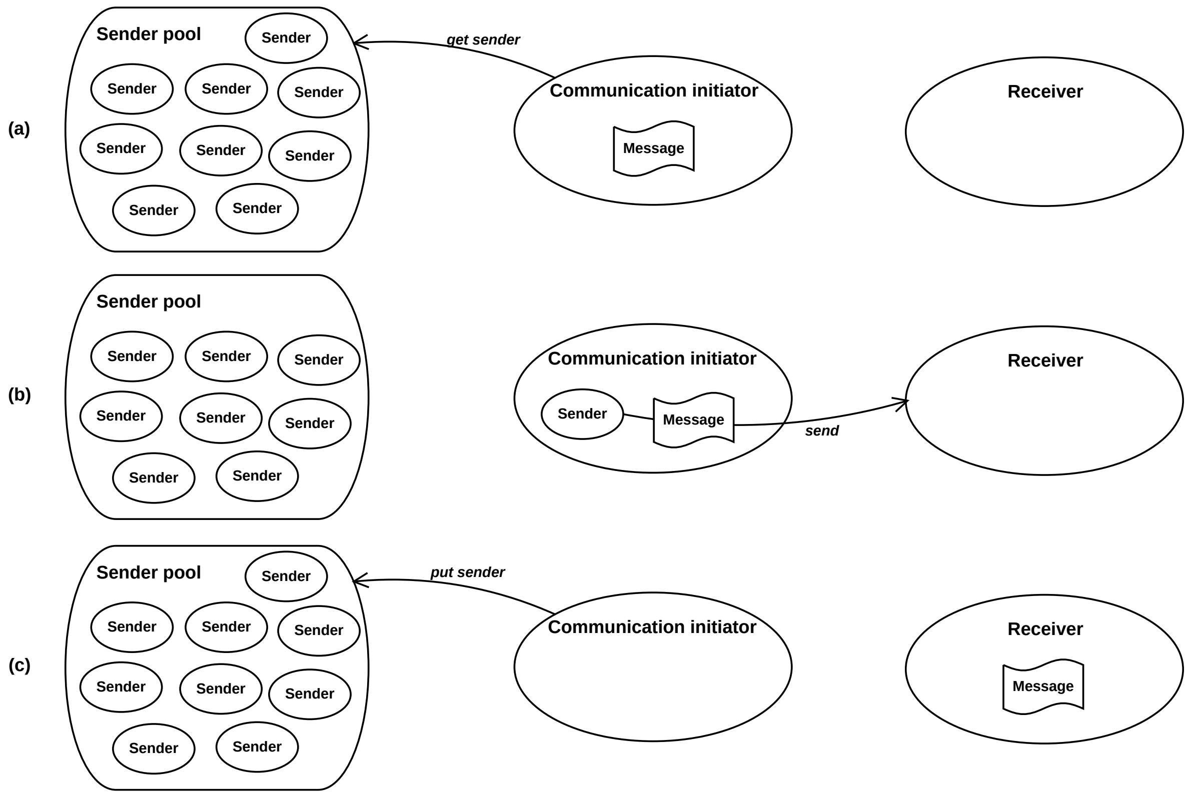


Figure 3.3: Senders. Before sending Message Initiator has to obtain a Sender from Sender pool (a), Sender takes care of communication (b), Initiator returns Sender to Sender pool after Message was delivered.

3.6. Taking Coordinated Checkpoints

We have described the distributed synchronization algorithm. Now we will discuss its application to the coordination of distributed checkpoints. Taking globally consistent checkpoints in our distributed environment is nothing more than adding synchronization barriers to the scheme described in chapter 2. Of course these synchronization points have to be added in appropriate places.

First, we are synchronizing all the processes running on different nodes *just after* they initiated the state capturing. Just after means that all local threads on all machines have to be suspended on top of their Java stacks.

The second coordination takes place *just before* starting the application. Again, just before means after all threads rebuild their Java stacks and are about to carry on with the suspended computations. In fact the system is in exactly the same state when the first and second synchronization take place.

At first sight two synchronization phases can be reduced to only one. However, a scenario in which a thread first captures its stack, then synchronizes and finally saves the checkpoint into stable storage can lead to inconsistencies. One barrier is not enough when we want to capture the state inside synchronized methods. If we skip the coordination before taking the checkpoint, we can end up with a checkpoint that sees more than one active process inside synchronized method. On the other hand, omitting barrier after state capturing may result in the race condition and prevent some processes from fully restoring their Java stacks.

State capturing in remote method calls requires two additional synchronizations. For details see chapter 5.

3.7. Storage

We use the term *stable storage* in regard to a component responsible for persisting checkpoints. Stable storage is a subsystem whose task is to guarantee availability of supervised data even if some of its elements fail. We want to have our system not only software but also hardware faults resistant. To fulfill this requirement we decided to use replication.

Again, in order to allow the end user for tuning up the system the *node mapping* function was introduced. The idea is similar to the neighborhood. However, computing descendants in a broadcast tree is replaced by determining nodes where checkpoint should be stored.

Figure 3.4 shows the arcana of data storing phase.

We may think about stored data in terms of local checkpoint. Note that data are always saved on the local node first. This decision will be explained in the next section.

Data retrieving is presented in figure 3.5.

If a node is down we just skip it and try to contact the next one.

We implemented two different storing systems. One is based on the file system, the other keeps all data in the random access memory.

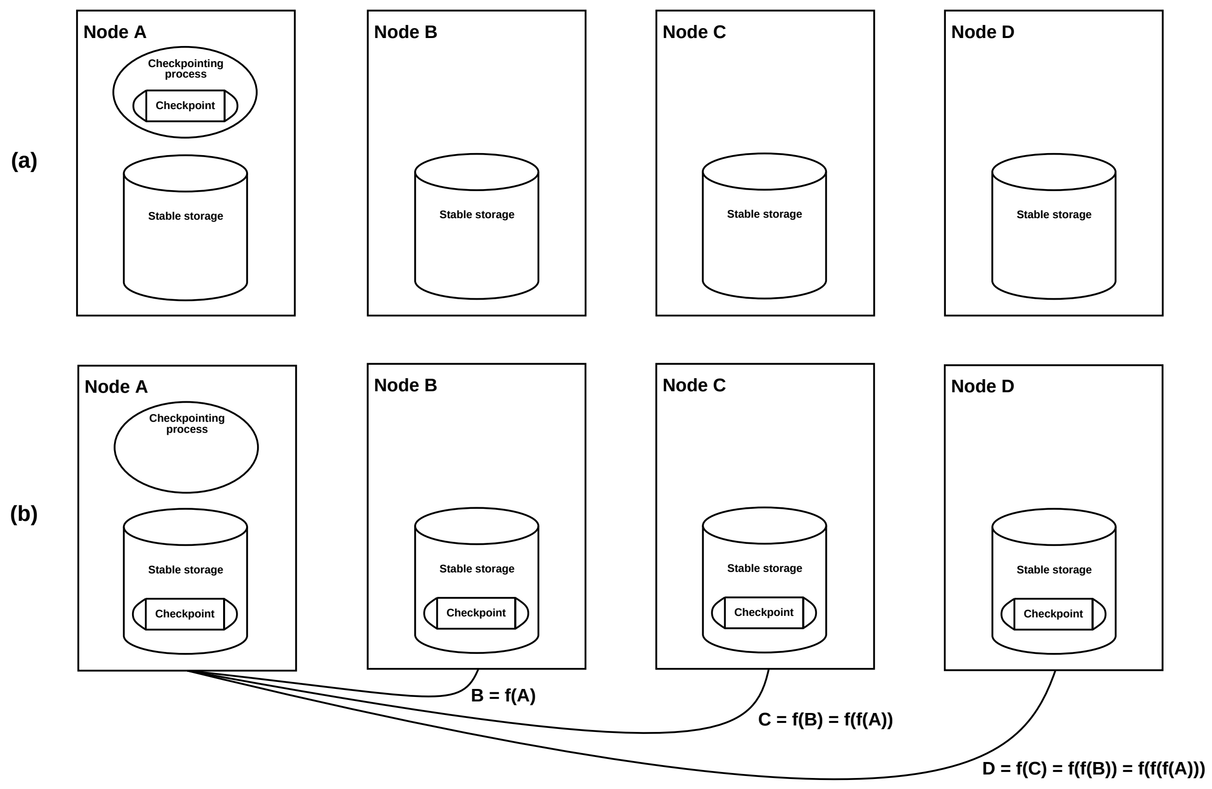


Figure 3.4: Data storing. Checkpointing process has to decide where to place the Checkpoint data (a), Node Mapping function determines locations of backups (b).

3.8. Faults During Checkpointing Phase

Our task was to provide a framework for fault tolerance in the distributed environment. Highly reliable system should be aware of the fact that some of its components may also be a subject of different faults. In our situation, when processes can declare readiness for global checkpoint at arbitrary point in time, it is even more important. Table 3.1 shows which kind of faults our system can deal with.

The column and row headers of the above table may require explanation. Phrase "before first/second barrier" describes a set of faults that occurred before the first node reached the first/second barrier. "After first/second" barrier characterizes all the faults that occurred after the last node left the barrier synchronization function.

$+$ means that system will be able to recover if fault occurs during this phase,

$-$ means that we do not give any guarantees according to success of the recovery process.

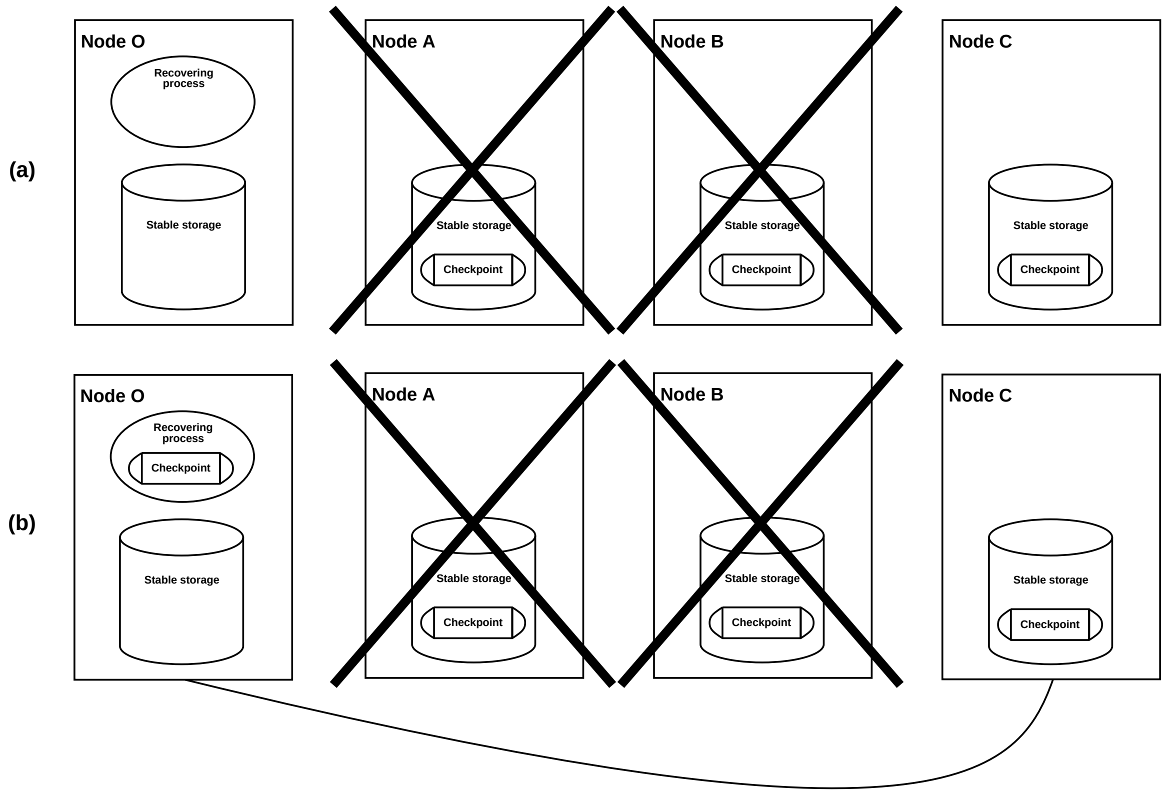


Figure 3.5: Data restoring. Recovering process does not have a copy of data it needs in local Stable Storage (a), data is retrieved from non faulty backup node (b).

Table 3.1: Fault tolerance coverage regions

	1	**2**	**3**	**4**	**5**
Support level	+	+	−	−	+

1: Before first barrier begin
2: First barrier begin – first barrier end
3: First barrier end – second barrier begin
4: Second barrier begin – second barrier end
5: After second barrier end

The problem with phases marked with minus signs lies in our policy according to checkpoints' storage. We do not use any versioning system. Only the newest checkpoints are stored. Somewhere between first and second barrier coordinators running on every node take a local checkpoint. If the fault occurs when some of the processes have already stored new state in the stable storage and some of them not, there is no consistent data available.

3.9. Distributed Fault Detection and Recovery

In this section we describe the recovery algorithms we used in our system. Algorithm 4 presents the arcana of the recovery process.

Algorithm 4 Distributed recovery

```
Require:
    n_nodes — number of all nodes
    node_no — actual node number
Ensure:
    processes on all nodes recovered to the latest consistent global checkpoint
 1: coordinator := node_no
 2: for all node in 0..n_nodes - 1 do
 3:   if node is not a faulty node then
 4:     local_coordinator := process FAULT_DETECTED(node_no) at node
 5:     if coordinator.getPriority() < local_coordinator.getPriority() then
 6:       coordinator := local_coordinator
 7:     end if
 8:   end if
 9: end for
10: if coordinator == node_no then
11:   for all node in 0..n_nodes - 1 do
12:     if node is faulty node then
13:       backup := node_mapping(node)
14:       process RECOVERY_REQUEST(node) at backup
15:     end if
16:   end for
17:   for all node in 0..n_nodes - 1 do
18:     if node is not a faulty node then
19:       if node <> node_no then
20:         process RECOVERY_REQUEST(node) at node
21:       end if
22:     end if
23:   end for
24:   process RECOVERY_REQUEST(node_no) at node_no
25: end if
```

The recovery phase, like the barrier, needs a global coordinator. We use the similar approach and elect the coordinator dynamically. FAULT_DETECTED message carries an identifier of the node where it was created. RECOVERY_REQUEST message instead is labeled with the identity of a node that should be recovered.

One thing that may be unclear in presented algorithm is the need for distinguishing between faulty and non-faulty nodes. We assume that errors do not occur too often one after another, so to reduce the costs of recovery we drop the old checkpoints before restarting the application. This condition requires recovering faulty nodes before rolling back the others to the last consistent checkpoint.

The algorithm of processing the `RECOVERY_REQUEST` message is straightforward. We just read the checkpoint from stable storage and restart the application on the appropriate node. The `node_mapping` function, as explained before, points out the node where the backup of part of the application we want to recover is located.

Now it should be clear why it is wise to keep a copy of the latest checkpoint locally. Backup storage should be used to recover processes that crashed on remote machines. Properly working parts of the application can resume computation on the same machine they were before. In this case the recovery requires hardly any network communication — all data are stored locally.

Our system contains a component which is responsible for replicating checkpoints on different machines. For simplicity this functionality was skipped in the description of algorithm 4.

