Operating System Support for Inter-Domain Type Checking

Alex Farkas, Alan Dearle and David Hulse

Department of Computing Science
University of Stirling
Stirling, FK9 4LA
Scotland

{alex,al,dave}@cs.stir.ac.uk

Abstract

Most existing file based operating systems tend to provide very little in the way of type related information about applications to the user. Instead, the user is required to construct applications that perform their own type checking, relying on information about existing programs and data to be obtained via other means, such as manual pages, or by visual inspection of source code. This lack of public type related information has, by and large, prevented the benefits of browsing technology as found in some persistent language systems to be delivered to the operating system level. A technique is examined in which detailed type information about operating system entities may be recorded, thus enabling more informative user interfaces, tools and applications to be constructed.

1. Introduction

The development of persistent systems has sparked the emergence of a variety of new programming techniques and tools. In particular, persistent store browsers [4,7,8,14] have been developed which permit users to discover the types of values contained in a persistent store, and the structure of values with respect to each other. In systems such as the Aberdeen programming environment [8] for Napier88 [17], the browser enables a user to quickly locate and execute programs in a persistent store without prior knowledge of the location of the data, programs or their interfaces. This is achieved by displaying graphical representations of the discovered values and their types. Using this information the programmer may manipulate data, invoke discovered programs, or construct new programs that use the browsed values in a type correct manner.

Until now, tools such as persistent store browsers have largely been language specific and the benefits of these tools have only been available from within the language environments that supported them. From the perspective of general purpose operating systems, corresponding tools have not been forthcoming due to the lack of type information about programs and data at the operating system level. Some operating systems do provide some of this functionality, for example, a tool called AppleScript [11] has been developed for the Macintosh system which enables applications to be invoked via textual scripts constructed by a programmer. Applications may provide a special interface which enables AppleScript to call functions within the application, such as the print option of a word processor. AppleScript even provides a simple browser which permits an application's AppleScript interface to be interrogated providing the programmer with a list of the permissible operations and their parameters.

However, the AppleScript language is not a general purpose programming language, and its specialised nature prevents it from being used to build complex applications. Furthermore, the interface information is not (readily) available to other applications and must be built separately into each application.

As most existing file based operating systems tend to provide very little in the way of type related information about applications, the user is required to construct applications that perform their own type checking. Information about existing programs and data must be obtained via other means, such as manual pages, or by inspecting source code. This has, by and large, prevented the benefits of browsing technology as found in systems such as Napier88, O_2 or Smalltalk from being delivered to the operating system level. This paper examines a user level architecture that permits detailed type information about operating system entities to be recorded, and later browsed and/or retrieved by the user. This architecture permits more sophisticated browsing and program development tools to be constructed. The operating system used in implementing the architecture described in this paper is the persistent operating system Grasshopper [6]. Section 2 gives an outline of this operating system and introduces the abstractions pertinent to the rest of the paper, as well as the IDL model used to represent types in the architecture. Section 3 describes the architecture, Section 4 examines the *G-shell*, a tool which is built using the architecture, Section 5 discusses some related issues and future work, and Section 6 concludes.

2. Grasshopper

The Grasshopper operating system has been expressly designed to provide support for orthogonal persistence. All entities in Grasshopper are constructed using three basic abstractions: an access and protection abstraction, the *capability* via which all Grasshopper entities are manipulated, an abstraction over all storage known as the *container* and an abstraction over execution known as the *locus*.

In order to operate on any Grasshopper entity, an appropriate capability must be presented. A capability in Grasshopper may be thought of as a pointer to an entity and a set of rights associated with that entity. Grasshopper implements a *segregated capability scheme* therefore all fields are protected from user access and are managed by the Grasshopper kernel [5]. For this reason, loci must refer to capabilities indirectly using a capability reference known as a *CapRef*. Capabilities in Grasshopper contain a number of different fields related to protection, however the conceptual view shown in Figure 1 is sufficient for the discussion in this paper. The system fields define various access rights and identify the entity to which the capability grants access. The user field is uninterpreted by the system and may be used for a variety of purposes; for example it could be used to identify a user or might be used as a tag mechanism in the implementation of a type system.

