Operating System support for Java

Alan Dearle, David Hulse, Alex Farkas
{al,dave,alex}@cs.stir.ac.uk

Department of Computing Science
University of Stirling
Stirling
FK9 4LA

Abstract

Over the last 15 years a number of persistent language systems have been produced whose implementation relies on the persistence mechanisms provided by an underlying operating system. We have produced an operating system called Grasshopper expressly designed to support orthogonally persistent systems. In this paper we demonstrate how the mechanisms provided by Grasshopper may be used to implement a persistent version of the language Java.

1. Introduction

Over the last ten to fifteen years we have seen a number of persistent language systems. Some of these were designed to be persistent from the outset [3, 19], others are persistent versions of ordinary programming languages e.g. [21]. What these language systems have in common is that they have been implemented as separate language systems above a non-persistent operating system. It is our belief that, although persistence can be implemented by a programming language run time system, the persistence of data should be provided by the operating system.

Implementing persistence at the programming language level suffers from two major drawbacks. First, the host operating system was not designed to support persistence; therefore the operating system interface does not usually provide abstractions sympathetic to a persistent language implementation. The consequence of this is that the language designer is usually forced to implement a persistent abstract machine above the operating system abstractions, resulting in a loss of efficiency. A similar problem is reported by the designers of database systems [22].

The second problem with this approach is that every persistent language implements its own persistent abstract machine duplicating much of the functionality found inside the operating system and other language implementations. Often these different implementations are entirely incompatible with each other, prohibiting interactions between programs written in different languages. This would appear to be a retrograde step compared to the mixed language environments supported by conventional systems.

The implementation of the orthogonal persistence abstraction by the operating system avoids these problems. We believe that such an approach to operating system design could be as revolutionary as virtual memory in terms of the advantages for user-level applications. We have constructed the Grasshopper operating system in order to investigate these assertions. In this paper we show how one non persistent language, Java [14], may be made persistent under

Grasshopper. In addition to this language system, our current Grasshopper prototype also supports persistent assembler, C, C++ and the language Napier88.

When considering the persistent storage requirements of Java we must consider six different aspects:

1. the persistent storage of Java source code,
2. the storage of Java executable code in class files,
3. Java object repositories,
4. binding to persistent objects,
5. the state of active Java threads, and
6. the recoverability of state following a failure.

In most persistent implementations, the first two categories would be handled by the (non persistent) operating system, the latter by some language layer. In Grasshopper, support for all six categories is provided by the operating system. The remainder of the paper discusses these six aspects.

2. Persistent storage of source and class files

The persistent storage of source and class files presents three requirements:

1. some persistent storage must be provided to store the data,
2. some naming mechanisms must be provided for finding the files, and
3. some protection mechanism must be provided to prevent unauthorised access to files and to prevent unauthorised modification.

In Grasshopper the first of these is satisfied by a single abstraction known as a *container* [11]. Containers are the only storage abstraction provided by Grasshopper and are persistent entities which fulfil the roles traditionally served by both address spaces and files. In most operating systems, the notion of a virtual address space is associated with an ephemeral entity, a process, which accesses data within that address space. In contrast, containers and processes (called *loci* in Grasshopper) are orthogonal concepts. A Grasshopper system consists of a number of containers which may have loci executing within them. Loci are migratory in nature and may move from container to container by *invoking* them. Each container that supports invocation has a well defined interface specified in IDL [20].

It is convenient, natural and efficient to implement Java source and class files using Grasshopper containers. The next problem is that of access to the data. For this purpose, Grasshopper provides attributed namespaces. Namespaces implement a mapping from names to Grasshopper *capabilities* [10]. In Grasshopper every object known to the operating system is represented at user level by one or more capabilities which consist of a reference to a Grasshopper entity and access rights associated with it. The capability system therefore implements both low level naming and protection. The capability mechanism is deliberately simple and low-level for reasons of efficiency and flexibility.

Namespaces are themselves implemented as containers containing both code and data. Namespaces present an abstract table interface similar to that supported by PS-algol. However,

the mappings may be associated with string, integer, boolean and capability attributes. Attributes are useful for representing modification dates, type information, ownership etc. Typically, namespaces are arranged in hierarchies as shown in Figure 1.