There are four situations that initiate recovery process. Firstly, the user application can explicitly invoke the scanning procedure that will check which nodes are down. This action is usually taken if the application part running on one node can not contact another node. Secondly, our framework can notice breakdowns of some elements when it performs global barrier synchronization. Thirdly, crash of a remote object may be discovered by fault tolerant RMI subsystem. Finally, there is a dedicated thread running on every node which task is to check periodically whether all remote processes are up and running.

The components responsible for resurrection of the application from the checkpoint are called *activators* (see figure 3.6).

3.10. Summary

In this chapter we revealed distributed aspects of our checkpointing scheme. We presented algorithms we used and motivated crucial decisions. Some of our ideas are general purpose solutions, but most of them are based on the specificity of distributed checkpointing model. In the following chapters we will present extensions to the model described here.

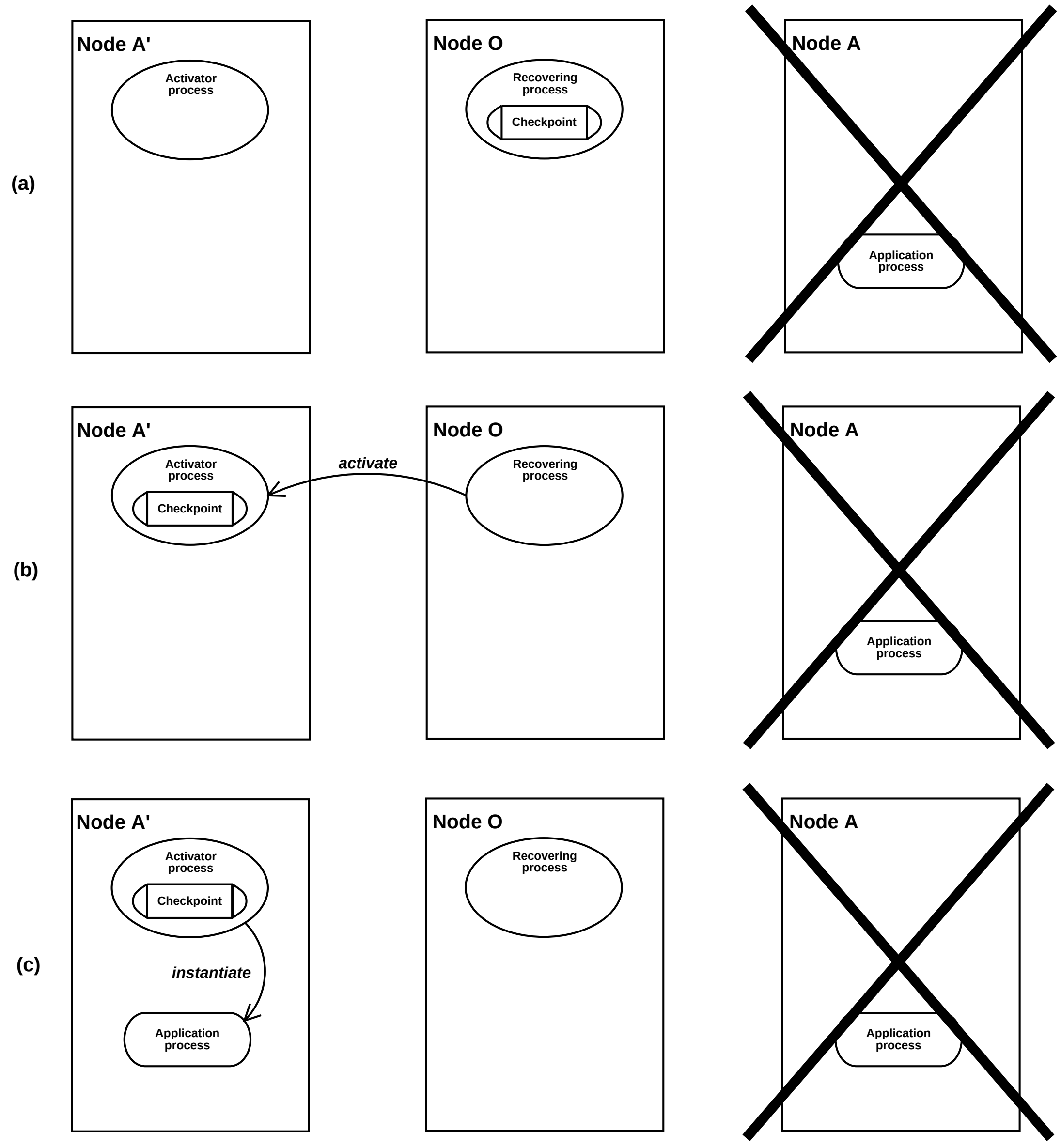


Figure 3.6: Activators. Checkpoint of the crashed node is in possession of the recovering process (a), Checkpoint is sent out to Activator on backup node (b), Activator instantiates application on local node (c).

Chapter 4

Fault Tolerant Remote Method Invocations

4.1. Introduction to Remote Method Invocation mechanism

Distributed systems require that computations running in different address spaces, potentially on different hosts, be able to communicate. For a basic communication mechanism, the Java programming language supports sockets, which are flexible and sufficient for general communication. However, sockets require an application programmer to design and implement protocols to encode and decode messages for communication between the client and the server. The design of such protocols is inconvenient and can be error-prone.

The Java programming language provides an alternative for sockets, which is the Remote Method Invocation (RMI) mechanism. The main advantage of this mechanism over plain sockets is the comfort of its use. It gives the programmer an illusion of calling just normal methods and does not engage the programmer into complicated communication details. Therefore, it became the most widely used mechanism for distributed computing in Java. In this section we briefly describe RMI. For a complete description of RMI, see Sun's RMI specification [12].

RMI applications often consist of two parts: a server and a client. A typical server application creates a number of remote objects, makes references to those remote object accessible, and waits for clients to invoke methods on those remote objects. A typical client application gets a remote reference to one or more remote objects in the server and invokes methods on them. RMI provides a mechanism by which the server and the client communicate and pass information back and forth.

From the programmer's point of view, each remote object provides an interface containing all methods that can be called by clients on that remote object. The programmer does not have to be aware of details of a remote method invocation, but simply calls methods on that interface. What actually happens is that the programmer calls methods on a local *stub* object implementing that interface, which plays the role of a proxy between the client and the server. The stub object establishes a connection with the remote object on the server, passes serialized method's parameters and returns a deserialized server's response. The stub object signals all errors which may occur during a remote method call by throwing an exception, which should be caught in the application.

Before a client application can call methods on a remote object, first it has to somehow obtain a remote reference to that object. It can use one of two mechanisms. An application can register its remote objects with RMI's simple naming facility, the *rmiregistry*, or the application can pass and return remote object references as part of its normal operation.

Figure 4.1 shows a distributed application that uses the RMI registry to obtain a reference to a remote object. The server calls the registry to associate a name with a remote object's reference. The client looks up the remote object's reference by its name in the server's registry and then invokes a method on it.

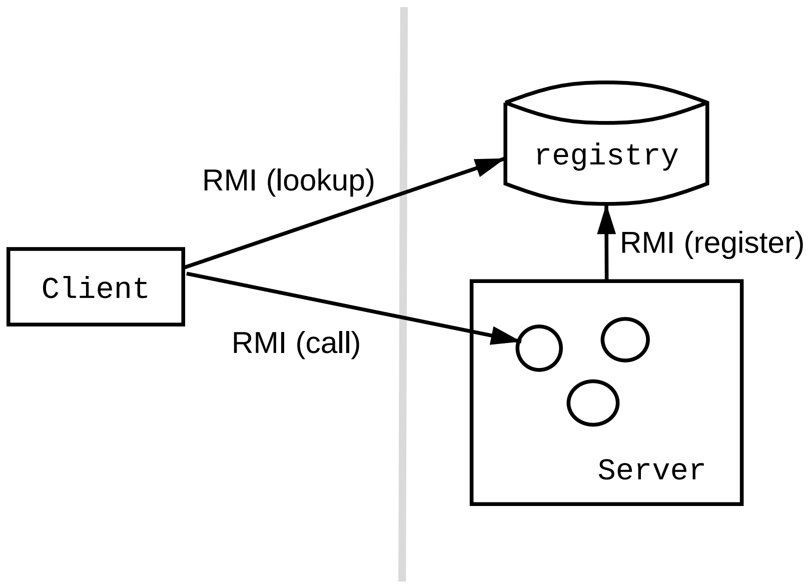


Figure 4.1: Remote Method Invocation

4.2. Problem Statement

Java Remote Method Invocation (RMI) is the primary model for distributed computing in Java. However, while Java RMI promotes access transparency and location transparency of remote servers to clients, it does not provide fault tolerance mechanisms to render faults transparently to the application. Instead, the occurrence of a fault in the system is exposed to the application, requiring application programmers to provide additional mechanisms to ensure correct, reliable and highly-available operation, even in the presence of faults.

A typical way to create a custom remote object is to extend the `UnicastRemoteObject` class. When such an object is instantiated, it is exported to a specified port number and since that time it is available for remote clients. Clients call methods on local *stub* objects to access remote objects. Each stub holds a remote reference object responsible for the whole low-level communication with the remote object. A stub class is generated using `rmic` Java tool. Stub objects are serializable and therefore can be sent to other machines in the network and used by them as long as remote object is working. Each remote object is uniquely identified by three parameters that are assigned to its stub's remote reference during instantiation:

- IP address,
- port number,
- unique ObjectId, which is assigned during instantiation, so each instance of the same class is uniquely identified.

All these properties make remote objects location dependent. When a machine on which the remote object is located fails, then all stubs corresponding to that object owned by the clients will become incorrect and their use will cause an error. Therefore, we developed mechanisms for Fault Tolerant RMI.

4.3. Our Solution

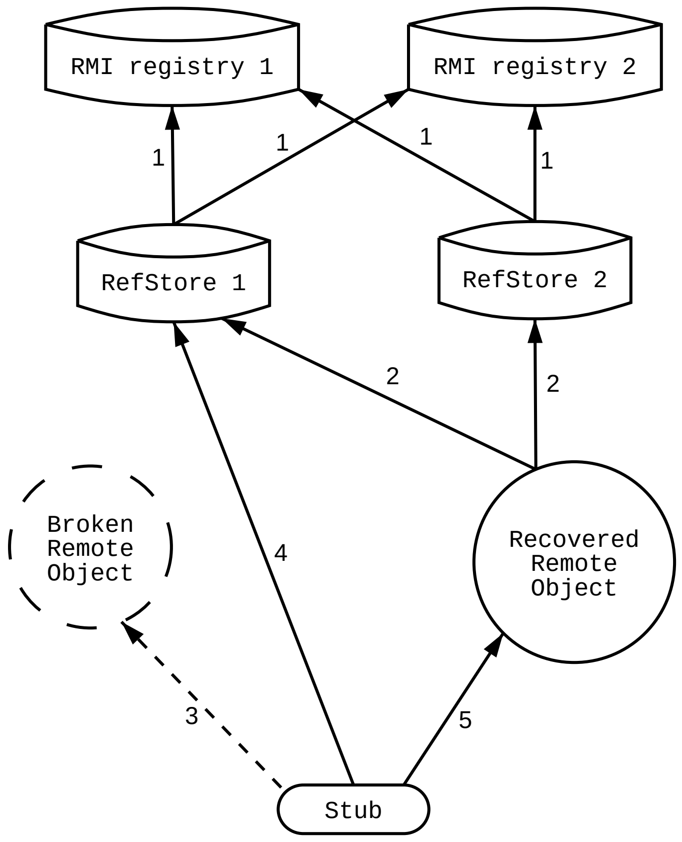


Figure 4.2: Fault Tolerant RMI

Our system was designed to have no single point of failure. In order to achieve this, components of our system use replication and checkpointing mechanisms.

4.3.1. Replicated RMI Registry Server

The first layer of our Fault Tolerant RMI forms a replicated RMI registry server (see figure 4.2). It is replicated for fault tolerance in the *read-one, write-all* fashion, which means that clients (in this case the clients are RefStore servers described in the next section) have to propagate changes to all replicas, but they can read from only one of them (A.Tannenbaum and M. van Steen in their book [14] present several replication algorithms). When one replica crashes, the rest will still serve clients. When either a new or a recovered replica is started, it will automatically update itself by retrieving data from other replicas. The locations of all replicas are stored in the environment variables propagated to all nodes when the application is started.

4.3.2. Replicated RefStore Server

The essence of our Fault Tolerant RMI lies in the use of a replicated server, which we called *RefStore*. It is responsible for storing pairs of <old (incorrect) remote reference, new remote reference>. Alike our RMI registry, it is also replicated for fault tolerance in the *read-one, write-all* fashion. In order to be found, each replica of the RefStore server registers itself in the replicated RMI registry server (1). Our system provides a `PersistentUnicastRemoteObject` class which extends the `UnicastRemoteObject` class and is used for creating fault tolerant remote objects. Instances of this class are stored in a checkpoint together with the rest of the user application. Each time they are deserialized from the checkpoint due to the failure of the original host, a new remote reference, containing new remote object's location, is sent to all replicas of the RefStore server (2).

4.3.3. Fault Tolerant stubs

Our system provides also a Java bytecode post-processor for transforming stub class files. After transformations stubs do not raise an exception when they cannot locate remote object (3), but they initiate the recovery process. When the application is being recovered, stubs obtain a new remote reference from one of the replicated RefStore servers (4) and after the recovery they use it for accessing the remote object (5). The transformation algorithm adds to each stub's class file an additional `ping` method that tries to get in touch with the remote object and in the case of failure it fetches the new remote reference. The ping method is invoked during the recovery process on every stub object in the user application.

4.3.4. Checkpointed RMI registry

User applications which adopt Remote Method Invocation model typically use the RMI registry for exchanging remote references to their remote objects. Since the node on which standard Java RMI registry runs may fail, thus causing faults in the whole distributed application, user applications should use a fault tolerant RMI registry. However, it may not be the replicated RMI registry, because it has to be strictly consistent with the application. Consider the following situation. An application makes a checkpoint, registers a remote object in the RMI registry and after that it crashes. It is recovered from the last checkpoint and once again it tries to register an object in the RMI registry but it will not succeed because the RMI registry already has that object registered. Therefore, we developed the RMI registry which is checkpointed together with the whole user application. Our system provides classes that are equivalent to the standard Sun's `Naming` and `RegistryImpl` classes.

Our RMI registry implementation uses our system's checkpointing facilities. It extends `PersistentUnicastRemoteObject`, it uses our stub class transformer for making its stub objects fault tolerant, thus after a crash it registers itself in the RefStore server. Moreover, it registers itself in the replicated RMI registry server, so our `Naming` class can easily find it without knowing its physical location.

4.4. Wrappers

Our system provides 'wrapper' classes for each of the standard RMI class, like `UnicastRemoteObject`, `Naming`, `LocateRegistry`. Wrapper classes have the same interface as the original ones, however they cooperate with our system to make the user application fault tolerant. Standard Java classes are replaced with wrapper classes transparently to the programmer by the transformation algorithm.

4.5. Summary

Our RMI extensions make a distributed application, which uses RMI mechanisms, tolerant for faults in a transparent way. An attempt to invoke a method on a remote object located in a broken process does not anymore result in an exception propagated to the application, but it initiates a recovery process. Since the broken process with its remote objects can be recovered on any host, stub objects held by clients will fetch new remote references pointing at new remote objects locations.

