IJISET - International Journal of Innovative Science, Engineering & Technology, Vol. 2 Issue 12, December 2015.
www.ijiset.com
ISSN 2348 – 7968

A continuation-based pervasive computing system using java's checkpointing and recovery techniques.

[1]Dr. Nlerum Promise. A. And [2]Dr. Onuodu Friday E.

[1] Department of Computer Science, Faculty of Science, Federal University Otuoke, Bayelsa State

[2]Department of Computer Science, Faculty of Science, University of Port Harcourt, Rivers State, Nigeria.
e-mail: [1]promebsys@gmail.com [2]gonuodu@gmail.com

Abstract

Check-pointing, recovery and process continuity are inherent issues in Pervasive Computing as they are integral parts in process continuity in pervasive environments. Current research on pervasive systems' design does not consider the execution environment of the threads and this is one of the reasons why continuation has not been addressed by the pervasive community. Equipping high-performance pervasive computing systems with check-pointing and recovery mechanisms allows to minimize work loss in presence of failure. This paper discusses concepts for check-pointing and recovery of continuation-based Pervasive computing systems using Pervasive Java Virtual Machines (PJVM). Check-pointing a PJVM is to capture the execution state of the PJVM and to make this state continuous. In order to recover the check-pointed execution state, it is necessary to reproduce the execution state and to resume the PJVM. In other words check-pointing and recovery allows the reconstruction of a PJVM at any arbitrary execution state. The check-pointing and recovery concepts have been prototypically implemented. The prototype provides generic mechanisms for extracting an execution state from a running PJVM and for initializing a PJVM with a continuous execution state. This prototype system is called Pervasive Computing Continuation-based System's Architecture, (PECCSA). These techniques can be used for a wide range of applications. Our prototype also uses these mechanisms for PJVM migration (relocation of a running PJVM from one pervasive environment to another)

1. Introduction

This paper discusses concepts for cheek-pointing and recovery of an active pervasive environment on a Java virtual Machine (PJVM) platform. Check-pointing a PJVM is to capture the execution state of the PJVM and to make this state continuous [8, 14]. In order to recover the check-pointed execution state, it is necessary to reproduce the execution state and then to resume the PJVM. In other words check-pointing and recovery allows the reconstruction of a PJVM at any arbitrary execution state and thus allows to minimize work loss in presence of failure.

Check-pointing and recovery mechanisms are well-suited for number crunching services or long-lived computations such as calculations in physics, computations for weather forecasts or performing an On-Line Analytical Processing (OLAP)[9]. Let us assume a service computing highly complex calculations in physics and some hours later, the computer crashes for unknown reasons or has to be shut down for some system administration purposes. In such a case all the work that has been done by the computer is lost, unless the computation was periodically check-pointed. In order to recover the computation and thus to avoid loss of work, a new PJVM may be started and initialized with the last check-pointed execution state. These concepts have been prototypically implemented in PECCSA.

This paper is organized as follows. Section 2 gives a short overview of related work. Section 3 discusses which run-time data is contained in an execution state of a running PJVM. In section 4 the problem of providing type information for local variables will be examined. Next, section 5 explains why a PJVM has to be in a mobile state in order to start a migration. Section 6 describes the concepts for check-pointing and recovery mechanisms. Section 7 describes how the prototype PECCSA is implemented. It discusses the extension of the PJVM and gives an overview of the PECCSA API.

1.1 REPRESENTATION OF THE EXECUTION STATE OF THE PJVM

The representation of the execution state of the PJVM is called *snapshot.* A snapshot consists of enough run-time data in order for the PJVM to be correctly reproduced on

the target computer. The execution state of a PJVM contains the following:

1. Heap: The heap is the memory area, where the dynamically allocated memory space is stored, i.e. Java objects and arrays. In order to include the heap into the snapshot, the PJVM makes use of the object serialization mechanisms. Object serialization allows the transformation of an object (graph) to an array of bytes[3].

2. All threads: The PJVM is multi-threaded and thus may host multiple threads. Simply including all the threads is not enough. It is further necessary to consider the execution environment of the threads. The execution environment of a thread comprises a Java stack (also known as call stack), Java frames (also known as activation frame), the local variables and the operand stacks[5].

3. Contents of the class files: The static information about a Java class such as the class name or the names of the methods are stored in a class file. Before loading this information into the memory the PJVM verifies the bytecodes of the corresponding class file. If the class file passes the verification, the PJVM loads it into memory.

Snapshot, however, does not need to take into account the complete execution state. It must contain the entire heap and all the threads (including their execution environments) but does not need to contain the contents of the class files, as many of them - especially those of the core Java classes such as java.lang [6]. System or java.lang.String - are provided on any Java installation. [4, 5].

