A Distributed Stable Store

A.L.Brown
Department of Computational Science
University of St.Andrews
North Haugh
St.Andrews, KY16 9SS
Scotland.

Abstract

This paper describes how a distributed stable store may be constructed from several independent Napier systems. A Napier system is composed of three layers: A stable object store, an abstract machine operating within the object store and Napier programs running on the abstract machine. The aim of the distribution implementation is to allow independent Napier systems to operate as if there were only one stable object store. That is a Napier program need not be aware of any distribution. Therefore the implementation technique described is a communication mechanism between abstract machines.

The basis of the technique is to use pointers to special link objects to represent pointers to objects in remote stable stores. Hence whenever an abstract machine dereferences a pointer it must check to see if the pointer refers to a link object. If the pointer does, the abstract machine must then request the remote abstract machine identified by the link object to perform the dereference.

In addition to their addressing function link objects are also used to distribute garbage collection and stabilise operations. This is done by maintaining lists of link objects for all exported and imported pointers. The link objects are used to identify the remote stable stores a stabilise or garbage collection should be propagated to.

To complete the distribution mechanism a method of identifying object formats is provided. This allows for Napier systems implemented on VAXs, SUNs, PERQs and ICL3900s to be combined into a distributed store even though they have different representations for data such as integers, floating point numbers and executable code.

1. Introduction

Modern computer systems are becoming ever smaller and more powerful. So much so that mainframe computers are being replaced by numbers of small powerful personal workstations. However if personal workstations are to be used for any group activity there is a requirement for the workstations to

behave like a single shared resource. Providing such a shared resource and distributing it raises several issues that must be addressed.

1. what resources of each workstation are to be shared ?
2. how are the various workstations to be connected?
3. how do the various workstations address each other ?
4. how are the potentially radical differences in workstation architectures to be overcome?

In most networks of personal workstations distribution of resources is achieved by adding networking support to an existing architecture whereas relatively few architectures actually integrate distribution at the design stage. This paper describes an architecture that is designed to support distribution. It also describes a particular distribution mechanism that uses instances of the architecture to construct a distributed stable store. Each instance of the architecture is an independent Napier system[1,2,3].

2. The Architecture of a Napier System

The architecture of a Napier system is designed to support experiments in concurrency, transactions and distribution in a persistent environment. As such a Napier system is composed of several layers corresponding to the logical levels of architecture it requires. These architecture levels provide virtual memory, stable storage, persistent objects, typed operations on objects, transaction mechanisms, sharing mechanisms and distribution mechanisms. The intention of the layering is to clearly distinguish the logical architecture levels required and allow experimental implementations of individual layers to be easily substituted.

A key factor in the design of any architecture should be the type of programming languages that it will support. If weakly typed programming languages such as C or assembler are supported then the architecture must incorporate elaborate mechanisms to constrain potentially erroneous programs. For example, in many computer systems every program is given its own virtual address space in which to run. This leads to further complexity when two programs wish to share data. In object oriented computer architectures such as capability machines protection is provided by

giving each object its own address space and a set of capabilities with which it may be accessed. These machines have their own special problems in protecting capabilities from erroneous programs. Protecting capabilities usually results in significant performance overheads particularly when context switches are required[4,5].

In order to reduce complexity the layered architecture only supports strongly typed programming languages. It also requires these programming languages to be compiled into the Persistent Architecture Intermediate Language, PAIL[6]. This ensures that all the operations a program may attempt to perform are legal. The use of PAIL simplifies the layered architecture by eliminating many of the complex protection mechanisms required. For example the PAIL type system can provide many of the protection mechanisms afforded by a capability system avoiding the need to give each program or individual object a separate address space.

A major benefit of this layered architecture is the ability to separate the use of an object from the way it is stored. Those layers of architecture related to the use of objects are constructed using the PAIL layer whereas those related to the storage of objects are constructed to support the PAIL layer.

In the context of strongly typed programming languages sharing, transactions and distribution are all ways in which objects may be used. As such, they are provided as layers constructed using the PAIL layer. This has several important consequences for the architecture layers that support the PAIL layer in that they need not directly support sharing, transactions or distribution. In fact the lower layers need only provide a single unshared store of persistent objects.