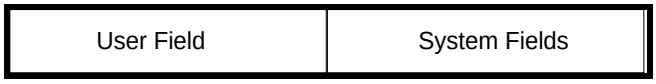

Figure 1: The logical structure of a capability in Grasshopper

Containers are the only storage abstraction provided by Grasshopper and replace the notions of address spaces and file storage found in conventional file based systems. A container is a protected address space which may be used to store both programs and/or data. Containers are used in a variety of ways: as code repositories (like executable files or libraries in Unix), as data repositories (like data files in Unix) or as protected abstract data types containing both code and data and exporting a set of operations that operate on the encapsulated data.

In most operating systems, the notion of an address space is associated with an ephemeral entity, a process, which is the only entity that may access data within that address space. By contrast, Grasshopper containers and loci are orthogonal concepts: the longevity of containers and loci are independent of each other, containers may or may not have (multiple) loci executing within them and loci may migrate between different containers. A locus always executes in the context of some container, its *host container*, and when an address is generated by a locus it is always interpreted relative to its host container.

Grasshopper provides two facilities that allow the transfer of data between containers: *mapping* and *invocation*. Container mapping allows data in a region of one container to be viewed within a region of another container. Unlike the memory mapping mechanisms provided by other systems [1,2,22] containers may be arbitrarily (possibly recursively) composed providing considerably enhanced flexibility and performance [15].

Invocation is the process whereby a locus moves between containers. During invocation, a locus may supply a parameter block which contains data to be used in the destination container. There is no distinguished format for the data in the parameter block and it may be interpreted arbitrarily by the destination code. Typically a collection of data structures are marshalled into the parameter block and unmarshalled in the destination container.

All containers in Grasshopper possess a single entry point known as an *invocation point*. When a locus invokes a container, it begins executing code at this point. Since it is the invoked container that provides the code to be executed, it controls the execution of the invoking locus. Combined with the capability protection mechanism this allows hardware protected ADTs to be constructed [15].

2.1 Containers as ADTs

In Grasshopper, containers are commonly used to implement abstract data types exporting an arbitrary set of operations that operate on an encapsulated state; this is illustrated in Figure 2.

Client applications may access the container via the set of exported operations. However, as described earlier, a basic Grasshopper container only supports a single entry point at which execution commences if the container is invoked by a locus. This means that whenever a locus enters a container it always enters at the same location; whilst this enables the low level implementation to be lightweight and secure, it does not provide a direct method for enabling a collection of operations to be exported by a container.

Consequently, the following convention has been established to enable an invoking locus to specify more precisely a wider variety of container access points. Each invoking locus may carry with it a value called an *operation number*. Upon a locus's arrival in the destination container, the code at the invocation point examines the locus's requested operation number

and branches execution to the appropriate section of code within the container. Thus, via an arbitrary set of operation numbers, a container may effectively export any number of operations. This is illustrated in Figure 3.

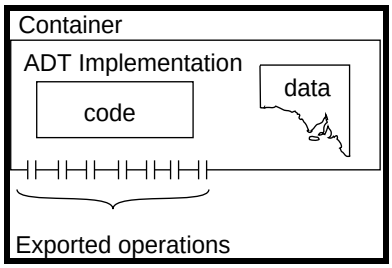

Figure 2: Conceptual view of an ADT implemented by a container.

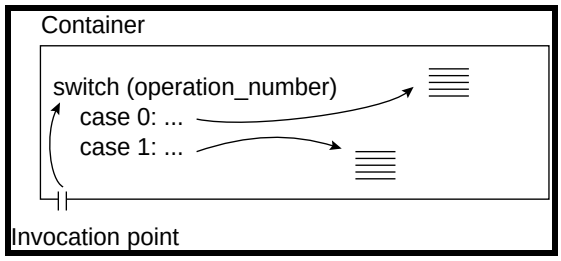

Figure 3: Exporting multiple operations via the operation number convention.

The main difficulty of programming with the basic Grasshopper abstractions as described above is that considerable effort is required on the part of the programmer in implementing containers and clients; typically, a client program must marshall the relevant data to be sent to a destination container, establish the correct operation number value, and apply a system level operation to cause the locus to invoke the destination container. Similarly, the code in a destination container must inspect the incoming locus's operation number, unmarshall the parameters, and branch to the relevant piece of code.

Such code may be generated automatically by a tool, provided that a specification of the container interface is available. In our system, the interface definition language, IDL [19] is used to specify container interfaces in a machine and language independent manner. IDL is the interface definition language specified in the CORBA standard [18], and may be used to generate the following components for any given container interface definition:

- code generated for use within a destination container,
- client code which may be used to invoke a container, and
- a structured representation of the container interface.