Each component of our RMI system is either replicated or checkpointed, thus there is no single point of failure. Moreover, since none of the actions taken by our RMI extensions influence the application, our fault tolerant RMI is also transparent.

It should be stressed that our solution is not fully tolerant for faults. Theoretically it may happen that all replicas crash and the system will not be able to function. In practice, however, a sufficient number of replicas should assure high enough fault tolerance.

Chapter 5

Checkpoints in Remote Method Calls

5.1. Introduction

The solution for making Java applications fault tolerant, which we presented in previous chapters is quite flexible. However, there is a class of programs for which the rules user has to obey are too strict. One of the most annoying limitations is the restriction, that the checkpoint request can not be invoked while residing inside a remote method call.

In this chapter we present a mechanism for serializing the execution state of a distributed Java thread. The notion of a distributed thread is defined later in this chapter, but intuitively the previous sentence states, that now the checkpoint can cross the borders of one virtual machine.

The solution we are going to present is fully integrated with the distributed checkpointing framework described in chapter 3.

5.2. Distributed Threads

When using a *distributed control flow* model like Java RMI one should take into account *shift of semantics*. There are many examples that show differences between Java RMI and "normal" Java programming. Let us mention the separation between class and interface of a remote object, the pass-by-copy semantics of non-remote arguments to a remote method invocation and the inherently more complicated failure modes of remote method invocation. These are all results of the shift of semantics that may lead to unexpected results, if the programmer did not take these differences into account.

In this chapter we are studying a very particular kind of shift of semantics — *shift of thread semantics*. We may experience this problem when we try to adapt a local Java program for execution in a distributed environment. A thread is often defined as a sequential flow of control within a single address space. For us this definition is not convenient. The notion of distributed thread has already been introduced in the Alpha distributed real-time OS kernel at CMU [3]. D. Jensen has identified the notion of distributed thread as a powerful basis for solving distributed resource management problems. We borrow the definition of distributed thread from this work:

> A distributed thread is the locus of control point movement among objects via operation invocation. It is a distributed computation, which transparently and reliably spans physical nodes, contrary to how conventional threads are confined to a single address space.

We may simplify this definition and describe the distributed thread as a logical sequential flow of control that may cross physical node boundaries. As shown in figure 5.1, the distributed thread is physically implemented as a concatenation of local threads performing remote method invocations when they transit JVM boundaries.

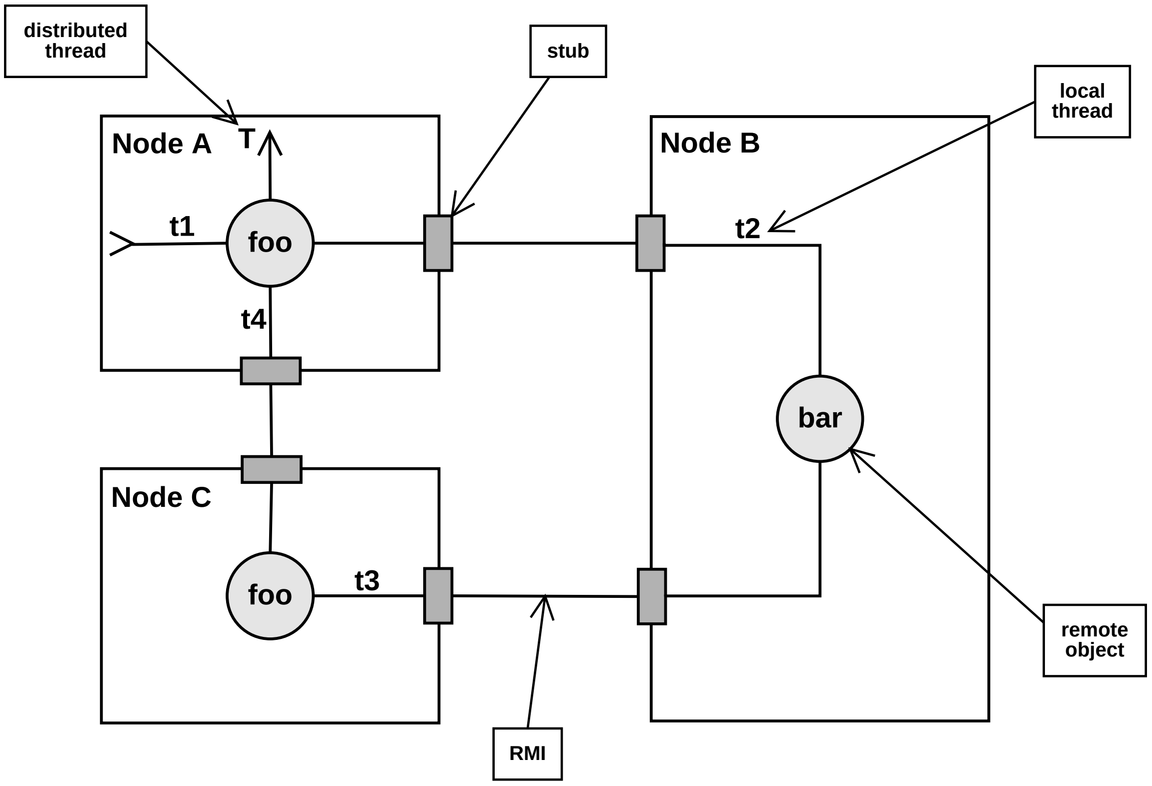


Figure 5.1: Distributed Thread

Distributed threads offer a general concept for distributed computation entity. Therefore we decided to add extensions for serialization of a distributed execution state on per distributed thread base.

5.3. Distributed Identity

In a local execution environment the JVM thread identifier offers a unique reference for a single computation entity. Distributed threads execute as flows of control that may cross physical node boundaries. Once the control flow crosses system boundaries a new JVM thread is used for continuing execution. Java itself does not offer any interfaces that allow us to recognize two local threads as parts of the same distributed thread. We extended Java programs with the notion of *distributed thread identity*. Propagation of globally unique identities allows for identification of local computations as parts of the same distributed computational entity.

Points we took into consideration while designing our distributed identity implementation:

- no execution time overhead is introduced when we are not performing remote method calls;
- the only overhead is connected with the remote method call event. It means that our solution should not have any influence on remote method execution time measured from the moment of invoking the local thread responsible for executing method's body till returning from this method;
- no execution time overhead is introduced when the remote thread's control flow can not lead to state capturing and this behavior can be predicted by method call graph analysis.

The identity is assigned to a distributed thread at its creation time. This behavior is integrated with our thread instantiation engine. It means that the programmer is not aware of her threads being modified. The distributed identity itself carries the following information:

Creator identifier Identity of the node where the distributed thread was created. More precisely — the node, where the oldest (with regard to creation time) local thread contained within the distributed thread was started;

Unique local thread identifier Identifier, that is unique across all the threads created on the same node. This is introduced only to guarantee global scope uniqueness of thread identities that is achieved by combining creator identifier with local thread id.

The internal structure of a distributed thread identity is illustrated in figure 5.2. The creator identifier points in this case to a local thread `t1`. The identity is propagated with remote method calls that result in creation of local threads `t2` and `t3`. The dictionary that maps local threads to corresponding identities is distributed. It means that node `B` holds the information only about the thread `t2` and node `C` only about the thread `t3`.

5.4. Distributed Context

In chapter 2 we defined the *context* object as a container for saved data. When we are working with local threads only, one context object for one thread is enough. In a distributed environment having one context per distributed thread is not enough unless we have one centralized database of contexts. The problem lies in the Java serialization mechanism. Serialization semantics guarantees "one step" consistency of references. It means, for instance, that deserializing the same object twice will result in instantiation of two different Java objects. This limitation requires saving state of all local processes in one common checkpoint.

In a distributed environment we can not base our solution on passing the context objects between nodes. This action requires serialization of contexts that may, as explained before,

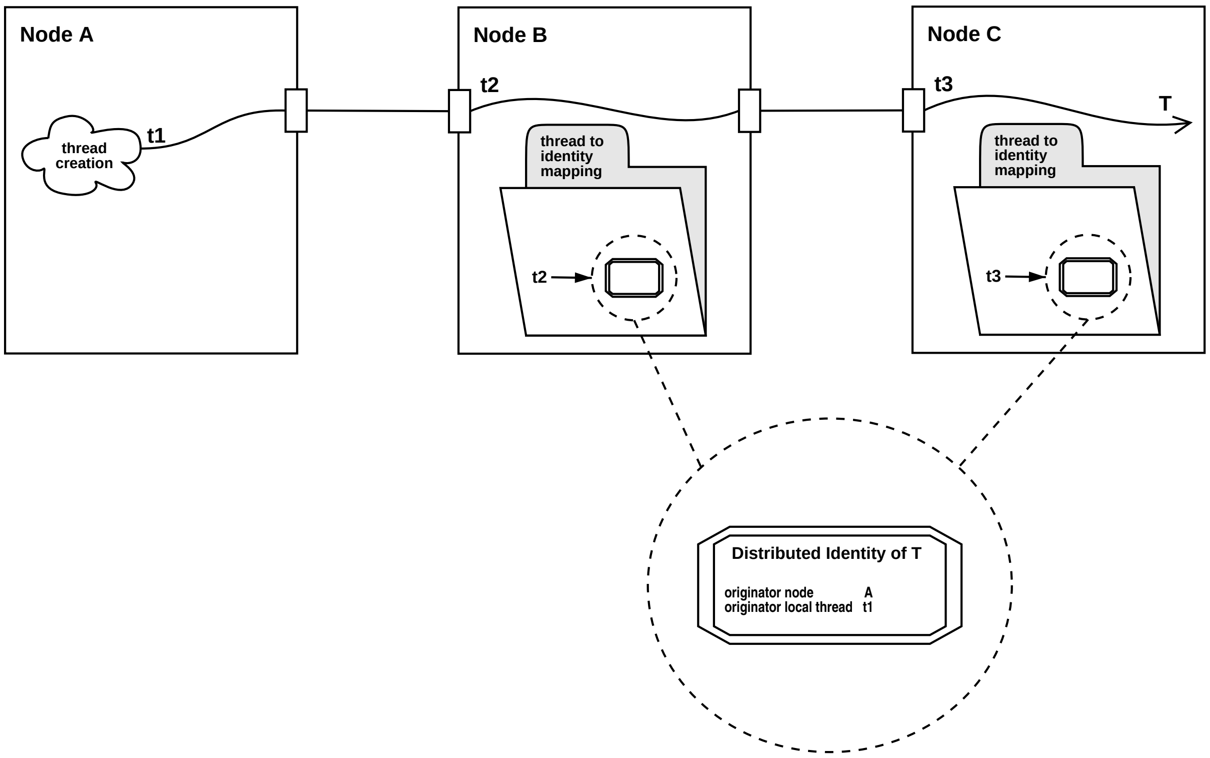

Node A
thread creation
t1
Node B
t2
thread to identity mapping
t2
Node C
t3
T
thread to identity mapping
t3
Distributed Identity of T
originator node A
originator local thread t1

Figure 5.2: Distributed Thread Identity

destroy referential integrity. To deal with this problem in an elegant way we introduce a notion of *distributed context*. Distributed context may be simply defined as a set of all "ordinary" context objects of local threads that belong to one distributed thread. For better understanding see figure 5.3.

As stated before, one of the main design goals of our framework is to minimize the overhead connected with execution of the inserted byte code. Chapter 2 describes the quite complicated and time-consuming process connected with starting a new thread. Several data structures have to be created to associate a context with a thread. Now imagine, that these steps have to be repeated every time we invoke a remote method. The overhead may be non-negligible.

Our solution uses so called *lazy binding*. It means that the remote thread registers in our middleware only when the checkpoint is initiated. The corresponding computation and context objects will not be created unless state capturing is requested. No modifications in the Java stack saving or restoring scheme are required.

The crucial difference in state capturing and restoring of the remote thread lies in the context object's storing policy. If the context belongs to a remote thread, it is stored in a dictionary data structure under a key representing the thread's distributed identity. By doing so we are able to identify local contexts that belong to one distributed thread.

5.5. Distributed Thread's State Capturing

After explaining our task and introducing new terminology we can describe the process of capturing the state of a distributed thread. The idea is quite simple, but the correct and

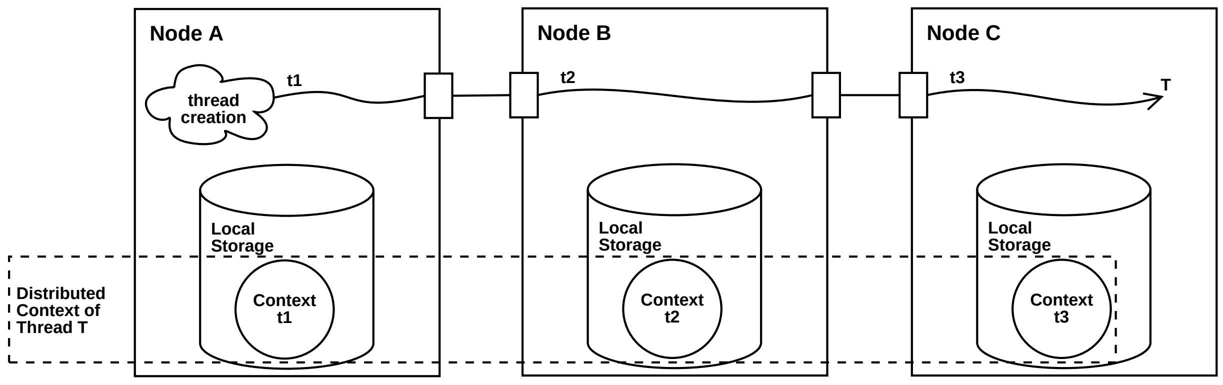


Figure 5.3: Distributed Context

robust implementation poses many difficulties.

State capturing is done in phases. Every phase corresponds to making a checkpoint of one of the local threads that are parts of the same distributed entity. The stack frame saving method is the same as the one described in the chapter dedicated to local checkpoints. The difference lies in the actions performed on the remote method call and exit.

Our algorithm behaves differently regarding whether state capturing was initiated or not. To make things more clear these cases will be described separately.

During normal execution the semantics of distributed control flow is as follows:

1. the distributed identity is created for every thread registered in the checkpointing subsystem, but is used only by the entities that perform remote method calls;
2. the distributed identity, once assigned to a thread, stays the same during its lifetime;
3. every remote method call is labeled with a distributed identity of the invoker;
4. the distributed thread identity is stored in the internal data structures of our system until execution of a remote method is finished.

Problems start when we initiate state capturing inside a remote method call. Checkpoint manager residing on every node synchronizes with other nodes only if all local threads declared readiness. Now local threads may be suspended waiting for the results of a remote call. Remote threads have to be coordinated on top of their (remote) stacks, because of the same reasons as local threads (synchronized methods) — for details see chapter 3. The solution is to contact the originator node and confirm state capturing initiation at the remote location. One question remains — how to identify the creator? The pointer to a creator node is included in the remote identity not only to guarantee uniqueness but also for being used as a key in the nodes' database.