Some protection mechanism must be provided to prevent unauthorised access to files and to prevent unauthorised modification. In Grasshopper the first of these requirements is provided by the capability system. No Grasshopper entity can be accessed without the presentation of a capability. Although capabilities are stored in namespaces, not all namespaces are publicly accessible. Each user has his or her own namespace from which other namespaces are reachable. Users may choose to make their namespaces known in a secure manner using functions provided by the login service, however, this is beyond the scope of this paper.

Access to a container does not give unrestricted access to the data as is the case in Unix. The capability system may prevent invocation or mapping (see below) and the kernel supports the usual page level protection mechanisms preventing reading or writing of data stored in containers. The ability to change page protections is itself protected using the capability system thereby preventing unauthorised users from performing illegal operations.

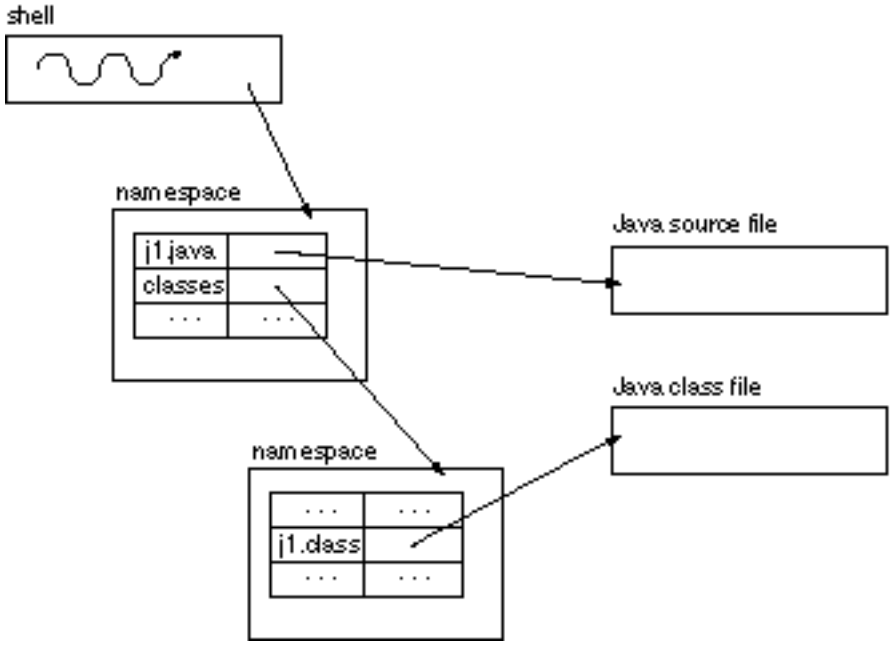

Figure 1: Grasshopper container organisation

3. Java object repositories

The persistent store against which Java programs operate (the object repository) may also be provided by a Grasshopper container. Containers may be organised in any manner required by the persistent application system using them. In the case of Java, the object repository is required to contain:

1. Java class files,
2. the Java run-time system (the interpreter),
3. stacks for threads,
4. at least one heap containing Java objects.

Grasshopper enables this via the provision of *container mappings*. The purpose of container mapping is to allow data to be shared between containers. This is achieved by allowing data in a region of one container to be viewed within a region of another container. Mappings may be either global, or visible only to a particular locus while executing in a particular container. Unlike other memory object mechanisms, containers may be arbitrarily (possibly recursively) composed using mapping which provides considerably enhanced flexibility and performance [16]. Another difference between mappings in Grasshopper and other systems is that mappings in Grasshopper are persistent and remain in force until the data is unmapped.

Using the container mapping mechanism, the Java run time system and the Java class files may be mapped into one or more object repositories; this is shown in Figure 2. This allows code to be shared in both virtual memory and on disk [16]. At this point is worth noting the C++ computations implementing the Java run time system (including global variables) are automatically persistent due to the fact that Grasshopper containers are truly orthogonally persistent address spaces. Note that the mapping mechanism is used because it is natural at the logical level, not as a mechanism with which to implement the recoverability of the persistent store [6].