The persistence of an object is independent of how the object is used and all objects may potentially persist forever[7]. Hence the persistence of any particular object cannot be deduced from or impact on its use. For this reason a layer to support persistent objects is provided below the PAIL layer. The PAIL layer of the architecture guarantees that the persistent objects will never be misused. Therefore the persistent objects layer may be constructed by implementing a simple heap on top of a stable store.

The lowest layers in the architecture provide a stable store. The stable store is the only storage mechanism provided and is therefore implemented using the entire physical store. Access to the physical store is made via a virtual memory mechanism. Since there is only one virtual address space, the virtual address space need only be big enough to address physical store keeping virtual addresses short. This use of short virtual addresses and the elimination of context switching allows the virtual memory mechanism to be very simple and efficient.

3. Support Required for a Distribution Architecture Layer

In order to provide distribution as an explicit layer of the architecture it is necessary to separate the support required by such a layer from the layer itself. The ideal environment in which to experiment with distribution is a homogeneous network. Therefore the ideal support for a distribution mechanism would be the emulation of a single architecture. However a strict emulation could lead to severe performance degradation even on physical architectures that are similar to the one being emulated. To minimise the performance penalties the layered architecture is defined to allow for minor variations in physical architecture. This is done in the following ways.

3.1. Object Formats

PAIL operates on many different kinds of objects such as vectors of integers, raster images, strings, structures, abstract data types, procedure frames and executable code. Although these objects have very different semantics their differences are purely in how they are interpreted by PAIL. In the architecture layers supporting PAIL there is only a single kind of persistent object. This persistent object is made up of three components, a header, a set of pointers to other persistent objects and a set of non pointers. For example, a vector of two integers:

Two Word Header	Vector's Type	First Integer	Second Integer	Lower Bound	Upper Bound

The Persistent Object Representing a Vector of Two Integers

Number of Pointers 1	Size of the Vector 7 words

The Two Word Header for the Vector of Integers

An object's header consists of two words that are sufficient to describe the object for use by a garbage collector. A word is a 32 bit integer. The first word contains the number of pointers the object holds in its least significant 24 bits, the remaining 8 bits being reserved for use as tag bits. The second word contains the size of the object in words. Each pointer in a persistent object is a word thus limiting an instance of the architecture to 4 giga persistent objects. The non pointers are laid out in the object's remaining words in any way the PAIL sees fit. By convention PAIL ensures that the first pointer in a persistent object always points to a PAIL type description of the object. This type is used for dynamic type checking in Napier and as a means of implementing a symbolic debugging tool.

The object format was chosen for two reasons. First the interpretation of a persistent object is not relevant to how it is stored. Therefore the lower layers of the architecture may be completely independent of the layers above PAIL. This allows experimental storage and garbage collection technologies to be used without the need to redesign the higher layers. The second reason for the chosen format is that integers can be moved between similar architectures without changing their interpretation. For example, a VAX and a SUN can represent the same 32 bit integers even though the bytes representing integers have a different physical ordering[8,9]. Hence the two word header of an object would have the same interpretation on a VAX as on a SUN.

3.2. Word and byte ordering

A 32 bit integer is made up of two short words, each two bytes long. Depending on the physical architecture used the bytes and short words can be arranged in four ways to represent an integer. For most kinds of objects this ordering is irrelevant since the interpretation of their non pointers is defined in terms of integer values. However strings, raster images and executable

code are partly defined in terms of a specific byte ordering. This is because the bytes holding the characters of a string, the pixels of a raster image or a piece of code may represent different integers on different physical architectures. For example, if a string is copied from a SUN to a VAX each group of four bytes in the string will be reversed. To overcome this ordering problem every object uses two of its tag bits to indicate the ordering of short words within integers and bytes within short words. This enables a string copied between different physical architectures to be correctly interpreted by inspecting its tag bits and performing the necessary reordering of its characters.