The first of these assists in the construction of a container supporting a specified interface. The generated interface code contains all the code necessary to unmarshall the parameters of an invoking locus and to transfer execution to the requested operation number. In addition, stub procedures for the various operation numbers are generated, the bodies of these procedures are the only parts of the program which need to be implemented by the programmer.

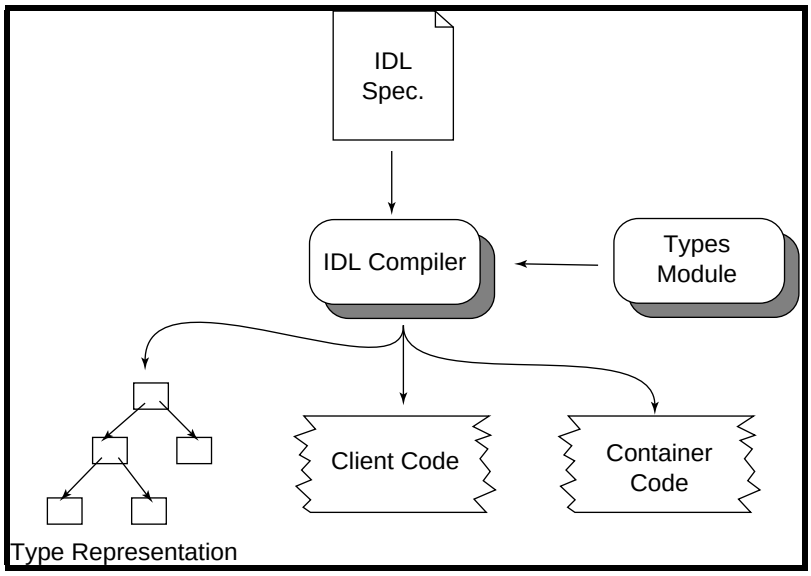

Figure 4: Generating components via the IDL compiler.

The second segment of code is intended for use by clients that wish to invoke any container with the given interface. This code may be mapped into a client's address space, allowing the programmer to invoke a container via a single function call, and removing the need for a programmer to implement the complex marshalling code otherwise required. The generated code performs all of the necessary marshalling of data and invoking, as well as the unmarshalling of any data returned.

Finally, a structured representation of an interface is generated which enables tools such as browsers to present information about an interface to the programmer. An independent library of operations for querying and constructing these representations is provided as part of the system, and they are analogous to those described for the Napier88 system in [3]. They are used by the IDL compiler, and may be accessed by any application to examine or manipulate the representations; for brevity, these operations are not described further here. The IDL compilation process is summarised in Figure 4.

The IDL specification technique described above provides an abstraction over the otherwise complex sequence of activities required to construct and use Grasshopper applications. Furthermore, the output from the compiler is made available in the Grasshopper system for use by browsers and other tools.

2.2 Containers as Libraries

In the section above, containers were described as providing a set of operations by which client applications could operate on encapsulated data. However, containers may also serve as data or executable code repositories, the contents of which are accessed via mapping rather than invocation.

As described earlier, a region of one container may be mapped into the address space of another, allowing, for example, a locus to execute code resident in another container. This does not require invoking the container which contains the code or data to be mapped, however the programmer must have knowledge about the internal composition of the container in order to know which regions to map. A somewhat crude solution is simply to document and make public the internal structure of a library, however, there are a number of drawbacks to this approach:

- it requires additional effort on the part of the library constructor,
- the method is open to error, especially if libraries are updated and documentation is not, and
- tools such as browsers or linkers are unable to determine the contents of libraries, making the implementation of such tools difficult.

Like the ADT style containers described above, libraries export a set of operations which may be described using IDL. The difference between these approaches is that generally containers providing an ADT interface are invoked, usually for protection reasons, whereas libraries are invoked by browsers or linkers and their contents mapped. Each library consists of a container containing the executable code, some linkage information describing whence the code may be mapped and an IDL specification of the exported operations.