Any data that is included in the snapshot is a regular Java value as specified in [10, 6]. This ensures that any PJVM implementation providing the migration facilities may handle the snapshot, as all data conforms to the regular Java data types.

Furthermore, it is necessary to associate the run-time data with type information. The necessity stems from the fact that the data types have different byte representations. If the run-time data is associated with type information the target PJVM knows which byte formats it has to read from the representation.

In order to correctly reproduce the frozen execution state on the target computer it is necessary to indicate the structure of the snapshot by means of additional type codes. These type codes indicate the structure of the snapshot such as the beginning and the end of a thread, a Java stack, or a Java frame.

4. MANAGING TYPE INFORMATION

In order to create a representation of the execution state, it is necessary to determine the type of the local variables and of the items on the operand stacks. Hereafter, we summarize the local variables and the operand stacks as *memory cells.* The memory cells are modified only if the PJVM executes Java instructions. As the operation code of a Java instruction indicates the type of the operands, e.g. iadd, ladd, fadd, or dadd, it is possible to manage type information quite straightforwardly[1, 2].

The way the execution of Java instructions modify the memory cells is specified in [10]. There are two approaches how the type information can be provided by the PJVM:

Actualizing during run-time: The type information of the memory cells is actualized on every execution of a Java instruction. Consequently, when a migration is requested the type information of the memory cells is provided.

Computing on demand: The type information of the memory cells is computed only on a migration request. In order to determine the type information the PJVM performs the following steps:

1. During the first step the PJVM performs a *control flow analysis* on the bytecode, as a result a control flow graph is created.

2. In the subsequent step the PJVM evaluates the instruction path between the method entry point and the

program counter (pc) based on this control flow graph. Knowing the instruction path, it is possible to gather type information of the corresponding memory cells by actualizing the type information for each instruction

within the path. This step is called *data flow analysis.*
After having finished both steps the PJVM provides the type information of the memory cells of a Java frame. However, as the PJVM is multi-threaded these steps have to be performed for every thread and every Java frame.
Actualizing the type information during the execution of a program strongly impairs the performance of the PJVM. The more Java instructions are executed, the slower is the execution of the PJVM[15]. As most Java instructions have an effect on the local variables and/or the operand stacks, the decline in speed is considerable.
The second approach only penalizes Java programs that make use of PJVM migration capabilities, i.e., only those programs which request a PJVM migration. Due to the computation of the type information, this approach causes latency[4, 15].
PECCSA implements the second approach, because we believe that Java programs that do not make use of the PJVM migration capabilities must not pay a penalty for the actualization of the type information.

5. MOBILE STATE

When a pervasive java program is migrated, the PJVM has to be frozen in a well-defined execution state in order for the frozen execution state to be reproduced on the target computer.
During execution, a thread executes the Java instructions contained in the bytecode.
The Java instructions are instructions of an abstract computer, namely of the PJVM. Thus, in order to perform a Java instruction' the PJVM executes several native instructions.
Depending on the underlying computer architecture and the pervasive environment, the sequences of the native instruction for the same Java instruction differ.
A thread, however, must not be suspended when it is executing native instructions because its execution state cannot be reproduced.

Unfortunately, the underlying operating system may suspend a thread while the thread is executing a Java instruction. Therefore the PJVM has to make sure that a thread is only suspended at a preemption point. We call a PJVM to be in a *mobile state* if all threads are suspended at a preemption point.
In order to determine the preemption point, we have to take a look at the PJVM's instruction cycle (see figure 1). The simplified Java instruction cycle comprises two stages:
1. Fetch stage: During this stage, the PJVM fetches and decodes the Java instruction. If the Java instruction comprises one or more operands the PJVM fetches and decodes them as well.

2. Execution stage: The PJVM executes the decoded Java instruction and updates the program counter (pc) of the PJVM. The PJVM actually executes several native instructions.
A thread reaches a preemption point if it has finished executing the current Java instruction and has not fetched the subsequent Java instruction. In other words, the PJVM instruction cycle comprises one preemption point (see figure 1).