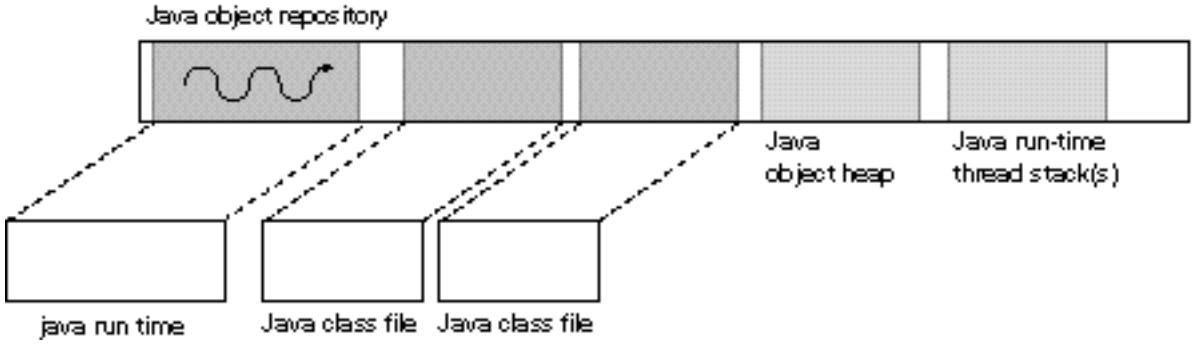

Figure 2: Mappings in the Java object repository

As described above, mappings may be established globally or at a per locus level. The latter provides a mechanism by which each thread (locus) may have its own stack and heap space which is invisible to all other loci in the system. Indeed, the stacks and heaps may be located at the same address within the container in which they are executing. This considerably simplifies memory management and adds a degree of inter-thread protection unavailable in most systems.

The technique of giving each concurrent persistent process its own local heap was used in the CASPER system [23]. In this system it was found that the use of *copy-out* techniques borrowed from generational garbage collection considerably reduced the work of garbage collecting the global heap. It is expected that similar results will be found in the case of Java whose architecture is very similar to that of Napier88 [8].

4. Binding

In all persistent systems, some mechanism must be provided to permit objects to bind to other objects in the persistent store. In systems such as Napier88 this is provided by the environment mechanism [9]. In Grasshopper, any locus can generate an arbitrary address in the container in which it is executing and access the data stored there. Such a mechanism is not particularly useful on its own, particularly in a language such as Java which does not support pointers. However, this simple mechanism can be used to implement an intra-container naming scheme similar to that provided by PS-algol [1] and Napier88 [9]. One of the Grasshopper libraries

contains an associative access (table) package which maps from strings to the C type `void*` (in fact this package is used in the implementation of namespaces described above). The placement of a table data structure at a known address in a container gives all the symbolic naming capability that is required for inter-compilation unit and compilation unit–persistent store binding. This mechanism is easily wrapped in some Java functionality to provide the `pJavaStore` functionality suggested by Atkinson et al. [4].

5. Persistent thread support

Java is inherently thread based. In the reference systems [18] released by Sun, two mechanisms are provided to support threads, namely *green* threads and threads implemented via Sun's Light-weight Process Library [17]. Green threads is a stand alone thread implementation for systems such as HP-UX-9.0 that do not provide thread support. This package uses timers and interrupts in a similar manner to LWP to implement threads at user level. The LWP library that has become a defacto standard amongst Unix vendors. Note that both approaches require a considerable amount of coding to be performed to make Java threads persistent.

The green threads package could be implemented on Grasshopper with appropriate changes to the system calls. Instead, Grasshopper loci may be used to implement Java threads. Multiple loci may execute within a single Grasshopper container and thus share state. The inability to do this in Unix processes was one of the original motivations for threads [5]. Like containers, loci are maintained by the Grasshopper kernel and are inherently persistent. Making loci inherently persistent greatly simplifies the management of orthogonal persistence and the implementation of a thread based language such as Java.

6. Persistence and recoverability

Thus far we have demonstrated that Grasshopper provides ideal abstractions for the implementation of persistent Java. However, we have not described how containers are populated with data nor how persistence is achieved. In the remainder of this paper we shall briefly describe the persistence mechanisms of Grasshopper. The management of containers is the responsibility of *managers* which are user-level entities. The use of managers is motivated by the desire, as far as practicable, to leave all *policy* decisions out of the kernel. The kernel provides *mechanisms* which can be used by higher level software to implement required policies. This provides maximum flexibility and avoids the kernel making decisions which impact upon performance. For example, the memory management policy can have major effects on the performance of garbage collection.