3. Name and Type Control

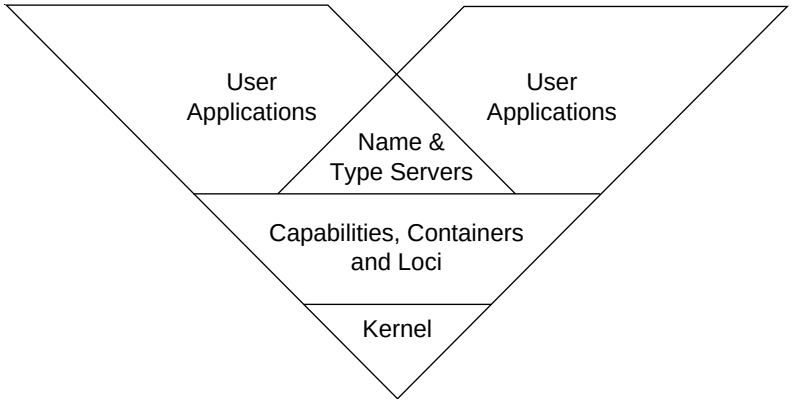

Figure 5: Name and type servers in the Grasshopper architecture.

Two abstractions are provided by Grasshopper to control names and types, namely *name servers* and *type servers*. The first abstraction is called a *name server*, and provides a mapping between textual names and capabilities. Name servers are similar in purpose to directories or folders in file based systems, but have the additional functionality that any number of labelled

attributes may be associated with each name server entry. These attributes may be used to store arbitrary information about an entry in the form of integers, booleans, strings and even capabilities for other entities.

The second abstraction is known as a *type server*, and provides a repository for general type related information and executable code. Specifically, type servers store type representations of IDL interface definitions and code which may be used by a client to invoke containers with a given interface. Both name and type servers are intended for use by higher level applications, however they themselves are constructed using Grasshopper containers and capabilities and therefore reside in the domain of Grasshopper applications. This refined architecture is illustrated in Figure 5, which also indicates that in general, applications are not obliged to employ the name or type services.