3.3. Floating Point Emulation

Integers are relatively simple data types that are easily mapped between physical architectures. However the same does not hold true for floating point numbers. Different physical architectures can have radically different floating point implementations, indeed an individual architecture may use more than one implementation. As a compromise solution the emulated architecture's floating point numbers conform to the IEEE standard 754[10]. This solution allows floating point numbers to be defined by bit significance in two integers. The standard is also easily mapped, except for extreme cases such as the infinities, to the floating point representations of the ICL 3900 and the VAX G-format reals[8].

3.4. Raster Graphics

Implementations of raster graphics vary in at least as many ways as floating point implementations. For this reason a common representation of raster images is included as part of the architecture. The chosen representation is composed of two parts, a persistent object containing the raster image's pixels and an object that describes a three dimensional section of the image. Previous experience gained in implementing the PS-algol raster images[11] on different machines suggests that efficiency will not be unduly impaired.

3.5. Executable Code

Executable code is the one part of a physical architecture that is guaranteed to be unique. It is usually directly supported by hardware and finely tuned to its host architecture. For these reasons it is not feasible to force all physical architectures to emulate a particular physical architecture's code. This

problem is overcome in two ways. The first solution is to use the PAIL definition associated with each executable code object to generate code for the host physical architecture. The alternative solution is to use the Napier abstract machine that will support PAIL[2]. This abstract machine is defined entirely in terms of bytes. Every instruction's operation code is a single byte and the interpretation of an instruction's parameters is strictly defined. Therefore abstract machine code objects can be easily swapped between different physical architectures and work efficiently.

4. A Possible Distribution Mechanism

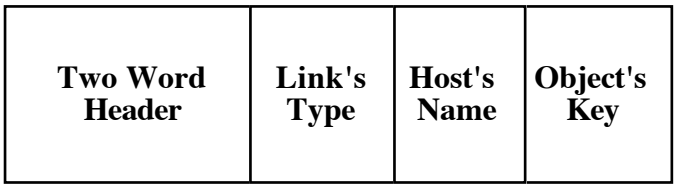
The distribution support described above is suitable for a wide range of physical architectures that use a 32 bit integer such as the SUN, VAX, ICL PERQ and ICL 3900. The problem of defining operations on remote objects is avoided since all objects can have the same interpretation whether they are local or remote. Therefore any distribution layer that may be constructed within the layered architecture need only concern itself with how objects are to be addressed and whether or not the distribution should be made explicit.

To demonstrate this, a simple distribution layer will now be described that only provides a distributed stable store. Facilities to support distributed execution of applications are excluded for simplicity. The model of distribution is transparent to all application programs. However a single interface is provided to PAIL that allows access to a new network host's root of persistence. This interface is sufficient to bootstrap the distribution mechanism, thereafter the distribution is also transparent to PAIL. For this reason the distribution layer is inserted between PAIL and the architecture layers that support PAIL. Since the distribution mechanism is transparent to PAIL it must provide a means of propagating garbage collection and store stabilisation operations. To avoid issues of reliability and simplify the task of distributed garbage collection it is assumed that the distribution layer will only be used on a small tightly coupled network.

4.1. Addressing and link objects

The distribution layer's addressing mechanism is based on the use of special link objects to indicate pointers to remote objects. An addressing mechanism that uses extended pointers is not permitted since this would require other layers in the architecture to be modified. These link objects contain details of

how the remote object is to be addressed, the name of its host machine and a key value identifying the object on its host.



A Link Object For a Remote Object

Link objects are created when a pointer to a host's object is passed to another host. On the local host an entry is made in an exports list and a unique key value is allocated to the object. The exports list is a record of all local pointers that have been exported to another host together with their associated key value. Key values are used so that the distribution layer does not affect the way the architecture layers implementing persistent objects may use or reuse pointer values. Once the export list entry is made the object's key value and the local host's name are passed to the remote host. The remote host then creates a link object to hold the key value and name and then enters the link object in an imports list. An imports list is a record of all pointers to remote objects. The remote host then uses a pointer to the new link object to represent the pointer to the object.

The communication of a pointer is a little less complex if the local pointer points to a link object. In this case the local host just sends the key value and host name recorded in the link object. The remote host can then construct its imports list as before. However it is possible that remote host is the host holding the object. In that case the remote host's exports list is consulted and the actual pointer to the object retrieved.