Now consider "nested" remote method calls. In this context nested means one remote call invoked inside another remote method call. We need a mechanism to inform local threads on the path of the remote invocation that the distributed thread, they are part of, is in the state capturing phase. Contacting explicitly all the nodes on the path of the remote call is not possible since the distributed identity contains only the identity of a creator node. We solved this problem with the help of the Java error mechanism. Before returning from the remote method the `isSwitching` flag indicating state capturing is checked. If it is set, instead of returning the result, we throw a special type of exception. This exception can then be captured on the node where remote call was initiated and additional steps may be taken in order to prepare the local thread for taking the checkpoint. We decided to use exceptions as a *distributed* `isSwitching` flag for two reasons. Firstly, no overhead is introduced during normal execution. Secondly, exceptions can be almost transparently integrated with existing implementations (see section 5.6).

The graphical illustration of the described process is presented in figure 5.4.

Distributed state reestablishing uses a similar concept as capturing. The additional set of instructions is invoked on the beginning of every remote method. It is easy to check whether we are inside the recovering phase by looking for a local context associated with a remote thread identity. The context object may be removed when we invoke the nested remote call or after full reconstruction of the distributed thread's state. This action guarantees that the next call of the same method will not be misinterpreted as part of the state reestablishing phase.

At the end of this section we address the issue of synchronization points. As stated in chapter 3, only two global barriers are needed when checkpoints in remote calls are not allowed. We claim that enabling the feature of state capturing in the remote calls needs two additional synchronization points.

One of them has to be added just before saving local checkpoints into a stable storage. The state of a distributed thread is distributed conceptually and physically. It consists of the states of all local threads, that are part of one distributed entity. Before saving the local data structures of a particular node, we have to make sure that there are no running local threads, that are part of distributed entity. Barrier synchronization just before the global checkpoint guarantees this behavior.

The second coordination point is required just after storing local checkpoints. This time the reason is not as obvious as before. In the description of the recovery phase we mentioned, that object containing the context of a local thread, which is part of a distributed thread is removed when we invoke nested remote method call during the stack rebuilding phase. As a consequence, some data, that should be part of a local checkpoint, may be lost, because one thread is faster than the other.

Claims that to capture the state of a distributed thread we just need to save the local states of the corresponding local threads are not so far from the truth. The next section describes briefly the modifications in byte code that are needed in order to support the new checkpointing features.

5.6. Byte Code Transformers

The byte code transformer had to be extended in order to deal with our new thread serialization scheme. The modifications are conducted in two phases. Each of them is served by a separate sub-transformer.

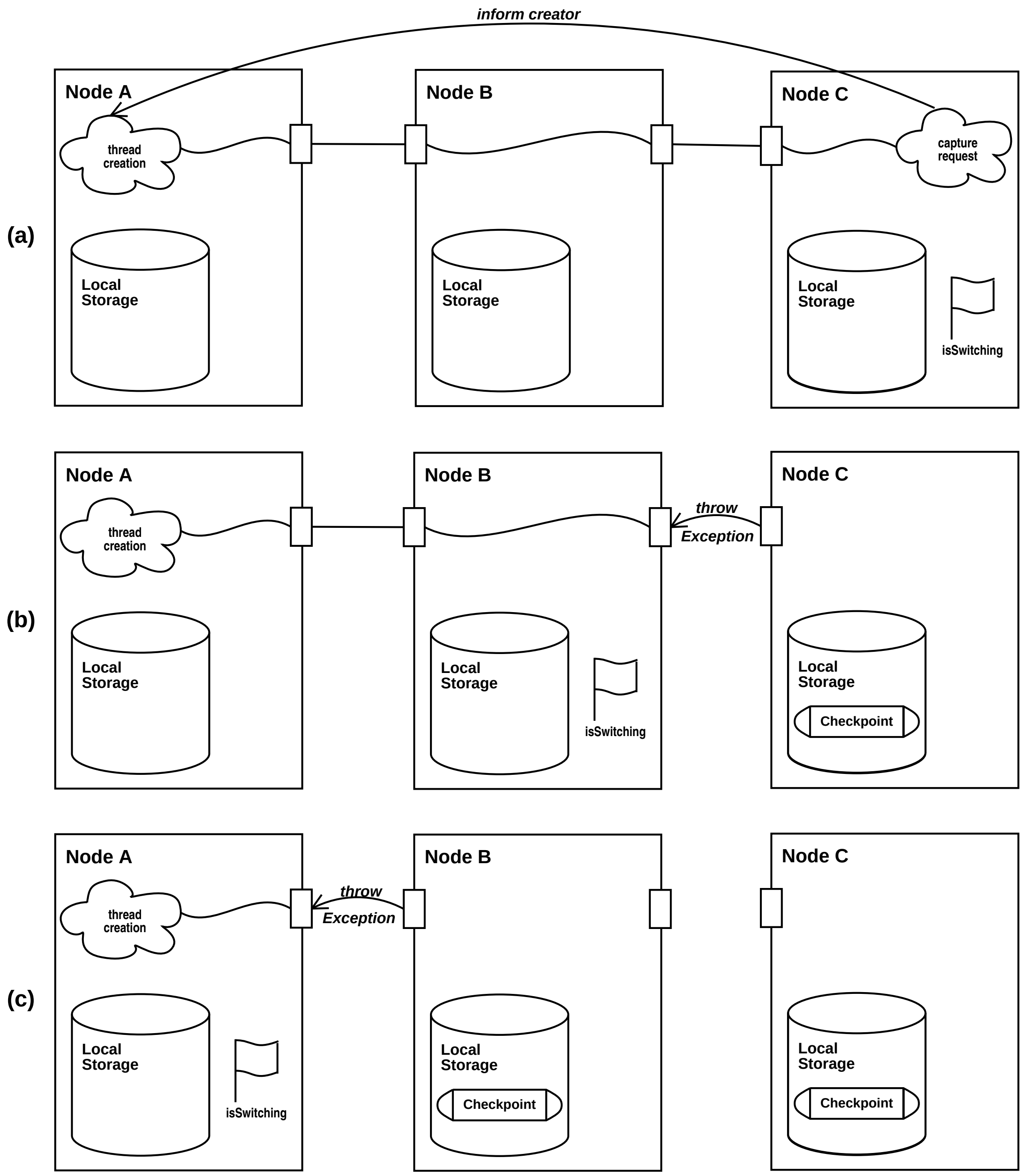


Figure 5.4: State Capturing in the Remote Method Call. Checkpoint request in the remote method call (a), local state captured; throwing exception sets the `isSwitching` flag on the remote node (b), the "capture state–throw exception" scheme is continued until control flow reaches the originator node (c).

5.6.1. Remote Method Transformer

In order to extend the functionality of the checkpointing framework, the semantics of remote method calls had to be slightly modified. We need to be able to identify a sequence of, not necessarily local, method calls as one (distributed) control flow. On account of this requirement the distributed identity was introduced. The task of the transformer, we are going to describe, is to modify remote methods in such a way, that the distributed identity can be sent along with the parameters of the remote method call.

The solution we propose is simple — the signature of a remote method is extended by adding the additional parameter — distributed thread identity. However, we promised that the overhead will be connected only to remote calls and will not influence the local control flow.

Remote method transformer creates a wrapper for every remote method. More precisely, the wrapper is a new method with the signature of the transformed one extended by the distributed identity. The body of the new method looks like in the example below.

```
public m_return_type wrapper_for_method_m(m_arguments, DistributedIdentity di)
{
    registerRemoteThread(di);
    m_return_type result;
    try {
        result = m(m_arguments);
    } catch (Exception e) {
        throw e;
    } finally {
        unregisterRemoteThread();
    }
    return result;
}
```

Remarks:

- `registerRemoteThread()` method, when invoked during normal execution, is responsible only for registration of a distributed identity in internal data structures of our system. Invocation during the recovery phase is much more complicated. Then the task of this method is to make the thread responsible for handling the remote call look like every other local (recovering) thread. We may call this step lazy binding;

- `unregisterRemoteThread()` is symmetric to `registerRemoteThread()`. During normal execution it just removes the distributed identity from the internal data structures. State capturing causes storing the context object and throwing a special type of exception, that indicates state capturing in a remote call. Note, that this method does not need any arguments. When invoked, the local thread is already known to the middleware;

- the wrapper is used only if the method was called using the RMI protocol. The original method is still available for local calls;

- the `finally` block of `try-catch` statement is evaluated regardless whether a checkpoint was thrown or not.

Additionally, for every class that is transformed, the remote interface is generated. This interface contains signatures of all wrapper methods that were added to the discussed class. The reasons are strictly technical. Wrapper methods are exposed through Java RMI. According to the RMI specification every remote method should be mentioned in an interface that extends `java.rmi.Remote`.

5.6.2. Stub Transformer

Java RMI uses a standard mechanism (employed in RPC systems) for communicating with remote objects: *stubs* and *skeletons*. A stub for a remote object acts as a client's local representative or proxy for the remote object. The caller invokes a method on the local stub, which is responsible for carrying out the method call on the remote object. In RMI, a stub for a remote object implements the same set of remote interfaces that a remote object implements. In the Java 2 SDK, Standard Edition, v1.2 an additional stub protocol was introduced that eliminates the need for skeletons in Java 2 platform-only environments. Instead, generic code is used to carry out the duties performed by skeletons in JDK1.1. Stubs and skeletons are generated by the `rmic` compiler.

The stub is an ideal place for our extensions, if they are to be invisible for the application programmer. The task of a stub transformer is to modify remote method calls in such a way, that passing the remote thread identity as one of the parameters is possible.

Beside having some additional features, our stub transformer is able to:

- generate the wrapper class for method arguments that are of primitive type. Every Java basic type has corresponding serializable class that may be marshaled and sent through the network;
- present to a user typical low level communication exceptions in a more convenient way.

Our compiler translates every remote method call to the corresponding wrapper method call. Exceptions thrown to indicate state capturing trigger actions that simulate local checkpoint request.

5.7. Overhead

The solution we presented has zero overhead when no checkpoints are initiated inside remote method calls. Some burden is connected with remote calls that can lead to the state capturing. This overhead is however rather theoretical than practical. Costs of extending method signature by the distributed identity parameter, one hashtable lookup on the client side and one hashtable insert on the server side introduce negligible overhead. Since checkpoints are stored along the path of the remote calls, the slowdown connected with initiating checkpoint on the remote node is also hardly visible.

5.8. Conclusions

The goal of this chapter was to present extensions for the fault tolerance mechanisms that eliminate some restrictions according to the placement of the state capturing initiation statement. The class of applications, that may be easily integrated with our system, was extended by allowing the programmer to invoke the checkpoint request also inside the remote method calls.

To model the actual situation we have to go one abstraction level up and think in terms of distributed rather than local threads. Notions of distributed identity and distributed context allow for treating a distributed thread as a set of local threads for which the system was originally designed.

Chapter 6

Related Work

This section presents a summary of relevant related work and places our project in this context. Although work on fault tolerance has a long history, we have not found any research projects in this area that use Java as a programming language for parallel computing. Since our approach is tightly related to the Java programming language, we had to come up with our own solutions for many problems. However, we discovered that many projects in the area of mobile agents address similar problems as our system.

6.1. Java Thread Migration

A major difficulty connected with transparent agent migration is preserving the execution state across migration. When developing our system we encountered the same problem — we had to preserve the state of the application in the checkpoint. Methods that address this problem can be divided into three categories.

The first category includes methods that modify the Java Virtual Machine. Such a modification exposes all necessary information, like the whole Java stack and program counter, and allows to preserve and restore them. Projects based on this idea include Nomads [13] and Sirac [2]. A major disadvantage of this methods is lack of portability. It is a problem, since our system was designed to work in a grid environment where it is not possible to assure that each machine in each cluster has the same version of the Java Virtual Machine.

The second category includes methods that make use of the Java Platform Debugger Architecture API. When the JVM is executed with special parameters, this API gives the programmer access to many internal JVM structures (including the Java stack). This method is very promising, but unfortunately, right now, only few Java Virtual Machines allow to use this API together with enabled Just In Time (JIT) compiler (probably only Sun 1.4 JVM supports it). Turning the JIT off introduces very big performance overhead, which is unacceptable for parallel computing. The CIA [7] mobile agents platform uses this technique.

The third category includes all kinds of pre- and post-compilers that instrument the code. They make the thread migration portable across standard JVM platforms although they introduce additional overhead. Our system uses the code from the Brakes [4] project as it was described in detail in section 2.2. Another transformation algorithm we considered to import into our system was used in the JavaGoX [10] project and is described in the next section. Projects JavaGo [11] and Wasp [5] use source code level approach.

6.1.1. JavaGoX code transformation algorithm

The JavaGoX [10] project uses basically the same execution state reestablishing algorithm as the one used in our system. The difference lies in the execution state saving algorithm. When the execution state capturing is requested, a *special exception is thrown*. This exception indicates that a program is in a state capturing mode. Each method invocation is surrounded by a try-catch clause, which catches the special state capturing exception and propagates it to the caller of the method after saving the current stack frame (see below for example pseudo code). This process is repeated until the exception reaches the bottom of the stack. Unfortunately, the operand stack is cleaned each time an exception is thrown and inside an exception handler it is not anymore available. To deal with this issue, the operand stack is copied to temporary local variables before every method invocation, but it makes this algorithm slower than the one used in our system.

A pseudo code of Fibonacci function (before the JavaGo transformation)[1]:

```
public class Fib {
    public static void fib ( int v1 ) {
        if (v1 <= 1)
            return 1;
        else {
            int v2 = fib(v1 - 2);
            return v2 + fib(v1 - 1);
        }
    }
}
```

A pseudo code transformed for state capturing:

```
public class Fib {
    public static void fib ( int v1 ) {
        if (v1 <= 1)
            return 1;
        else {
            int v2;
            try {
                v2 = fib(v1 - 2);
            } catch (Notify e) {
                ST_Fib_fib s = new ST_Fib_fib();
                s.EntryPoint = 1;
                s.v1 = v1;
                e.append(s);
                throw e;
            }
            try {
                return v2 + fib(v1 - 1);
            } catch (Notify e) {
                ST_Fib_fib s = new ST_Fib_fib();
                s.EntryPoint = 2;
                s.v1 = v1;
```

[1]The example comes from [10]

```
                s.v2 = v2;
                e.append(s);
                throw e;
            }
        }
    }
}
```

6.1.2. Source Code Modification v.s. Byte Code Transformation

Although developed transformation algorithms work both on a source code and a byte code level, we have chosen byte code transformation for our system. A source code transformation is rather limited compared with a byte code transformation. First of all, with a source code transformation, it is not possible in Java to extract the complete execution state of a running thread. It is, for example, not possible to inspect the values that are on the operand stack of the current executing method. One way out is to modify the source code in such a way that before each method invocation the operand stack is copied to temporary local variables. Consider the following piece of code:

$$x = foo() + bar();$$

To save the result of foo, the above expression is split up in advance as follows:

$$\begin{gathered} tmp = foo(); \\ x = tmp + bar(); \end{gathered}$$

Secondly, a byte code transformation is more efficient in terms of time and space overhead, due to the higher precise control offered at the bytecode level. Low-level bytecode instructions make it easier to manipulate the control flow in a program. For example, to prevent re-execution of already executed method code during reestablishment we skip the already executed code with a simple `goto` instruction. This instruction is however not available at the source code level. Some transformations introduce instead a huge amount of `if` statements to organize the skipping of already executed code.