Each container has an associated manager, which is an ordinary user-level program, held within a container. The manager is responsible for:

- the provision of the pages of data stored in the container,
- responding to access faults,
- operation within a limited amount of physical memory (page discard), and
- the implementation of a stability algorithm for the container [12], i.e. maintenance of the integrity and resilience of data.

The kernel provides a standard framework in which managers may operate. This includes automatic invocation of the appropriate manager on an access fault, and a set of interfaces which allow managers to arrange the hardware translation tables in such a way that the required

data is visible at an appropriate address in the container. Thus managers provide user-level virtual memory management in common with other operating systems [2, 7]

The Grasshopper kernel treats loci and the data accessed by them during computation as the unit of recovery. Loci are able to snapshot the state of their computation at any time, a task which is co-ordinated by the kernel and draws on services provided by the managers to snapshot user level data. A snapshot consists of all the data related to the computation of a locus and includes:

1. any modified container data seen by the locus,
2. any data maintained within the kernel to represent the state of the locus (including the registers) and the containers in which it has executed.

Since a locus can move between containers during the course of its computation, a snapshot typically involves recording the state of pages within a number of different containers. In contrast to other persistent systems in which a snapshot involves making the entire persistent store stable, the snapshot mechanism in Grasshopper only affects the stability of the portions of containers seen during the computation of a particular locus. Since loci are free to use shared memory as a means of inter-process communication, the actions of one locus can be influenced by the actions of another. This interaction creates causal dependencies between loci. During the normal operation of the system it is possible to ignore these causal dependencies because they are automatically preserved. However, if the system needs to be restarted after a shutdown or crash, locus snapshots must be used to rebuild a consistent system state.

It is therefore necessary to detect causal dependencies and ensure that they are preserved across failure of the system, thus guaranteeing global consistency. Detection of causal dependencies is performed by the kernel and managers which monitor read and write faults to compile modified page lists containing an entry for every modified page seen by a locus since its last snapshot. In addition, the kernel also maintains a list of containers in which a locus has seen modified data. The kernel uses this list to determine which managers it must request to snapshot data modified by the snapshotting locus.

The kernel co-ordinates the processing of locus snapshots and maintains dependency information such that it is possible to recover the state of the system from a causally consistent set of locus snapshots following a failure. Causal dependencies between loci are represented using vector time [13]. Each locus has an associated vector time which is lazily updated whenever a snapshot is performed . The vector time contains a list of pairs representing the state of each computation on which the snapshotting locus is dependent. Each pair contains the identity of a locus and a timestamp derived from a Lamport clock [15] associated with the locus which is used to identify points in time during its execution. This information is sufficient to characterise the causal dependencies of loci and their snapshots [12].

The above mechanism guarantees that a Grasshopper system will always recover data and processes to a self consistent state. It does not guarantee that the snapshotted state of the system was semantically consistent when a snapshot was made. Such guarantees require either co-operation or exclusion at the application level. Grasshopper supports these activities in two ways. Firstly, mechanisms that allow concurrent loci to co-operate are provided. These mechanisms include semaphores and conditional locks. Secondly, the Grasshopper kernel and managers co-operate to provide transactional semantics to those loci that require it.

7. Conclusion

We have shown how a persistent version of the Java language may be implemented on Grasshopper. Java is the fifth persistent language we have implemented on Grasshopper; the other being (in order of appearance): assembly language, C, Napier88 and C++. Each of these implementation efforts has proceeded in a relatively pain free manner demonstrating the power and suitability of the underlying Grasshopper abstractions. We hope to be able to demonstrate Java running on Grasshopper at the workshop.

Acknowledgements

We would like to thank Larry Sendlosky (larry@amt.tay1.dec.com) for his help in supplying us with a 64 bit clean version of Java and for his help in porting Java to Grasshopper.

References

1. "PS-algol Reference Manual - fourth edition", University of Glasgow and St Andrews, Technical Report Persistent Programming Research Report 12/88, 1988.