ISSN 2348 – 7968

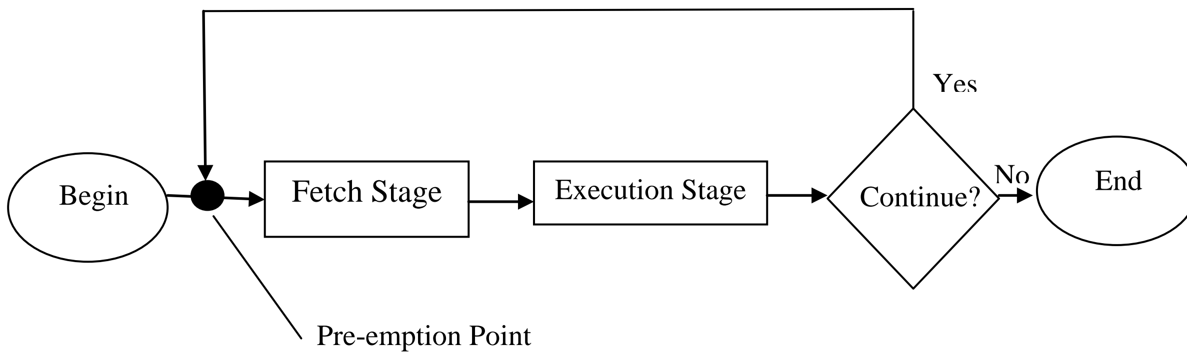


Figure 1: Simplified instruction cycle

Besides ensuring that every thread suspends at a preemption point, it is further necessary to perform an additional check.

The PJVM has to guarantee that no Java stack consists of a native Java frame, i.e. no thread is executing a native method. We call a PJVM to be in a mobile state if all threads are frozen at a preemption point and if there is. no active native method invocation.

6. CHECKPOINTING AND RECOVERY

To checkpoint a PJVM means to extract the execution state of the PJVM and to make the snapshot continuous, e.g. to write it to a file. The persistency of a snapshot is an important characteristic because the snapshot is used for a recovery sometime later. That is, the PJVM is initialized with the execution state contained in the snapshot[9,14].

6.1 Entering into Mobile State

A PJVM may host several threads running concurrently. In order for a PJVM to reach a mobile state, it is essential to suspend all threads at a preemption point.

On a migration or in this case, on a check-pointing request, the PJVM immediately creates a temporary thread called SnapshotGenerator [7].

This thread is responsible for generating a snapshot. Its necessity relies on the fact that the PJVM, once in a mobile state, must not resume the execution.

That is, the execution state has to be kept unchanged. If a thread takes over this job that existed before the migration request, the execution state would be changed. This is because PECCSA uses mechanisms written in Java to write to a snapshot or read from a snapshot.

The SnapshotGenerator thread is responsible for the control and the exception handling during the preparation phase. It is responsible for:

• monitoring the number of threads suspended at a preemption point -

• observing the elapsed time until the PJVM reaches the mobile state -

• writing all relevant run-time information to a snapshot.

After the SnapshotGenerator is created, it has to guarantee the correct suspension of the threads. This is how it works. The SnapshotGenerator notifies all other threads (expect the SnapshotGenerator) that a migration has been requested. It sets the state of the other threads to '~pending migration" and suspends until the PJVM reaches the mobile state. Every other thread completes the currently executing Java instruction and suspends at the next preemption point.

In order to notify the SnapshotGenerator that all other threads are suspended at a preemption point, we define a counter variable which is shared among these threads.