Thirdly, a bytecode transformation gives more flexibility. For example, at the bytecode level, it is possible to skip the execution of default super-call within the constructor, while this is not allowed at the source code level.

Finally, the source code for third party libraries is not always available and in such cases bytecode transformation is the only option.

6.2. Distributed State Serialization

There were not many attempts for serialization of distributed Java applications. This section discusses the Distributed Brakes project [16].

The goal of the project is the development a mechanism for serializing the execution state of a distributed Java application that is programmed by means of an Object Request Broker like Java RMI. To validate their research, the authors built a prototype for repartitioning distributed Java applications at runtime. This mechanism enables applying of existing partitioning methods at any point in an on-going distributed computation. Runtime repartitioning aims to improve the global load balance or network communication overhead of a running application by repartitioning the object configuration of the application dynamically over the available physical nodes at run-time. Existing work offers this support in the form of middleware platform with a dedicated execution model and object migration support that aligns well with run-time repartitioning. A disadvantage of this approach is that existing ordinary RMI-based applications, which have obviously not been developed with support for repartitioning, must partially be rewritten such that they become compatible with the programming model of the new middleware platform. The approach of the Distributed Brakes project is to develop a byte code transformer that transparently adds new functionality to an existing Java application such that this application becomes automatically repartitionable by the external monitoring and management architecture. The repartitioning component, although it has many interesting aspects, lies outside the scope of this paper.

Distributed Brakes uses a byte code transformer that extends Java programs with the notion of distributed threads, more specifically distributed thread identity. In the rest of this section we refer to this transformer as the *DTI transformer*. Distributed thread identity is, in this context, a serializable class that implements an immutable, globally unique identifier. In comparison with the notion of distributed thread identity introduced in chapter 5, the distributed identity implemented in the Distributed Brakes system does not encapsulate any additional information.

To achieve propagation of distributed thread identities, the DTI transformer extends the signature of each method with an additional argument — distributed identity. The signature of every method invoked in the body of the methods must be extended with the same identity argument too. For example, a method `f()` of a class `C` is rewritten as[2]:

```
// original method code
void f(int i, Bar bar) {
     ...
     bar.b(i);
     ...
}

// transformed method code
void f(int i, Bar bar, D_Thread_ID id) {
     ...
     bar.b(i, id);
     ...
}
```

where `D_Thread_ID` is the class of distributed thread identity. When `f()` is called the `D_Thread_ID` is passed as an actual parameter to `f()`. Inside the body of `f()`, `b()` is invoked on `bar`. The body of `f()` passes on its turn the `D_Thread_ID` it received as an extra argument to `b()`. This way the distributed thread identity is automatically propagated with the control flow from method to method. Dynamic integration with a distributed object-based middleware such as Java RMI is simply achieved if the stub classes are generated after the DTI transformation has been performed.

Note, that in this solution not only remote but also local methods have to be modified which of course introduces additional overhead even if no remote methods are called.

To deal with distributed threads, the local implementation of the context manager was adopted. Distributed Brakes manages one Context object per distributed thread (see figure

[2]The example comes from [16]

6.1). The appropriate Context object is looked up with the current thread identity as a hashing key.

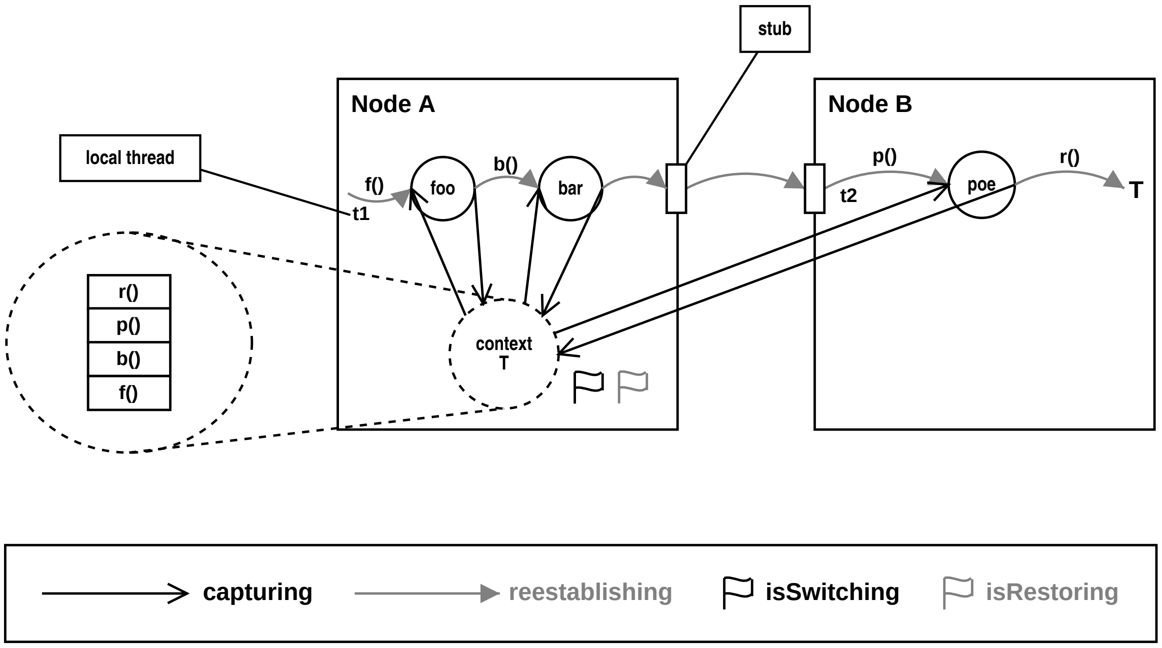


Figure 6.1: Context per Distributed Thread (adopted from [16])

However, in order to allow the context manager to manage Context object on a per distributed thread basis, the static bottleneck interface for inspecting distributed thread identity had to be introduced. Figure 6.1 motivates clearly that the implementation of the context manager must become distributed now. Figure 6.2 sketches the architecture of such a distributed implementation.

Capturing and restoring code blocks still communicate with the bottleneck interface of the local context manager, but the captured execution state is now managed per distributed thread by a central *distributed thread manager*. The distributed thread manager manages global flags that represent the execution state of the distributed application as a whole. These global flags are kept synchronized with the flags of the local context managers.

The solution used here and in particular the idea of having one common Context for all local threads that are part of one distributed identity is not flexible. The problem, that discards Distributed Brakes as a framework for taking distributed checkpoints, is connected with sharing Java objects by different threads. If we want to capture a state of more than one thread in one checkpoint we have to guarantee consistency of data. In particular it means that the state of all threads running inside one Java Virtual Machine has to be serialized in “one serialization step” (the reasons were explained in chapter 5). In Distributed Brakes however, the execution state of a thread is written to a distributed context frame by frame. The Java stack frame is sent out to the distributed task manager just after being captured. This approach is adequate when the middleware is used for tasks that require capturing the state of one distributed thread at once, such as runtime repartitioning.

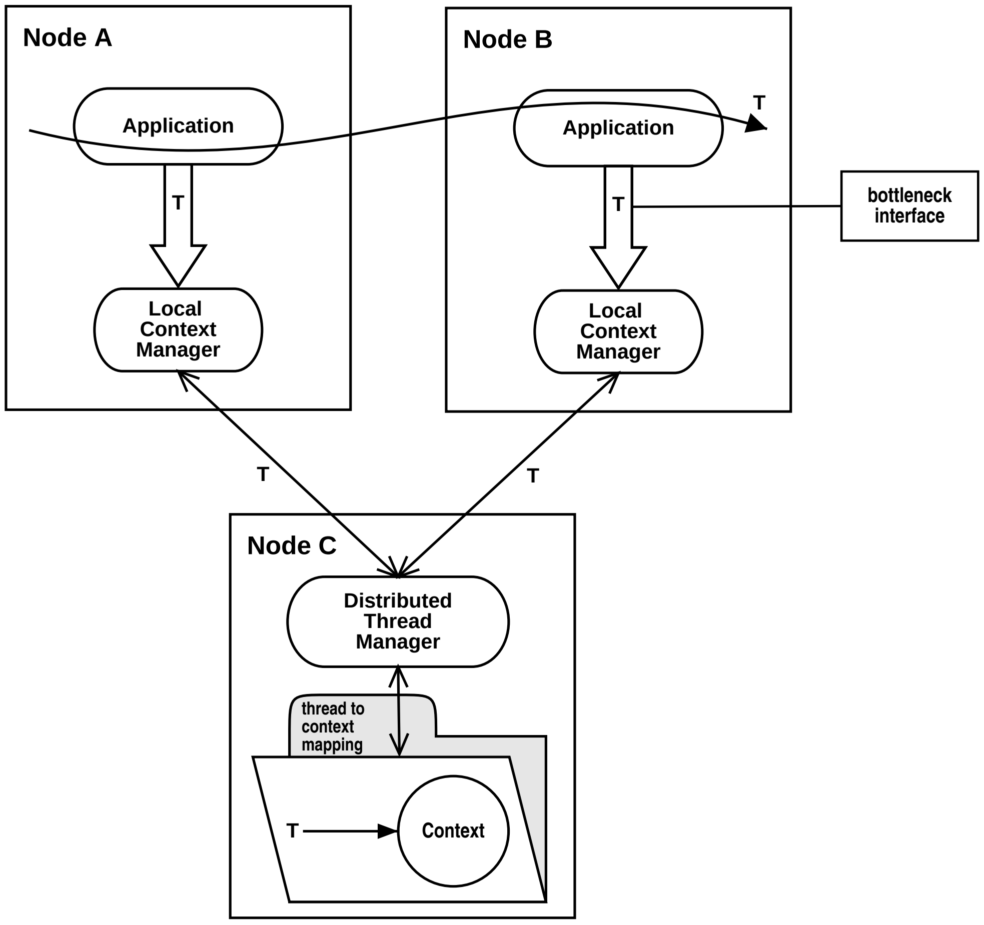


Figure 6.2: Distributed Architecture of the Context Manager (adopted from [16])

Distributed Brakes is an interesting extension that provides the original Brakes system with distributed thread's state capturing functionality. We borrow the notions of distributed thread and distributed identity from this system.

Chapter 7

Measurements

7.1. Introduction

In this chapter we evaluate the middleware providing fault tolerance for distributed Java applications. Since inserted byte code introduces not only time but also space overhead we measure as well the loss of performance as the byte code blowup. To get a representative picture, we did tests on different types of applications. We first describe our evaluation environment, and then present the performance figures of our Java fault tolerance mechanisms.

7.2. Evaluation Environment

The performance results presented here were obtained on a DAS-2 ([1]) cluster computer. The cluster we used contains 72 homogeneous nodes. Each node contains:

- two 1-GHz Pentium III processors, 16 KByte L1 cache, 256 KByte L2 cache,
- at least 1 GB RAM (2 GB for two "large" nodes),
- a 20 GByte local IDE disk,
- a Myrinet interface card,
- a Fast Ethernet interface (on-board).

The cluster is running on Red Hat Linux release 7.2 with kernel 2.4.18. Tools included in IBM Java 2 Standard Edition version 1.4.0 release were used for compiling and running evaluated applications. Each presented test was repeated at least three times and the average of results was considered.

7.3. Applications

We selected the applications based on the criteria that they are challenging to the checkpointing subsystem, i.e. they use a complicated control flow scheme and gather much temporary data. Making these programs fault tolerant manually requires non negligible effort.

We used four applications in this study: ASP, SOR, TSP and Shallow Water. The applications vary widely in their complexity. For example ASP and SOR are relatively simple while Shallow Water consists of several advanced mathematical transformations (Fast Fourier and Legendre).

Our applications differ greatly in the type and frequency of synchronization, the degree of sharing data, the communication scheme, the degree to which the data domain of a particular process changes over the length of a program execution, and the granularity of exchanged data. The diversity of our application suite ensures that the results of this study are representative of a large class of programs, rather than being specific to a single type.

7.3.1. Successive Over Relaxation

Our *Successive Over Relaxation* (SOR) uses a simple iterative relaxation algorithm. The input is a two-dimensional grid. During each iteration, every matrix element is updated to a function of the values of neighboring elements. In this case, the function is an average of the four neighboring elements. To avoid overwriting an element before neighbors use it for their computations, we use a "red-black" approach, wherein every other element is updated during the first half-iteration, and the rest of the elements are updated during the second half-iteration. The work is parallelized by assigning a contiguous chunk of rows to each process. Exchange of data between processes is therefore limited to those pages containing rows on the edge of the chunks.

The implementation we used for tests has several configurable properties. They include:

matrix size number of columns and rows of the matrix we are going to compute;

iterations number of iterations to calculate. One possibility is to detect termination automatically;

communication inter-node communication scheme. Possible options allow for either synchronous or asynchronous communication;

communication thread type this parameter is enabled if and only if we use the asynchronous communication scheme. There are two possibilities. One is to start a new thread every time we want to send data to our neighbor. Other allows us for reusing a single thread.

In the original SOR algorithm after every phase processes running on different nodes synchronize on a global barrier to guarantee that no data will be removed before they are used. In the variant we used, the problem of premature deletes was solved by locking data on per-node basis.

7.3.2. Shallow Water

The *Shallow Water* application (known as Parallel Spectral Transform Shallow Water Model — PSTSWM) is a message-passing Java program that solves the nonlinear equations on a rotating sphere using the spectral transform method. The shallow water equations in the form solved by the spectral transform method describe the time evolution of three *state* variables: velocity, divergence, and a perturbation from an average geopotential. The velocities are computed from these variables. Shallow Water advances the solution fields in a sequence of time steps. During each time step, the state variables of the problem are transformed between the physical domain, where the physical forces are calculated, and the spectral domain, where the terms of differential equation are evaluated.

Transforming from physical coordinates to spectral coordinates involves performing a real fast Fourier transform (FFT) for each line of constant latitude, followed by integration over latitude using Legendre transform (LT) to obtain the spectral coefficients. The basic outline of each time step is described below.

1. Evaluate non linear product and forcing terms.
2. Compute forward Fourier transform of non-linear terms.
3. Compute forward Legendre transforms.
4. Advance in time the spectral coefficients for the state variables.
5. Evaluate sums of spectral harmonics, simultaneously calculating the horizontal velocities from the updated state variables.
6. Compute inverse Fourier transform of state variables and velocities.

The parallel algorithms in PSTSWM are based on decompositions of the physical and spectral computational domains over a logical two-dimensional processor mesh of size $PX \times PY$. Initially, the longitude dimension of the physical domain is decomposed over the processor mesh row dimension and the latitude dimension is decomposed over the column dimension. Thus, FFTs in different processor rows are independent, and each row of PX processors collaborates in computing a "block" of FFTs. Similarly, the Legendre transforms in different processor columns are independent, and each column of PY processors collaborates in computing a "block" of Legendre transforms. The computation of the nonlinear terms at a given location on the physical grid is independent of that at other locations.