4.2. Operations on Remote Objects

When an operation is to be performed on an object a check is first made of the tag bits. All link objects use a tag bit to identify themselves. If the object is a link object then the operation must be performed by the remote object's host. One advantage of using PAIL is that all operations that are performed know what type of objects they should be using. Therefore this type information can be used by the distribution layer to ensure the appropriate

interpretation of values that are copied between hosts. In particular it is able to use the tag bits that identify byte and short word ordering to reorder the appropriate parts of objects such as strings.

4.3. Propagating Stability

Each instance of the layered architecture provides a mechanism for stabilising its own persistent store. The stabilisation saves a consistent state of the persistent store that can be recovered after a system crash. Therefore the distribution mechanism must provide a similar operation that saves the entire network in a single consistent state. This is done in the following way.

When a store stabilisation operation is performed the imports list is consulted. Every host mentioned in the list is requested to perform a stabilise operation. These hosts will in turn consult their imports lists and further propagate the stabilise. A store stabilisation operation does not terminate until the local stabilise has been performed and all the remote stabilise operations have been completed. A remote stabilise will not terminate until it has received replies from any stabilise operations it may have requested. This two phase stabilise operation ensures that all hosts in a network stabilise at the same time. If a failure occurs in any host then all the hosts in the network can be restored to a single consistent state. This type of state restoration is only practical in a small tightly coupled network. Hence the initial assumption about the network type.

4.4. Distributed Garbage Collection

Garbage collection is performed at two levels. The first level is local to a particular host. In this case all objects reachable from the local root of persistence, the imports list and the exports list are preserved. The second level of garbage collection is performed across all the hosts in the network. It requires each host to perform a local garbage collection marking only objects reachable from the local root of persistence. Any entries in the imports list that are kept cause the host holding the remote object to mark the object and all those objects reachable from it. These local garbage collections continue until all reachable objects in the network have been marked. All unmarked and therefore unreachable objects in the network and their associated entries in exports and imports lists are discarded. This garbage collection mechanism relies on the distributed stabilise operation to ensure that all hosts in the

network are in a consistent state. If an individual host could revert to a previous stable state independently of the rest of the network it could restore objects that point to deleted objects. Such a possibility would make distributed garbage collection infeasible.

5. Conclusion

A layered architecture has been described that is suitable for conducting experiments in distribution. It has a flexible architecture definition that can be efficiently emulated by similar physical architectures. This allows the layered architecture to separate the issues of combining different physical architectures from the issues of modeling distribution. Therefore the layered architecture is a flexible and easy to use tool for experimenting with distribution.

References:

1. Morrison, R., Brown, A.L., Carrick, R., Connor, R.C. & Dearle, A.
   The Napier Language Reference Manual, In preparation.
2. Morrison, R., Brown, A.L., Carrick, R., Connor, R.C. & Dearle, A.
   The Napier Abstract Machine, In preparation.
3. Brown, A.L.
   The PINT POT, Persistent Information Space Architecture Persistent Object Store, In preparation.
4. Needham, R.M. & Walker, R.D.H.
   Protection and Process Management in the "CAP" Computer, International Workshop on Protection in Operating Systems, IRIA, Rocquencourt, August, 1974.
5. Wulf, W.A., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C. & Pollack, F.
   HYDRA: The Kernel of a Multiprocessor Operating System, CACM 17(6), 337-345, June, 1974.
6. Dearle, A.
   A Persistent Architecture Intermediate Language, The Universities of Glasgow and St.Andrews PPRR-35 (1987).
7. Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
   An Approach to Persistent Programming. Computer Journal 26, 4 (November 1983), 360-365.

8. VAX Architecture Handbook, Digital Equipment Corporation , 1981.
9. Motorola,
MC68020 32-bit Microprocessor User's Manual, 2nd edition, Prentice Hall, 1985.
10. A Proposed Standard for Binary Floating Point Arithmetic, Computer, March 1981, 51-62.
11. Brown, A.L. & Dearle, A.
Implementation Issues in Persistent Graphics, University Computing, Vol. 8, No. 2, (Summer 1986), 101-108.