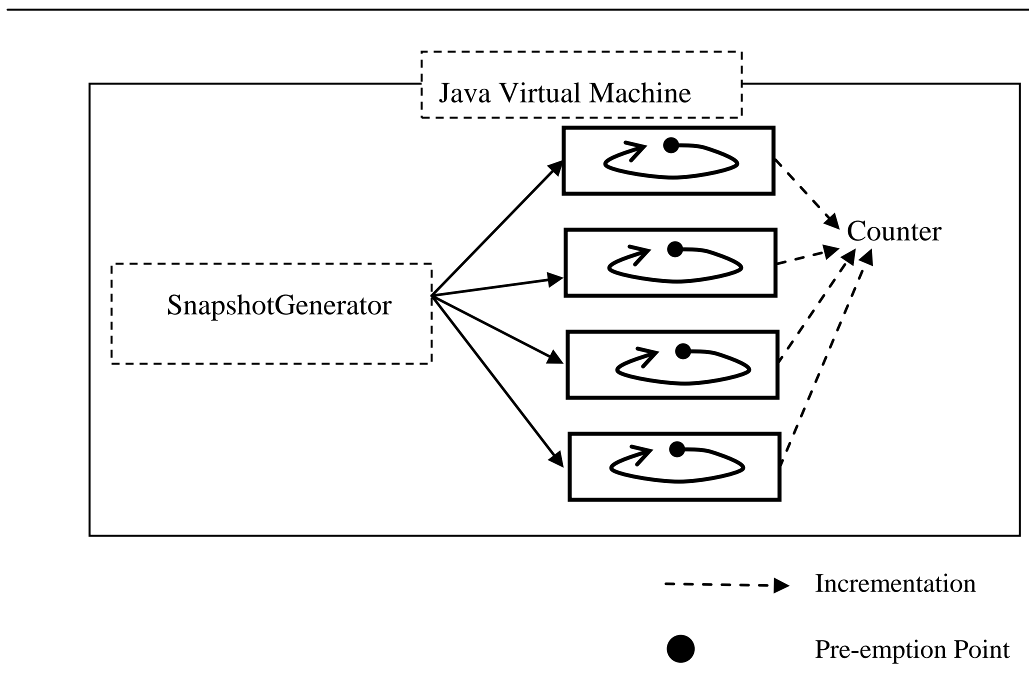


Figure 2: Simplified figure of a PJVM in a mobile state

Every thread that reaches the preemption point increments the counter by one. If this counter reaches the required value, i.e. total number of threads minus one the SnapshotGenerator has to be notified.

We use a technique called conditional variables [11]. A conditional variable is a technique that can be used to implement conditional critical regions. A conditional variable is an associated mutez (mutual exclusion) variable [7, 8] and indicates a Boolean state of that variable. In our case, the conditional variable is true if the number of the threads suspended at a preemption point is equal the number of threads hosted by the PJVM minus one. Unfortunately, it may happen that a thread is not able to reach the next preemption point within an acceptable period of time, e.g. if a thread is waiting for user input. The reasons for this are the following:

- Thread is blocked: A thread may be blocked for synchronization purposes, if it executes the Java instruction monitorenter, monitorenter makes a system call and blocks the thread until the monitor is released by some other thread.
- Thread is executing a native function: During the execution of a native function, the PJVM does not execute bytecodes and therefore the PJVM releases the control over the execution.

Thread is in a waiting queue: A queued thread waits to be dispatched in order to resume its execution. Because the thread scheduling is not entirely specified, we do not know whether the scheduler or the dispatcher may

ISSN 2348 – 7968

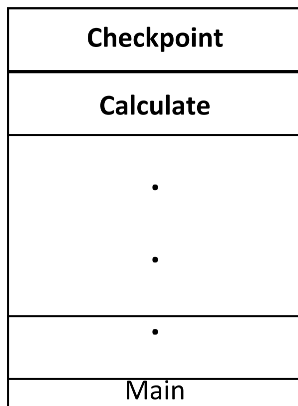
"suspend" a thread while being in the execution stage of the Java instruction cycle (see figure 1).

To omit the PJVM waiting during an arbitrary period of time, PECCSA defines a time limit that indicates the maximal period for reaching the mobile state. If the time limit is exceeded, PECCSA throws a timeout exception and the migration or the checkpointing is cancelled. As soon as the PJVM reaches the mobile state within the specified time limit, it starts to create a snapshot (see section 6.2).

6.2 Generating a Snapshot

Once in a mobile state, the SnapshotGenerator thread continues its execution. It iterates over all threads and scans through all the Java frames on the Java stacks and writes all relevant information to the snapshot. The control flow and the data flow analyses were implemented according to the algorithm described in [1, 2]. All data is written using the primitive data serialization and object serialization mechanisms.

Java Stack While Checkpointing



Java stack written into snapshot

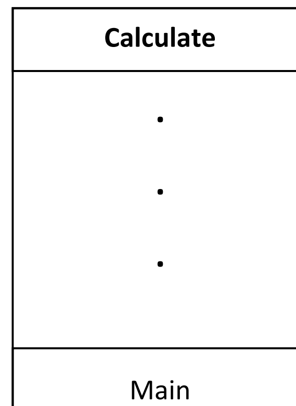


Figure 3: Java stack of the thread initiating the check-pointing

The SnapshotGenerator thread processes the thread that initiated the check-pointing (or migration) differently. The top most Java frame of this thread is ignored and is consequently not written to the snapshot. If the SnapshotGenerator considered the top most Java frame, the PJVM would enter into an endless loop. In other words, the PJVM would invoke the method checkpoint anew. The omission of the top most Java frame guarantees the correct continuation of the Java program.

6.3 Initializing a PJVM with a Snapshot

In order to recover a Java program, it is necessary to create a new instance of the PJVM and to initialize it with this snapshot. After the bootstrap of the new PJVM, but before the execution of the first bytecode, the PJVM creates a PJVMInitializer thread which is responsible for the initialization process.

The lifetime of this thread is limited to the preparation phase. After the creation of the PJVMInitializer thread all other threads are suspended. Like in the preparation phase, these threads have to be resumed at a preemption point (see section 6.1). If the PJVM reaches the mobile state, i.e. all other threads are suspended at a preemption point, the PJVMInitializer starts to initialize the PJVM run-time information corresponding to the snapshot.