7.3.3. Traveling Salesman Problem

In the Traveling Salesman Problem (TSP), the goal is to find the shortest route for a salesman to visit (exactly once) each of the n cities in his territory. The algorithm we used for our measurements uses a *tree* to structure the space of possible solutions. A node of the tree represents a partial tour. Each node has a branch for every city that is not on this partial tour. The parallel algorithm uses a *manager process* which traverses the top part of the tree, up to certain depth D. For each node on depth D, the manager generates a job to be executed by a *worker* processes. A job consists of the evaluation of the subtree below of the given node. Effectively, the searched tree is distributed among several processes. The manager process searches the top D levels: one or more worker processes traverse the nodes at the lower N – D levels. In order to reduce the number of searched branches workers exchange between each other information about the shortest path already found. The workers can skip parts of the tree below nodes which represent partial tours already longer than the global minimum. However, exchanging information about the global minimum introduces additional communication overhead.

Since in the algorithm we described, there is no obvious synchronization point where the global checkpoint could be done, we used the timer to request the checkpoint (and thus global synchronization) once every chosen time period.

7.3.4. All-pairs Shortest Paths Problem

In the All-pairs Shortest Path (ASP) problem, the goal is to find the length of the shortest path between any two nodes in a given graph. The standard solution to this problem uses an iterative algorithm. It assumes that nodes are numbered sequentially from 1 to N (total number of nodes). During iteration *k* it finds the shortest path from every node *i* in the graph to every node *j* that only visits intermediate nodes in the set [1..k]. During iteration *k*, the algorithm checks if the current best path from *i* to *k* concatenated with the current best path from *k* to *j* is shorter than the best path from *i* to *j* found so far. After the last iteration, the resulting path between every node *i* and every *j* is the shortest one because it may visit any other node from the set [1..N].

The standard algorithm uses a sequence of matrices for storing lengths of all these paths. Each iteration is represented by a new matrix. Matrix element (i, j) corresponds to the currently shortest path between nodes *i* and *j*. In a parallel version of this algorithm, each processor takes care of some of the rows of the matrices. When a process finishes computing the row, it sends it to all the others. A process should not start working on iteration *k* until the value of row *k* from *k-1* iteration is available.

In order to make this algorithm fault tolerant, we instrumented the code with additional instructions. Whenever a process starts a new iteration, it checks if a checkpoint was requested by any other processor and if so, it synchronizes with the others in order to make a global checkpoint. A global checkpoint is requested by a process whenever it reaches a certain iteration.

7.4. Results

7.4.1. Performance overhead of the normal execution

In this section we present the time overhead introduced by the byte code transformations applied to the distributed SOR, ASP, TSP and Shallow Water algorithm implementations. More precisely, we are describing the overhead introduced by instrumenting the application's byte code with additional `if` statements after method calls that may result in state capturing and the overhead connected with replacing standard Java RMI with our fault tolerant counterpart.

Parallel programs usually use one of two kinds of communication models: synchronous or asynchronous. Synchronous communication occurs when a process sending data cannot continue until the message has been successfully delivered to the receiver. The standard Java RMI supports only synchronous message passing, which usually causes big performance overhead. In order to use asynchronous communication, a programmer has to explicitly create a separate Java thread responsible for sending messages. This can be done in one of two ways. A new thread can be created each time a message is sent or one thread can be reused (the thread has a queue for send requests and serves them one by one). All three types of communication types have been implemented in the ASP and SOR algorithms and therefore we have tested them with our system.

Figures 7.1 – 7.7 present the performance overhead of our transformations for each tested algorithm. For every case we have two plots. One of them compares the execution times of the original and transformed application when no checkpoints were taken. Note, that normally applying the postprocessor to the program code that does not contain `make checkpoint` statement will not modify the byte code at all. To constrain the byte code modifications we placed the `make checkpoint` statement inside a conditional block that was never executed. The second subplot presents the same overhead in a normalized form. The plots show the relation between the number of nodes used for computation and the execution time.

Figures 7.1, 7.2 and 7.3 present three sets of experiments on the SOR algorithm for matrix 1000x10000 and 200 iterations. The first one uses synchronous communication. As one could expect, the overhead increases with the number of nodes used in computation. This fact may be explained in many different ways. The SOR algorithm works in phases. During the n-th phase every process first performs computations on the local data, sends results and waits for results from other workers. Each of the `if` statements added after method calls that may lead to a state capturing is executed once per phase. The highest noticed execution time overhead was 9%.

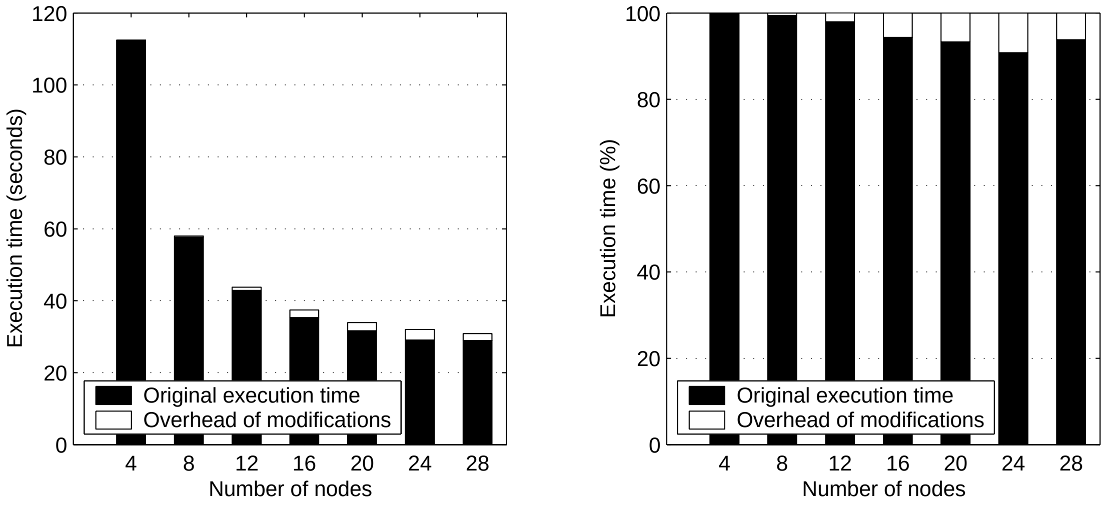


Figure 7.1: The performance overhead of byte code transformations of a *single thread per node* variant of distributed SOR application

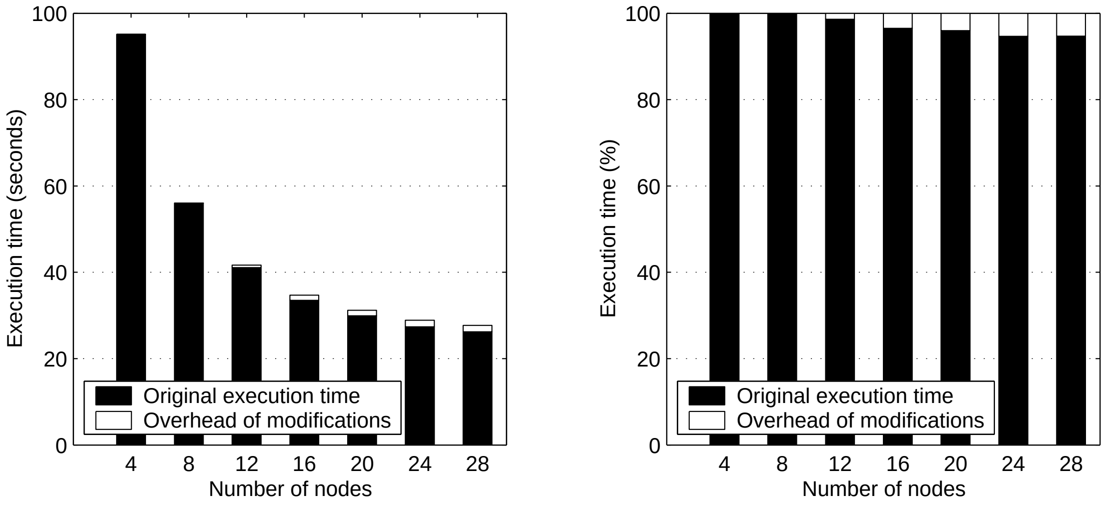
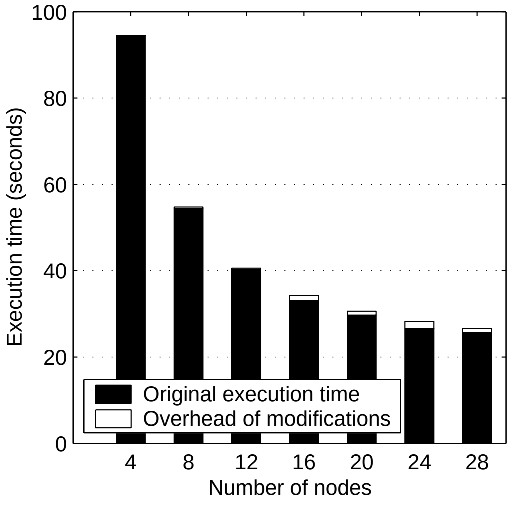


Figure 7.2: The performance overhead of byte code transformations of a *thread pool* variant of distributed SOR application

The second set of experiments was based on asynchronous communication scheme with a thread pool (a pool of threads which are reused for sending messages). The results are presented in figure 7.2.

For the given SOR implementation, there are three application threads per node. One is responsible only for computation. Communication with neighbors is served by separate threads on one dedicated thread per neighbor basis. These threads are reused in different phases of the computation. The sender threads are created once — on the application startup and stay active until the end of the computation. Because of their longevity, the communication threads have to be checkpointed with the application. When a thread is registered in our framework several data structures as the `Context` and the `Computation` objects have to be created. Some overhead is introduced also by the statements added by the postcompiler. Now not only the main thread (responsible for computation), but also the sender threads include `make checkpoint` request. To support serialization of those threads corresponding method calls have to be modified. The results, however, show that these actions does not influence the performance and the overhead is even lower than in the previously considered case. The maximal noticed execution time overhead was 5%.

The most complicated variant of the distributed SOR application we used for testing, starts a new thread for every data send request. The results presented in figure 7.3 are again very similar to previous tests and reach 6% for 24 nodes.



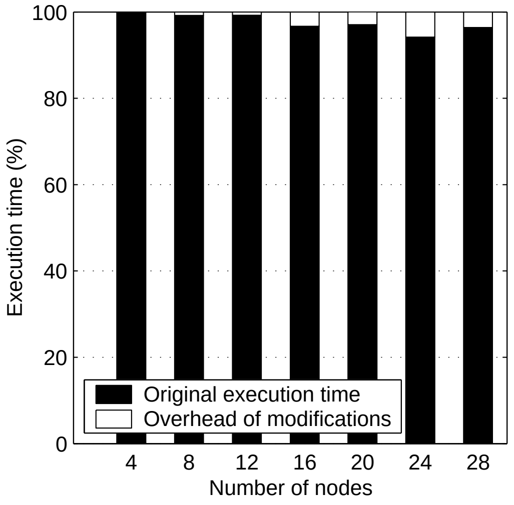


Figure 7.3: The performance overhead of byte code transformations of a *new thread per message* variant of distributed SOR application

The results of the ASP algorithm (figures 7.4 – 7.6) show that the standard Java implementation is not appropriate for parallel programming. Rather than getting speedups with the growing number of processors, we got slow-downs. It can be explained by the fact that the ASP's implementation we have tested, uses binary trees for broadcasting messages to all processors. When a process wants to broadcast a message to all other processes, it sends it to two of them which in turn send it to other two processes which have not received this message yet, and so on. Finally, all processors receive a copy of the message. However, this introduces performance overhead when using the standard Java RMI implementation. Each time a process receives an object (which actually is the message), it deserializes it, serializes it again and propagates to the next processes. Since serialization and deserialization of Java objects is very expensive, it causes big performance overhead. When the group of processors to which a message has to be broadcast is growing, the performance overhead of every broadcast request increases causing observed speed-downs. This problem has already been addressed in the Ibis [9] project, where the Java serialization mechanisms have been noticeably improved.

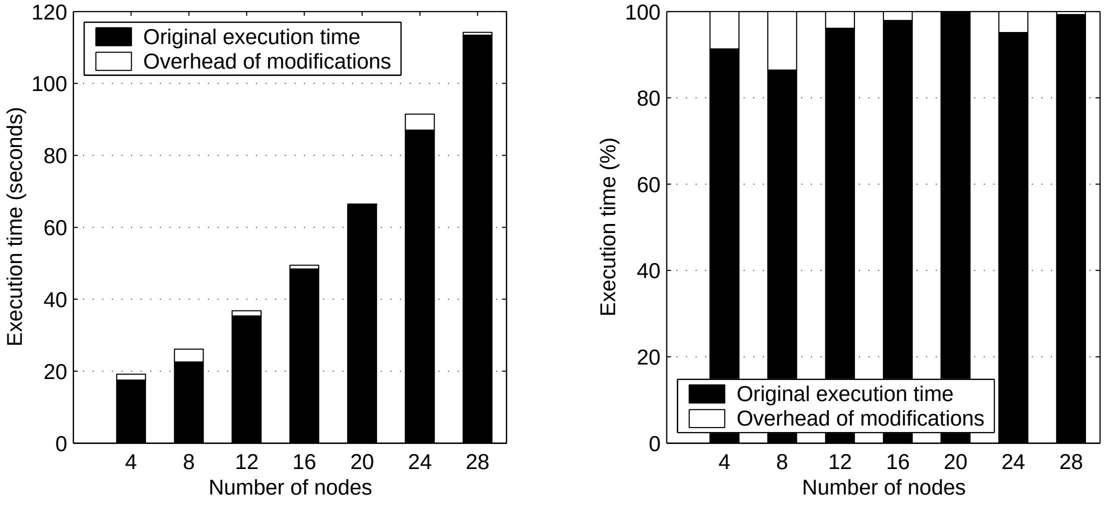


Figure 7.4: The performance overhead of byte code transformations of a synchronous variant of distributed ASP application

As for the SOR algorithm, the All-pairs Shortest Path (ASP) algorithm's implementation that we used, can be executed in three different modes. For each test we used matrix 1000x1000. Figure 7.4 shows the results for synchronous communication. The performance overhead in this case is very unstable — for 20 nodes the results did not show any overhead at all, but for 8 nodes we measured the overhead of 15%. The variety of results is probably due to a very frequent communication between nodes, which causes nondeterministic behavior of the application.

The second set of experiments (figure 7.5) was based on asynchronous communication with a thread pool. Here, the results are much more stable than for the synchronous mode and vary around 9%.

The third set of experiments (figure 7.6) on the ASP algorithms uses asynchronous communication with a separate thread for each message sent. In this case, the overhead started at the level of 10% for 4 nodes and was decreasing with the growing number of processors. Finally, it was not visible at all.

The TSP application is much simpler than ASP. It uses only synchronous communication and the overhead of our transformation algorithm is negligible — the highest noticed was less than 2% (see figure 7.7). The `if` statement inserted by the byte code transformation algorithm is executed only once each time a job is fetched by the worker process from the server process. Since processes do not communicate with each other as frequently as in the ASP or SOR algorithms, the overhead caused by our transformation algorithm is not so visible.