2. Acceta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M. "Mach: A New Kernel Foundation for Unix Development", *Proceedings, Summer Usenix Conference*, pp. 93-112, 1986.

3. Atkinson, M. P., Chisholm, K. J. and Cockshott, W. P. "PS-algol: An Algol with a Persistent Heap", *ACM SIGPLAN Notices*, vol 17, 7, pp. 24-31, 1981.

4. Atkinson, M. P., Jordan, M. J., Daynes, L. and Spense, S. "Design Issues for Persistent Java: a type safe, object oriented, orthogonally persistent system", *7th International Conference on Persistent Object Systems*, Cape May, New Jensey, Springer-Verlag, pp. to appear, 1996.

5. Birrell, A. D. "An Introduction to programming with threads", DES SRC, Palo Alto, Technical Report 35, 1989.

6. Brown, A. L. and Cockshott, W. P. "The CPOMS Persistent Object Management System", Universities of Glasgow and St Andrews, Technical Report PPRR-13, 1985.

7. Chorus-Systems "Overview of the CHORUS Distributed Operating Systems", *Computer Systems - The Journal of the Usenix Association*, Vol 1 No 4., 1990.

8. Connor, R., Brown, A., Carrick, R., Dearle, A. and Morrison, R. "The Persistent Abstract Machine", *Proceedings of the Third International Workshop on Persistent Object Systems*, Newcastle, Australia, Springer-Verlag, pp. 353-366, 1989.

9. Dearle, A. "Environments: A Flexible Binding Mechanism to Support System Evolution", *Proc. 22nd Hawaii International Conference on System Sciences*, Hawaii, vol II, pp. 46-55, 1989.

10. Dearle, A., di Bona, R., Farrow, J., Henskens, F., Hulse, D., Lindström, A., Norris, S., Rosenberg, J. and Vaughan, F. "Protection in the Grasshopper

Operating System", *Proceedings of the 6th International Workshop on Persistent Object Systems*, Tarascon, France, Springer-Verlag, pp. 60-78, 1994.

11. Dearle, A., di Bona, R., Farrow, J., Henskens, F., Lindström, A., Rosenberg, J. and Vaughan, F. "Grasshopper: An Orthogonally Persistent Operating System", *Computer Systems*, vol Summer, pp. 289-312, 1994.

12. Dearle, A. and Hulse, D. "On Page-based Optimistic Process Checkpointing", IW*OOOS '95*, Lund, Sweden, pp. 24-32, 1995.

13. Fidge, C. "Timestamps in Message-Passing Systems That Preserve Partial Ordering", *11th Australian Computer Science Conference*, University of Queensland, pp. 56-66, 1988.

14. Gosling, J., Joy, B. and Steele, G. "The Java Language Specification", Addison-Wesley, 1996.

15. Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System", *CACM*, vol 21, 7, pp. 558-565, 1978.

16. Lindstrom, A., Rosenberg, J. and Dearle, A. "The Grand Unified Theory of Address Spaces", *Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, USA, IEEE Press, pp. 66-71, 1995.

17. Sun Microsystems,."Chapter 2 Lightweight Processes", *Programming Utilities and Libraries*, 1990.

18. Sun Microsystems, "Java Release 1.0.2", http://java.sun.com/, 1996.

19. Morrison, R., Brown, A. L., Connor, R. C. H. and Dearle, A. "The Napier88 Reference Manual", University of St Andrews, Technical Report PPRR-77-89, 1989.

20. OMG "IDL Syntax and Semantics", *The Common Object Request Broker: Architecture and Specification*, pp. 45-80, 1991.

21. Richardson, J. E. and Carey, M. J. "Implementing Persistence in E", *Proceedings of the Third International Workshop on Persistent Object Systems*, Newcastle, Australia, Springer-Verlag, pp. 175-199, 1989.

22. Traiger, I. L. "Virtual Memory Management for Database Systems", *Operating Systems Review*, vol 16, 4, pp. 26-48, 1982.

23. Vaughan, F., Schunke, T., Koch, B., Dearle, A., Marlin, C. and Barter, C. "Casper: A Cached Architecture Supporting Persistence", *Computing Systems*, vol 5, 3, California, pp. 337-364, 1992.