For each thread contained in a snapshot, a Java stack is created and Java frames initialized. These Java frames are then pushed onto the Java stack. The local variables and the operand stack are initialized as well. For every

ISSN 2348 – 7968

class that has not been cached in the constant pool, the PJVM verifies the corresponding class file and caches it in the constant pool.

To avoid any synchronization failure among the threads, all threads except the PJVMInitializer stay suspended until the completion of the PJVM initialization process. After this initialization, the PJVM resumes its execution. The PJVMInitializer thread exits after the PJVM is completely initialized.

7. IMPLEMENTATION ISSUES

PECCSA is an extended Java platform that provides functionalities for migrating a running PJVM to a target machine. Therefore, the standard PJVM had to be extended by some functionality which is described later in this section.

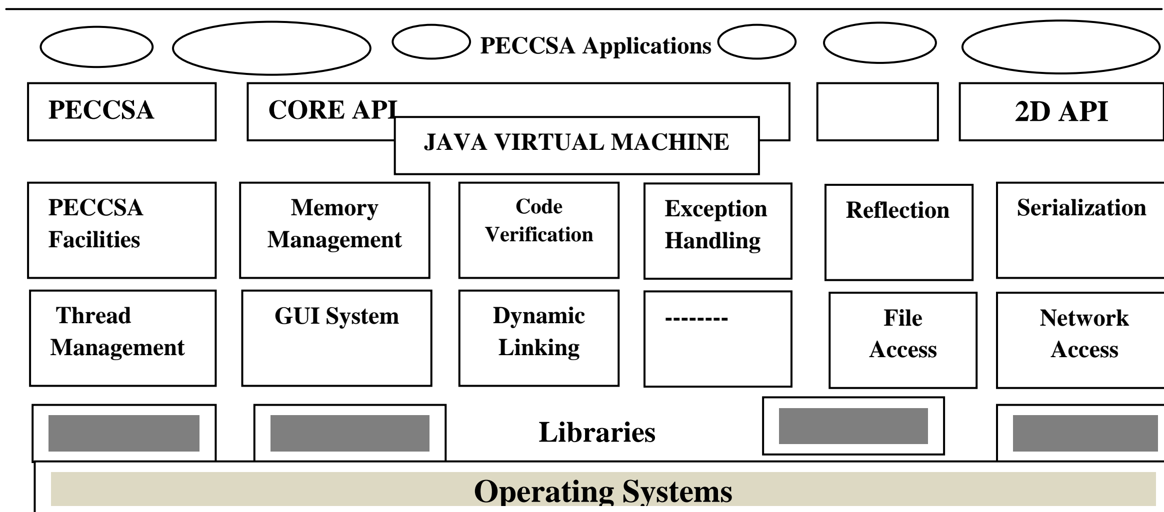


Fig. 4: ARCHITECTURE OF PECCSA

7.1 Extension of the PJVM

The PJVM is actually a black box, i.e. it is not possible to have either an explicit read- or write-access to the PJVM's run-time data such as the Java stacks or the operand stacks.

We therefore extended the PJVM with the following functionalities:

1. Entering into a mobile state: On a migration request the PJVM has to make sure to reach a mobile state.

2. Providing type information: As discussed earlier in this paper (see section 4) it is necessary to provide type information of the memory cells. PECCSA implements the computing-on-demand approach.

3. Generating a snapshot: The PJVM generates a snapshot that includes the entire heap and all threads.

4. Initializing a PJVM with a snapshot: The target PJVM is initialized with a snapshot.

The latter two functionalities were implemented as generic mechanisms that may be used not only for check-pointing and recovery of Java programs. These mechanisms were used for PJVM migration purposes as well.

7.2 The PECCSA API

PECCSA provides a thin API that comprises mechanisms for check-pointing, recovery and migration of a PJVM.

The design goal for the PECCSA API was to keep it as small and as simple as possible. In effect, the PECCSA API is very simple to use and the application programmer only has to be concerned about 6 Java classes. All PECCSA related Java classes are organized in the java.PECCSA package.

ISSN 2348 – 7968

The PJVM class represents the underlying Java virtual machine (see table 1). As a PJVM may only run one Java program at the same time, we decided to implement the PJVM class as a singleton class [3,12]. All the methods of the PJVM class are static and this class may not be instantiated.