The Shallow Water application is much more complicated from a mathematical point of view than the previous algorithms. At the same time the distributed flow control of water molecules' positions computation is easier to analyze. The algorithm consists of several iterations which are further divided into steps. After each step all the processes running on different nodes perform global barrier synchronization. There are no separate

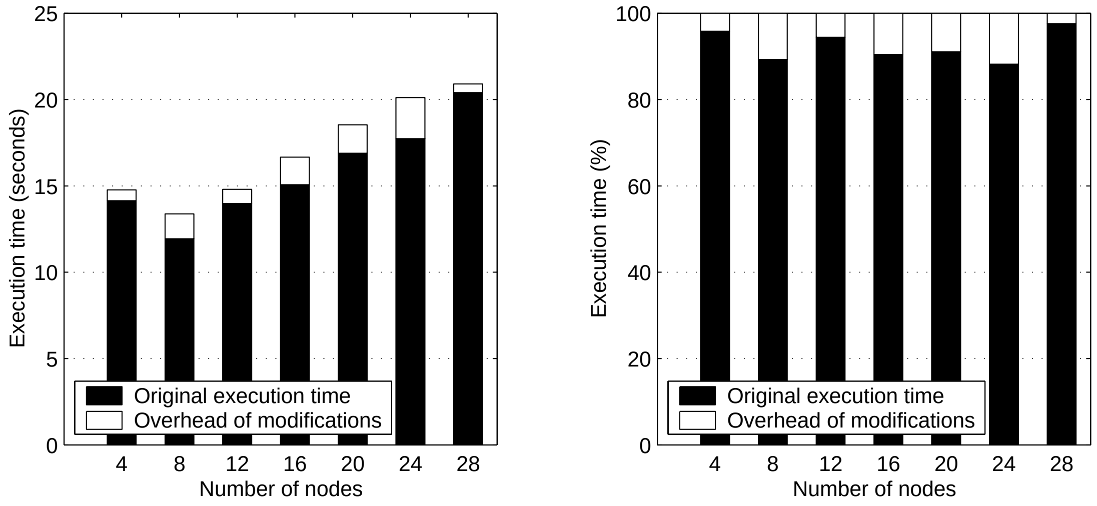


Figure 7.5: The performance overhead of byte code transformations of an asynchronous with thread pool variant of distributed ASP application

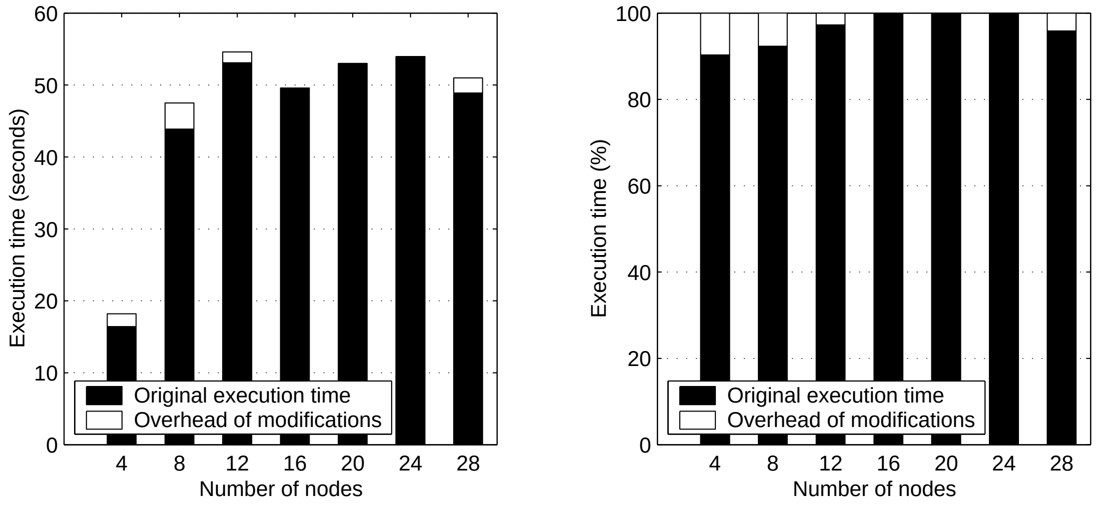


Figure 7.6: The performance overhead of byte code transformations of a *new thread per message* variant of distributed ASP application

sender threads that need to be synchronized before checkpoint, so the byte code compiler transforms the methods of one thread only. All these circumstances explain extremely low overhead below 1% for tests on 1728 molecules. The details can be found in figure 7.8.

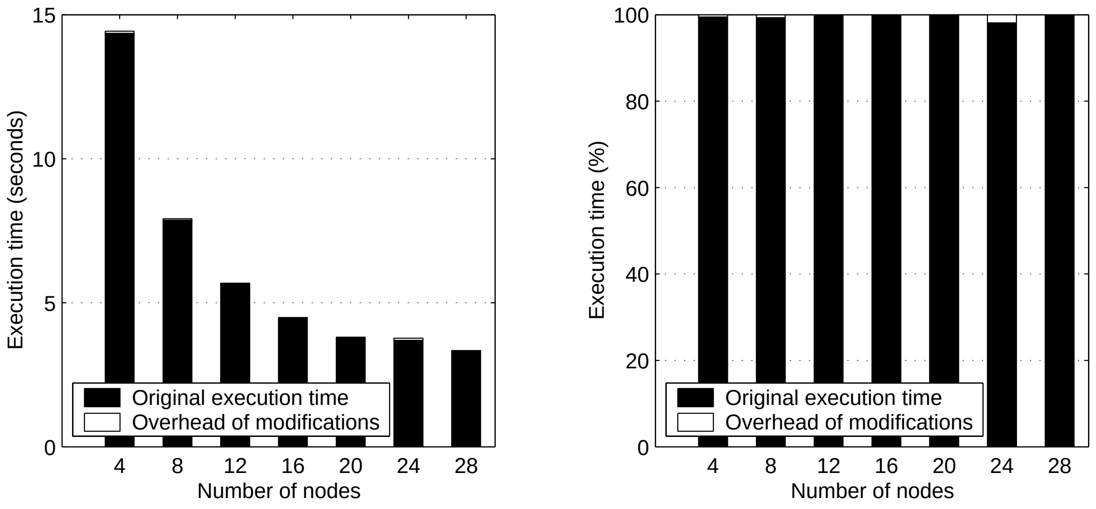


Figure 7.7: The performance overhead of byte code transformations of distributed TSP application

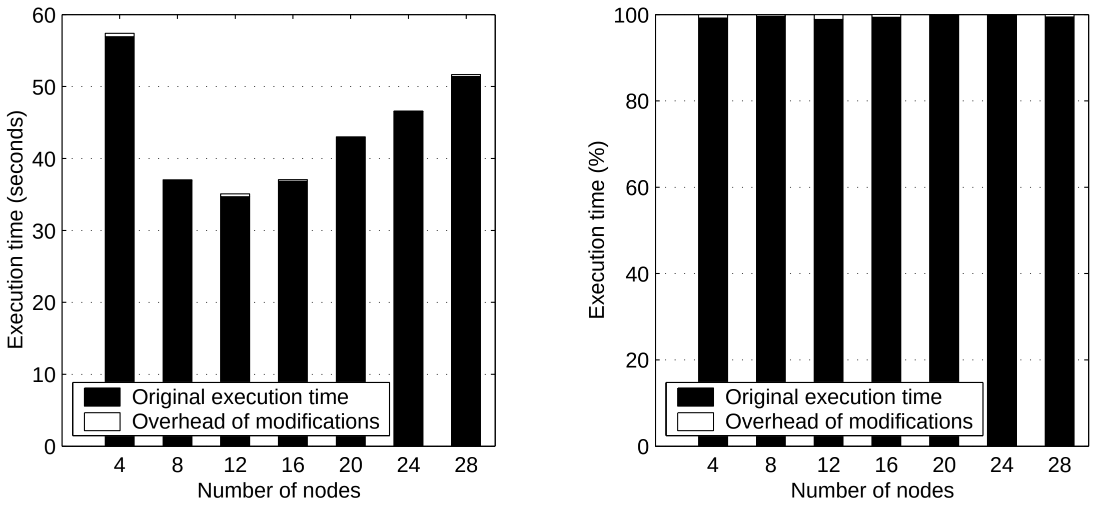


Figure 7.8: The performance overhead of byte code transformations of the Shallow Water distributed application

7.4.2. Performance overhead of taking distributed checkpoint

In this section we analyze the delays of checkpoints. More precisely, we measure the time interval between the last checkpoint request and the last thread resumption (after checkpoint). The plots present the delay as functions of the checkpoint size and the number of nodes involved in the global synchronization.

Figure 7.9 shows the dependency between the size of the saved data and the time needed for taking globally consistent checkpoint for our applications running on 20 nodes. Since the size of the applications data vary during the execution, we measured the size of the checkpoint.

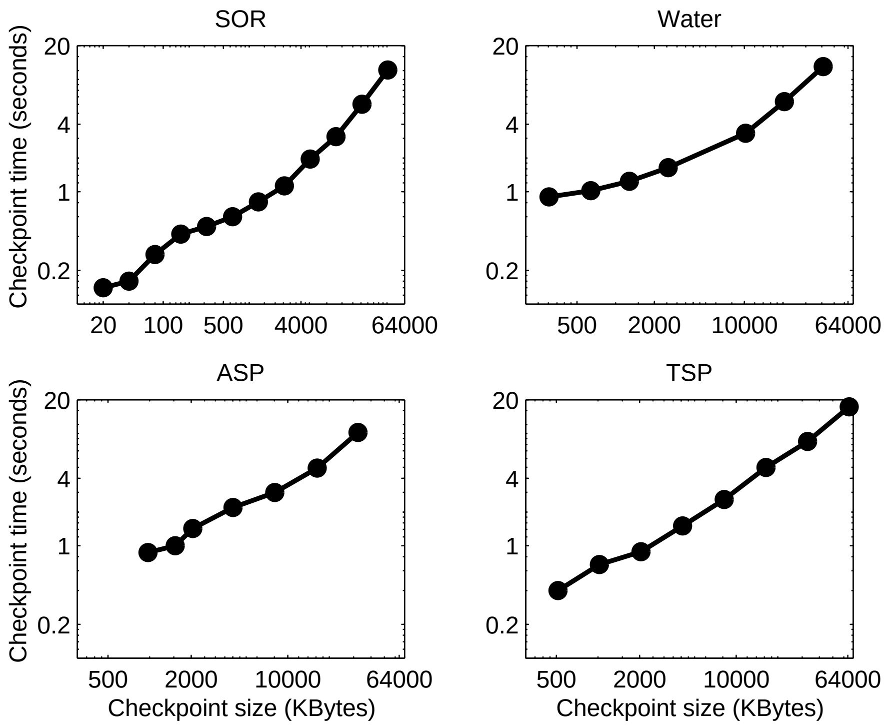


Figure 7.9: The time of the checkpointing phase as a function of data size

For all the tests described in this section we used the most complicated setup, namely asynchronous communication with a thread pool. TSP and Shallow Water are exceptions as they do not allow for modifying the communication method. Our system has a configurable option that allows to specify the *replication factor*, that tells how many times the checkpoint should be replicated. Higher number means higher reliability. In other words, replication factor is the number of nodes that may crash without making the system unable to recover. However, more replicas means also bigger overhead of the serialization phase. In our tests we used the lowest possible replication factor value — 1.

Note that the graph in figure 7.9 has logarithmic scales on both axes. The function we are plotting is monotonic so log-log plot keeps the relation of original data. As one could expect, the time of a checkpoint can be approximated by a linear function of data size. If we look closer we may notice that our plot is in fact composed of two linear functions. For smaller checkpoints (less than 3MB) the time needed for capturing the state of the distributed application grows slower than for bigger data. For smaller checkpoints the time depends mainly on the efficiency of state saving extensions. Checkpoints, that contain huge amounts of data move the overhead from the data serialization to data sending and storing phase. This may explain the differences in performance behavior.

Figure 7.10 shows the influence of increasing the number of nodes on the time needed for taking a global checkpoint.

The size of the checkpoint was approximately the same for all configurations — 500KB. SOR and ASP use pooled send threads for communication. All the applications indicate very similar characteristics in regard to the length of the checkpointing phase. The algorithm is quite insensitive to increasing the number of processors involved. Deeper analysis of SOR and ASP cases reveals groups of node numbers with similar characteristics. These

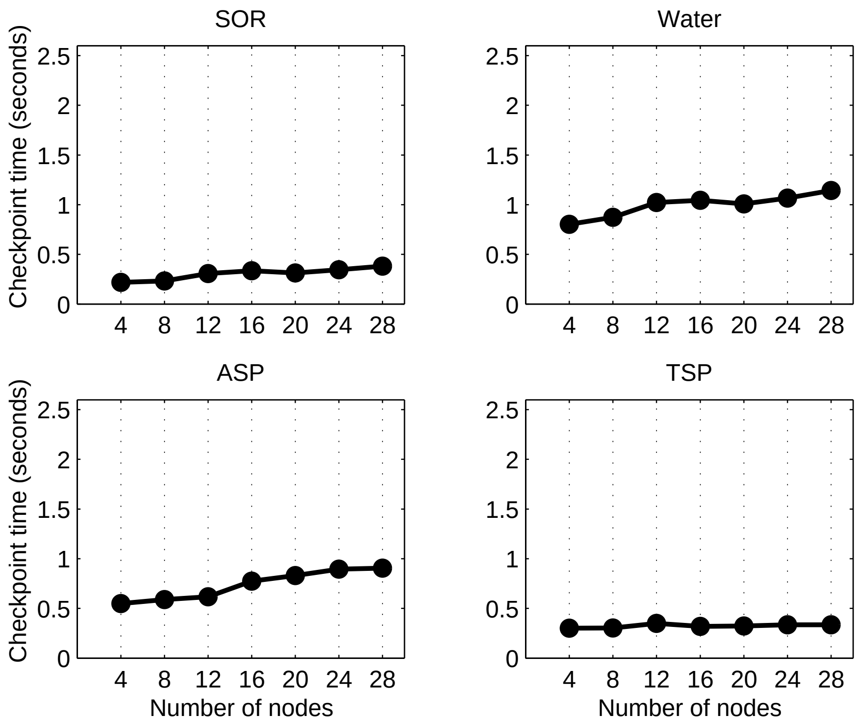


Figure 7.10: The time of the checkpointing phase as a function of the number of nodes

groups are delimited by numbers of nodes that are powers of 2. It may be explained by the characteristics of the synchronization algorithm. The neighborhood function used for purpose of presented tests defines a broadcasting structure in a form of the binary tree. Processes running on different nodes should synchronize in logarithmic time. Obtained results show that our algorithm can deal with bigger number of nodes without much loss on performance. Furthermore, the overhead of the checkpointing phase when a number of participating nodes is increased depends not only on the global synchronization. Threads running on different nodes may need different amounts of time for capturing their state. The oscillations of the state serialization times may accumulate and introduce visible delays. These facts may explain the results obtained for the Water application where the overhead for 20 nodes is lower than for 16. The best results were obtained for the TSP algorithm. In this case there is almost no overhead connected with increasing the number of nodes.