Table 1: Public methods of PJVM

final public class PJVM {
public static void checkpoint(CheckpointProtocol prot)
throws PECCSAException;
public static void migrate(MigrOutProtocol prot)
throws PECCSAException;
}

Furthermore, PJVM is a creator class of the snapshot, that is the class PJVMSnapshot does not provide its own constructor and therefore, it cannot be created by any other class except by PJVN [8]. Actually, the existence of the class PJVMSnapshot is transparent to the Java programmer.

7.2.1 Checkpointing and Recovery

The method checkpoint () generates a snapshot and makes it continuous. There are various ways to make a snapshot continuous. The checkpoint( ) method therefore delegates the behaviour to the Java interface CheckpointProtocol.

This interface is the formal parameter of the checkpoint method and comprises the template method check-point().

In order to recover a PJVM PECCSA provides a tool called PJVMRecovery. This object delegates the recovery process to the Java interface called RecoveryProtocol (see figure 5).

PECCSA provides a check-pointing/recovery protocol which is implemented by the following classes:

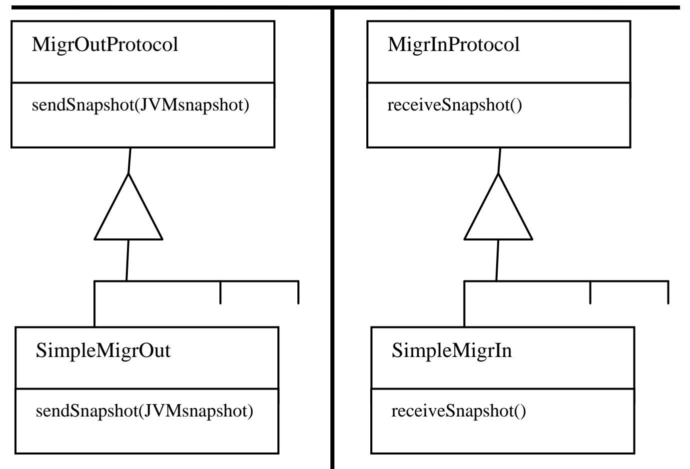


Figure 5: Class diagram of CheckpointProtocol and RecoveryProtocol

• java.PECCSA.SimpleCheckpointProto extends the checkpoint protocol class which is depicted in figure 5. The method checkpoint extracts the execution state and writes its representation, i.e. a snapshot, to a file.

• java.PECCSA.SimpleEecoveryProto extends the recovery protocol class which is depicted in figure 5. The method recover reads a snapshot from a file, initializes the PJVM with it, and resumes the PJVM. Both methods throw exceptions if the underlying PJVM detects a fault.

7.2.2 Migration

The method migrate() transparently generates a snapshot and migrates the PJVM to a target computer in one step. The formal parameter of this method is a Java interface called Migr0utProtocol. The migrate() method delegates the behaviour of the source PJVM to this interface during the transmission phase. This interface specifies the skeleton of the source-side transmission protocol and comprises a template method sendSnapshot ().

ISSN 2348 – 7968

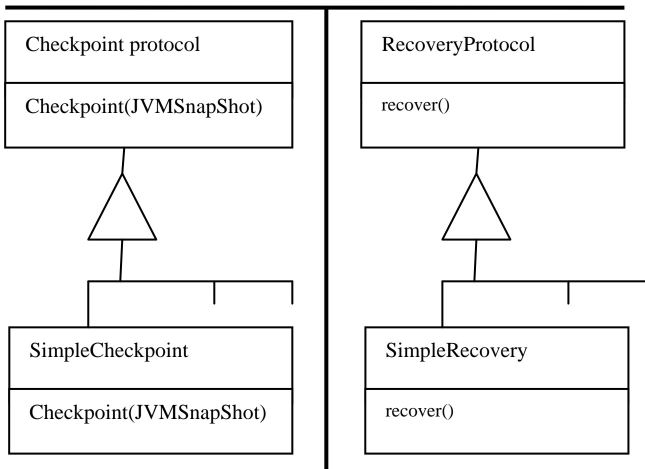


Fig. 6: Class diagram MigOutProtocol and MigrInProtocol

The corresponding interface of the migration protocol of the target-side is called MigrInProtocol. The PECCSA demon which runs on every target computer delegates the behavior of the target PJVM to this Java interface during the transmission phase. MigrInProtocol also comprises a template method which is called receiveSnapshot ().

PECCSA provides some simple migration protocols called SimpleMigrOut for the source computer and SimpleMigrIn for the target computer.

7.2.3 Check-pointing and Recovery Example

Let us assume a Java program performing complex and long computations such as a weather forecast analysis. In order to avoid losing computer work in case of a crash the program is check-pointed periodically.