7.4.3. Performance overhead of the recovery process

When a distributed application crashes, our system allows to recover it. However, before the recovery process can be initiated, the failure of the process has to be noticed by another process. It happens when the global checkpoint is requested (all nodes try to synchronize) or earlier when the other process fails to send a message to the broken process (our fault tolerant stubs detect an error). When one of these situations happens, the recovery process is initiated. The entire process of recovery consists of several steps. First of all, the coordinator process scans all processes in order to discover which of them are alive. All alive processes are informed about the result of this scan. Processes which keep checkpoints for broken processes are requested to recover them. In order to do that, they contact an activator processes (responsible for restarting the processes), send them the latest checkpoint and request restarting that checkpoint. Then, all alive processes request their activators to restart them using the latest checkpoint (which they keep in the memory). When an activator process receives the restart request together with a serialized checkpoint, it saves the checkpoint in a file on the hard disk, starts the application in a new Java Virtual Machine and passes the file name where the checkpoint is saved to this application. The application reads the checkpoint from that file and reestablishes its state.

In our tests we were killing one of the processes and we measured the time of the recovery of the entire distributed application. Since the time necessary to discover the failure of the process is entirely application dependent (depends on the frequency of communication between processes), we did not include it in our measurements. We have measured the time elapsed since the first process noticed the failure until all processes recovered and continued the execution.

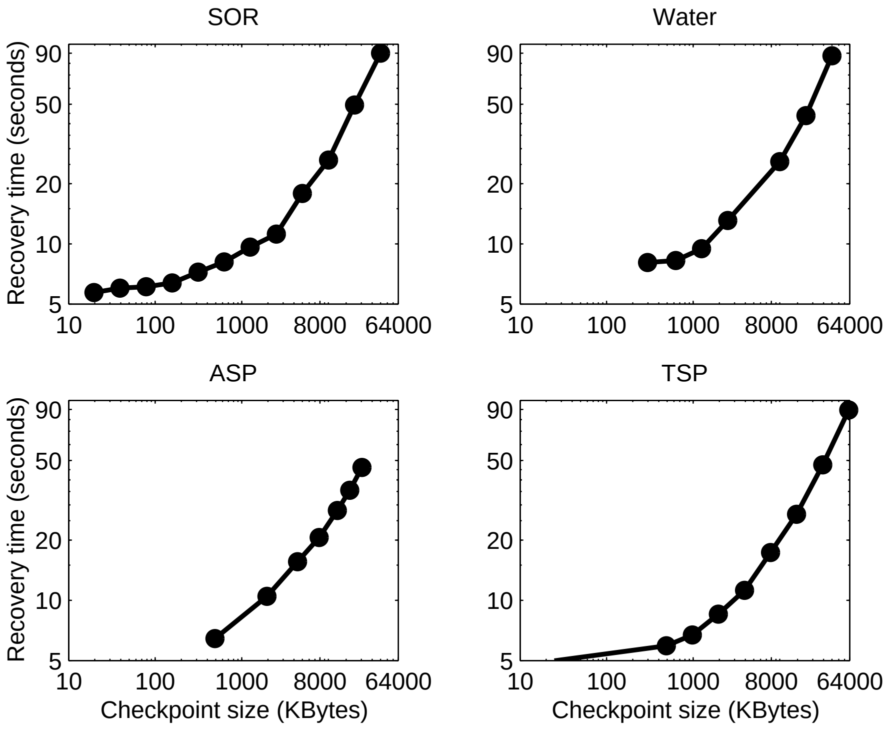


Figure 7.11: The recovery time as a function of checkpoint size

Figure 7.11 presents the relationships between the average size of the checkpoint and the recovery time for each tested application. Notice that all of them are presented on the log-log scale due to the exponentially growing checkpoint sizes chosen by us. We measured the size of saved data (checkpoint) that had to be restored. We tested checkpoints up to 64 megabytes, which in the case of the TSP and the Shallow Water applications were artificially increased by adding big dummy arrays of data to the application. All tests show that for checkpoints smaller than one megabyte the recovery time increases much slower than for larger checkpoints, although for both intervals the increase is linear. We expect that for checkpoints smaller than one megabyte the recovery time is influenced mostly by global synchronizations of processors and starting a new Java virtual machine. The high reliability of the global coordination algorithm used here decreased its efficiency. For bigger checkpoints the overhead is caused mostly by hard disk access when an activator process is recovering the application. Each node of the DAS-2 cluster consists of two processors and each of them is executing a separate JVM with a different computation. The disk access is a bottleneck when both computations at the same time transfer big amounts of data to and from the same hard disk. When the checkpoint size increases, the disk access time also increases extending the whole recovery process.

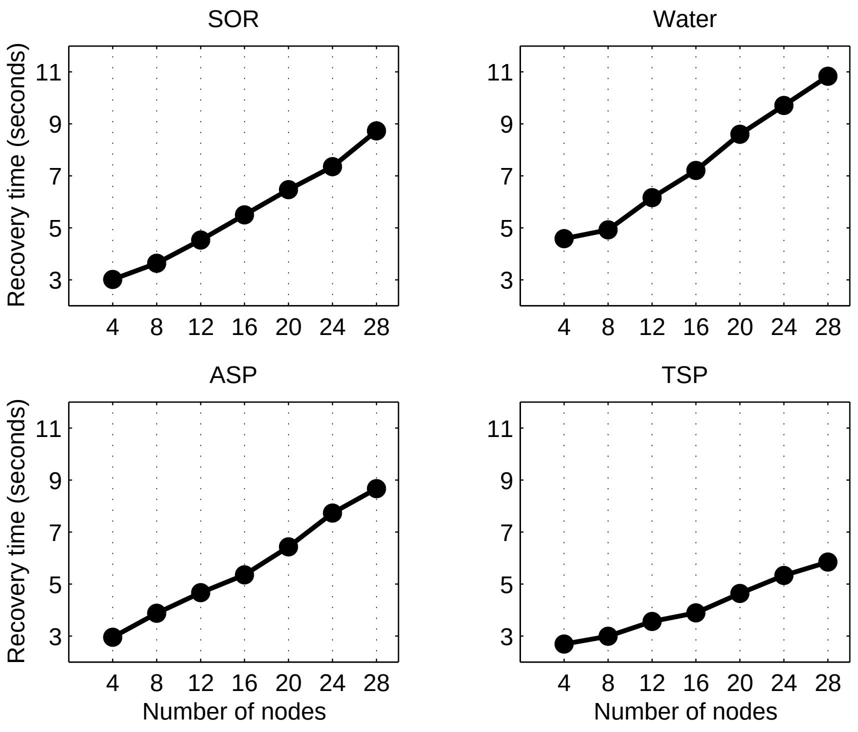


Figure 7.12: The recovery time as a function of the number of nodes

Similarly, figures 7.12 present the relationship between the number of nodes and the recovery time for each tested application. They all show that the recovery time depends in a linear way on the number of processors, however the recovery time increases very slowly. The bigger number of processors cause longer global synchronization times and bigger probability that one of the processors will slow down the whole recovery process (the rest will be waiting for that process).

Note that we did not present the figures of the real-time overhead of the checkpointing and recovery phases, because it depends on the frequency of making a checkpoint. Moreover, our system was designed for large-scale applications running for a long time and the total execution time of example programs is too short.

7.5. Methods call graph improvements

In this section we compare the original transformation algorithm used in the Brakes project with the version improved by us. We optimized the original Brakes algorithm by rewriting only invocation of methods which can lead to the checkpoint request (see chapter 2.3 for details). Figure 7.13 shows the gain in the performance when using methods call graph for described parallel programs running on 8 nodes. The results show that the performance of the TSP application increased over 30% and the performance of SOR over 10%. Since the TSP application uses a recursion, it has many method calls which are rewritten by the Brakes algorithm, though they never lead to a state capturing. Our optimized algorithm did not improve the performance of ASP and Shallow Water applications. Since these applications does not have many method calls in the main computation loop, our algorithm reduces only few `if` instructions that do not have big influence on the overall performance of the whole application.

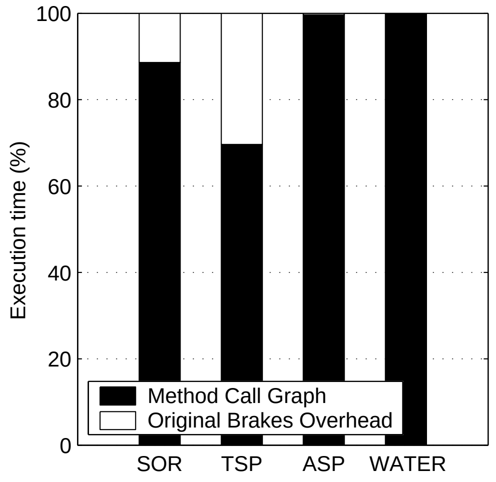


Figure 7.13: Methods call graph performance improvements

Table 7.1 shows the comparison between the total number of rewritten methods when using the original Brakes algorithm and our optimized version for all four described parallel algorithms. Note that many of saved method call transformations occur in the initialization code and therefore does not influence the performance.

Table 7.1: The comparison of the number of rewritten method invocations with and without the methods call graph analysis

Application	Original Brakes algorithm	Our algorithm
ASP	36	6
TSP	37	2
SOR	53	5
Shallow Water	52	5

The results clearly show that the original Brakes algorithm does much unnecessary method rewriting and thus causes too much performance overhead.

Chapter 8

Conclusions and Future Work

We have described a system for making parallel Java programs fault tolerant. Since the system was implemented in pure Java without any modifications in the Java Virtual Machine, it is portable across different Java Virtual Machine implementations and operating systems. Our system is transparent to a very high extent for the most popular class of parallel Java programs that take some input, compute, communicate between each other using the Java RMI, and yield a result.

In order to make an application fault tolerant, a programmer has to provide only an explicit code which calls the blocking `make checkpoint` method in all user threads on every computation node. The programmer may use a timer to request a checkpoint every some time or explicitly call it after finishing some computations to make them persistent. However, for more complicated parallel programs it may require some effort. When a thread is waiting on a semaphore for data from other nodes, it may happen that it will never wake up if those nodes are already blocked on the `make checkpoint` request. It may happen for both local and remote threads. In order to prevent such a situation, it is sometimes necessary to wake up all threads before making a checkpoint. In the future it would be convenient to eliminate this impediment. It might be possible to implement a clever bytecode post-processor which uses heuristic techniques to insert an appropriate code to prevent such situations in a way transparent to the programmer.

Our system cannot deal with standard Java classes that are not serializable. Among others they include classes for accessing sockets, files and operating system resources. When our state capturing algorithm encounters an instance of one of these classes on its way (on the Java stack), an exception is raised and the checkpoint request fails. Similarly, instances of these classes cannot appear as object's variables, because serialization of such an object causes an error. In our system, the programmer has to take special precautions when using non-serializable classes. Instances of these classes can be used only inside methods that never appear on the Java stack when state capturing is initiated. In the future, it would be convenient to improve our system, such that a programmer would not have to care about it. Paper [17] describes an approach based on automatic generation of serializable *wrapper classes* for resource classes by using techniques similar to the compile-time reflection. The wrapper objects would have the same functionality, would be serializable and perform re-initialization procedures at the recovery. An application could use these wrapper classes instead of the original classes.

There is also a small limitation of the state capturing algorithm, which does not allow to initiate state capturing during the execution of an exception handler. This limitation was described in section 2.5 and should be fixed.

The big advantage of our system is the ability to resist a crash of even a number of processors at the same time. Checkpoints may be duplicated on several machines and all external components, like the RMI registry are replicated for fault tolerance. The upper limit on the number of processors which can fail between two successive global checkpoints is configurable, but setting it too high may influence the performance of the system.

Since checkpoints can be initiated also inside remote method calls, our system can be integrated with applications that use RMI for something more than data transfer. Part of the framework responsible for the mentioned functionality is the client side stub postprocessor.

Finally, the tests show that our system has very low performance overhead during the normal program execution (without making checkpoints) which even for complicated parallel programs usually does not exceed 10% and for many programs is not visible at all. The overhead of making a global checkpoint is also very low assuming that it is made in reasonable time intervals — it is increasing very slowly with the number of processors. The recovery time, although it is not that important assuming that faults are not too frequent, is much worse but can be improved in the future by reducing the number of disk accesses.

We believe that the system we developed can be used with success for large-scale parallel computing.

Bibliography

[1] H.E. Bal et al. *The distributed ASCI supercomputer project*, ACM Special Interest Group, Operating Systems Review, Vol. 34, No. 4, p 76-96, October, 2000.

[2] S. Bouchenak. *Making Java Applications Mobile or Persistent*, In Proceedings of COOTS'01, San Antonio, Texas, USA, 2001.

[3] R. Clark, D.E. Jensen and F.D. Reynolds, *An Architectural Overview of the Alpha Real-time Distributed Kernel*, in Proceeding of the USENIX Workshop on Microkernel and Other Kernel Architectures, April 1992.

[4] T. Coninx, E. Truyen, B. Vanhaute, Y. Berbers, W. Joosen and P. Verbaeten. *On the use of Threads in Mobile Object Systems.*, Proc. of 6th ECOOP, 2000.

[5] S. Fuenfrocken. *Transparent Migration of Java-Based Mobile Agents*, in Proceeding of MA'98, pages 26-37, 1998.

[6] B. Haumacher, T. Moschny, J. Reuter, W.F. Tichy. *Transparent Distributed Threads for Java*, 5th International Workshop on Java For Parallel and Distributed Computing, April 2003.

[7] T. Illman, T. Krueger, F. Kargl, M. Weber. *Transparent Migration of Mobile Agents Using the Java Platform Debugger Architecture.*, Proceedings of the MA'01, Atlanta, USA, December 2001.

[8] A. Nguyen-Tuong, *Integrating Fault-Tolerance Techniques in Grid Applications*, Ph.D. Thesis, August 2000.

[9] R. van Nieuwpoort, J. Maassen, R. Hofman, T. Kielmann, H.E. Bal. *Ibis: An Efficient Java-based Grid Programming Environment*, Accepted for publication in *Joint ACM Java Grande - ISCOPE 2002 Conference*, Seattle, Washington, USA, November 3-5, 2002.

[10] T. Sakamoto, T. Sekiguchi and A. Yonezawa. *Bytecode Transformation for Portable Thread Migration in Java*, In Proceedings of ASAMA'2000, Springer, Zuerich, Germany, 2000.

[11] T. Sekiguchi, H. Masuhara, A. Yonezawa. *A Simple Extension of Java Language for Controllable Transparent Migration and its Portable Implementation*, 3rd International Conference on Coordination Models and Languages, April 1999.

[12] Sun Microsystems, *Java Remote Method Invocation Specification*, 2002. Revision 1.8, JDK 1.4.

[13] N. Suri, J. Bradshaw, M. Breedy, P. Groth, A.G. Hill, R. Jeffers. *Strong Mobility and Fine-Grained Resource Control in NOMADS*, In Proceedings of ASAMA'2000, Springer, Zuerich, Germany, 2000.

[14] A.S. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*, Prentice Hall, 2002.

[15] Voyager Core Package Technical Overview, 1997. ObjectSpace Inc.

[16] D. Weyns, E. Truyen, P. Verbaeten. *Distributed Threads in Java*, Accepted for publication at the International Symposium on Distributed and Parallel Computing, ISDPC 2002, Iasi, Romania, July 2002.

[17] H. Yamauchi, H. Masuhara, D. Hoshina, T. Sekiguchi, A. Yonezawa. *Wrapping Class Libraries for Migration-Transparent Resource Access by Using Compile-Time Reflection*, In Proceedings of Workshop on Reflective Middleware (RM2000), pp.19-20, New York, April 2000.