At first, it is necessary to create a check-pointing protocol object that implements where and how the snapshot shall be made continuous [13]. In this example, the snapshot is written into a file using java.PECCSA. SimpleCheckpointProto (see table 2). The name of the file to which the snapshot is written is composed of the name of the program and a time stamp.

Table 2: Checkpointing a PJVM

```
1  . . .
2  try{
3  SimpleCheckpointProto proto = null;
4  proto = new SimpleCheckpointProto(prog +"-"+
        time);
5  PJVM.checkpoint( proto );
6  }catch(PECCSAException me){
7  System.out.println(me.getMessage() );
8  } catch (IOException ie){
9  System.out.println(ie.getMessage() );
1o  }
11  ...
```

In order to recover a crashed program, we may use the simple tool called java.PECCSA. PJVMRecovery. This tool allows a new PJVM to be started and initialized with a snapshot.

It can be started from the command shell. Thus, we type the following into the command shell[12] (see table 3).

Table 3: Recovering a PJVM from a Unix shell

```
<ii capri test/migr> PJVMRecovery <prot name> <arg list>
```

The name of the recovery protocol is SimplelRecoveryProto and the snapshot we would like the PJVM to be initialized with:

is FreezeTester-9.

8. PERFORMANCES

In the following we give an idea of the cost of time for a PJVM migration. We measured two aspects:

1. How is the performance of the PJVMs implementing the approaches discussed in section 4?
2. How long does it take to create a snapshot?
3. How long does it take to initialize a PJVM with a snapshot?

All measurements are based on a Java program that allocates a certain number of threads These threads loop endlessly.

ISSN 2348 – 7968

The measurements were produced for 1, 10, 20, and 40 threads (the thread that runs the main method is not counted, but is check-pointed as well). In order to get reliable time measurements we ran the Java program 50 times for each number of threads. The times which are presented in this paper axe mean values of the 50 runs.

8.1 Benchmarking the PJVM

PECCSA implementation is based on the Sun PJVM v1.1.7. As the official PJVM, i.e. java, is partly implemented in assembler language, we switched over to the debugger version of the PJVM, i.e. java_g which is written in C/C++. The performance gap between java and java_g is tremendous. Thus an implementation of PECCSA based on the assembler PJVM would considerably improve the performance of the current version of PECCSA.

These PJVMs are benchmarked using the official PJVM benchmark released by SPEC and is called SPEC JVM985.

Table 4: Benchmarking the PJVM

Benchmark	PJVM 1.1.7 g (sec.)	PECCSA ADR (sec.)	PECCSA COD (sec.)
_200_Check	**0000.676**	**0000.934**	**0000.855**
_227_mtrt	**1392.498**	**1962.578**	**1661.883**
_202_jess	**1096.585**	**1672.114**	**1349.894**
_201_compress	**5756.637**	**9572.148**	**6056.326**
_209_db	**2163.247**	**3278.479**	**2447.532**
_222_mpegaudio	**4993.922**	**8298.367**	**5201.140**
_2.28_jack	**1398.741**	**2587.683**	**1554.928**
_213_javac	**1304.241**	**1966.839**	**1496.943**

As listed in table 4, the PJVM implementing the approach "actualizing-during-run-time" (ADR) pays a penalty of 35 - 85% which is not acceptable.

The benchmarks of the PECCSA PJVM implementing "computing on-demand" (COD) are slightly worse than those of the java_g but the slowdown is neglectable.

8.2 Creating a Snapshot

In order to create a snapshot the PJVM first has to reach a mobile state. Once in a mobile state it starts to create a snapshot by performing control flow and data flow analyses. The measurements are listed in table 5

Table 5: Performance of CF- and DF- Analysis

No. of Threads	Generate a Snapshot (sec.)	Get into mobile state (sec.)
1	**0.8378**	**0.000160**
10	**0.9295**	**0.000511**
20	**1.0032**	**0.000860**
40	**1.1499**	**0.001641**

8.3 Initializing the PJVM

The initialization of the PJVM with a snapshot is the last step that has to be performed during a migration. The time measurements listed in table 6 do not include the time elapsed to bootstrap the PJVM. The time measurements quantify the period between the moment in which the bootstrap is started and the initialization of the PJVM with a snapshot is completed.

Table 6: Measurements on the i n l t i a l i z a t i o n of a PJVM with a snapshot

No. of Threads	Estimated Time to initialize a PJVM
1	**0.7350682**
10	**0.8214124**
20	**0.9143485**
40	**1.1940306**

.

9. CONCLUSION

Mechanisms for check-pointing and recovery of running PJVM are a powerful technique for pervasive computing applications. They are well-suited for the development of fault-tolerant high performance computing services.

A central question is which run-time data should represent the execution state of a PJVM. A portable representation allows the cross-platform migration of running pervasive computing virtual machines, i.e. the

migration between platforms running in different pervasive environments (having different operating systems and computer architectures).

The generic mechanisms that encompass the extraction of an execution state from a PJVM and the importation of an execution state into a PJVM cannot be used for check-pointing/ recovery purposes only. They can be used for a wide range of applications such as PJVM migration. PJVM migration allows to migrate a running PJVM from a source computer to a target computer where it resumes execution exactly at that point, where it was frozen on the source computer.

Concerning the approaches for actualizing run-time type information, we have found that actualizing the type information during run-time is not acceptable for two reasons. *First,* all applications that are executed by the virtual machine pay a penalty for the actualization of the type information, even if the applications do not use the migration facilities. *Second, as* the type information has to be actualized on every execution of an instruction, the slowdown of the Java virtual machine is tremendous.

The approach which computes the type information on demand is preferable because it only affects those applications which make use of the migration facilities. Furthermore, the latency of the migration due to the additional computations is reasonable.

10. FUTURE WORK

Presently, the external state is not under control of PECCSA. With external states we address the native graphic system, e.g. the AWT peer state, socket state or connection information. Our future work will focus on handling external state by PECCSA. A possible technique to handle external state is to define callbacks which are made on critical system events based on the Java event model.

The performance of PECCSA shall be improved. We therefore are considering two approaches. Porting of PECCSA to the PJVM written in assembler essentially improves the performance of our system. Additionally, the adaptation of the PECCSA concepts to a PJVM environment equipped with a JIT (Just In Time) will be considered as well. This adaptation requires additional changes, because the execution state of a PJVM that runs JIT generated codes is not reproducible[5, 10]. Thus, the JIT environment has to be modified if the additional preemption points have to be introduced within the native instruction cycle [15].

11. REFERENCES

[1] Aho, A. V., Sethi, A., and Ullman, J. D.(1986), *Compiler Principles, Techniques, and Tools.* Addison-Wesley.

[2] Appel, A. W., (1998) *Modern Compiler Implementation in Java.* Cambridge University Press.

[3] Gamma, E., Helm, A., Johnson, R., and Vlissides, J (1994), *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

[4] Gosling, J., And Yellin, F. (1996) *The Java Application Programming Interface (Java Series).* Addison-Wesley Pub Co.

[5] Gosling, J., and Yellin, F. (1996) *The Java Application Programming Interface: Window Toolkit and Applets (Java Series).* Addison-Wesley Pub Co, June 1996.

[6] Jordan, M., and Atkinson, M. (1999). *Orthogonal Persistence for the Java Platform* – Draft Specification. Tech. rep., Sun Microsystems Laboratories, M/S UMTV 29-01,901 San Antonio Road, Palo Alto, CA 94043 USA.

[7] Lamport, L. (1986) *The Mutual Exclusion Problem:* Part I -- A Theory of Interprocess Communication. Journal ot the ACM 33(2), pp 313 - 326.

[8] Hadi, H. and Rasool, J. (2010). *Self-reconfiguration in Highly Available Pervasive Computing System*. Simula Research Laboratory, P.O. Box 134, NO-1325 Lysaker, Norway.

[9] Cao, G and Singhal, M, 2001: "*Mutable Checkpoints: A New Checkpointing Approach for Mobile Computing Systems*", IEEE Transactions On

Parallel And Distributed Systems,Vol.12,No.2, February 2001,pp 157-172.

[10] Lindholm, T., and Yellin, F. (1997) *The Java TM Virtual Machine Specification (Java Series).* Addison-Wesley.

[11] Silberschatz, A., Peterson, J., and Galvin, P. (1991) *Operating System Concepts,* 3rd ed. Addison-Wesley.

[12] Eric. Van Wyk, Lijesh Krishnan, Derek Bodin, and August Schwerdfeger, (2007). *Attribute Grammar-Based Language Extensions for Java.* Lecture Notes in Computer Science, 4609:575.

[13] Chaudhary V. and Jiang, H. (2006). *Techniques for Migrating Computations on the Grid*. Engineering the Grid: Status and Perspective.

[14] Lalit K. A. and Kumar P. (2007): "*A Synchronous Check-pointing Protocol For Mobile Distributed Systems :Probabilistic Approach"*. Int. Journal. Information and Computer Security, Vol.1, No.3, pp 298-314.

[15] Ungar, D , Spitz, A and A. Ausch (2005). *Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment.* In Proceedings of the Companion to theObject-Oriented Programming, Systems, Languages, and Applications Conference, pp 11–20, San Diego.