3.1 Using Name and Type Servers

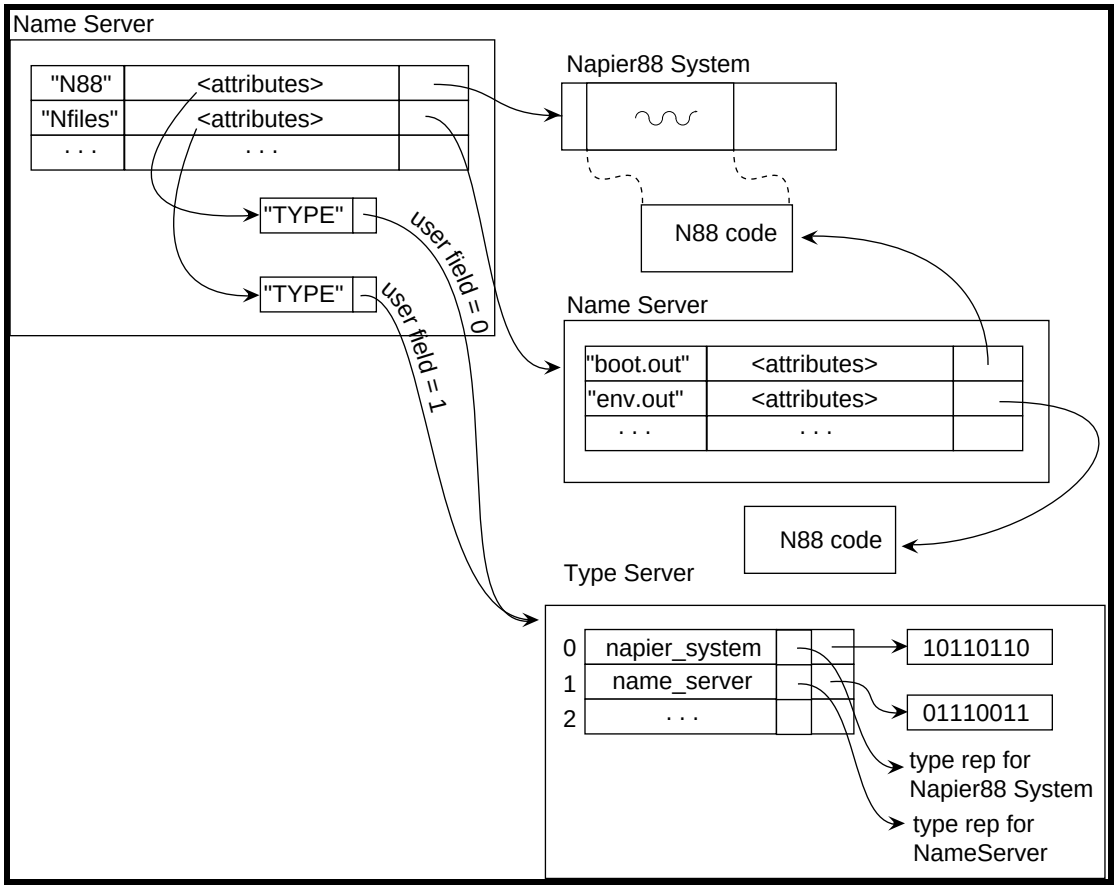

Figure 6: Name and type servers.

In general, name servers and type servers may be treated as independent entities. However, as shown in Figure 6, in Grasshopper they are used in conjunction with each other and permit applications to:

- obtain references to existing entities,
- determine the types of those entities, and
- to access executable client code required to call the entities.

The name server in the top left corner of Figure 6 contains entries for capabilities to two containers: a Napier88 programming system and another name server. These are associated with the names *N88* and *N88files* within the name server. In addition, each entry has been given an associated attribute, *TYPE*, which contains a reference to a type server that stores interface related data and client code appropriate to each container. In this case, each *TYPE* attribute is a capability for the type server shown on the bottom right of Figure 6. Encoded within the user field of each capability is a tag which may be used by the type server to locate the appropriate type representation.

To illustrate, consider the example where a client wishes to run a program using the Napier88 system. Figure 7 shows the IDL specification for the Napier88 container; a simplified representation of this interface type along with the client code are stored in the type server with the tag 0.

```
interface napier_system
{
        Status run_program( in CapRef program );
        Status initialise_store();
};
```

Figure 7: Specification of a Napier88 system interface.

In order to obtain a capability to the Napier88 system, the client performs a lookup operation on the name server with the name *N88*. This operation returns a capability for the Napier88 container with which it may be invoked, but does not provide the client with information about the operations exported by the Napier88 container. To determine the container's interface, the client retrieves the *TYPE* attribute associated with the entry *N88* in the name server. A capability for the type server is returned containing the tag for the *napier_system* entry. With this capability, the client may extract the type representation of the Napier88 system. Finally the client may request that the client code associated with the user field tag be mapped into the client's container, enabling the client to invoke the desired operations of the Napier88 system. As discussed below, this process may be performed at a variety of times.

3.2 Accessing Client Code

The example above has indicated broadly how name and type servers may be used co-operatively. The task of accessing the client code used to invoke a container may generally occur at three different times [9,13]:

1. Firstly, it is possible for the programmer to interrogate a name and type server during client construction. The interfaces to the various required containers may be discovered, and code constructed to call the correct interface procedures. The client code may be copied from a type server into the client's address space. This process is analogous to static linking in Unix.
2. The second possibility is similar to the first in that the programmer may interrogate a name and type server during client construction, and construct code to call the correct interface procedures. The client code is mapped into the host application. This may be performed statically when the application is being constructed or dynamically when the program is executed.
3. The final method is the topic of example in the next section; via this method, the client may be constructed in such a way as to query name and type servers dynamically, and to dynamically construct a call to the appropriate client code.

The first two techniques have counterparts in most other operating system environments and are not discussed further. The third technique, however, is not generally supported at the operating system level, largely due to the fact that insufficient type information is available, thus inhibiting any form of dynamic type checking. Systems such as Multics [20] enable a style of dynamic linking, however no type checking takes place unless programmed explicitly by the programmer. The next section illustrates an application which uses this third technique to provide a dynamically type checked, general purpose user shell for Grasshopper.

4. A Typed Shell for Grasshopper

An example application of the architecture described above is a general purpose command line oriented shell for Grasshopper called G-shell. The G-shell was constructed as a simple user interface to many of the features of Grasshopper, but serves to illustrate how name and type servers may be used. More sophisticated, graphics oriented tools such as those provided by the Napier88 system could be constructed using the same techniques, but this is the topic of future work and is not discussed further.

Associated with each instance of the G-shell is a default name server which, like a home directory, provides a starting point from which other entities may be discovered. The G-shell itself supports a number of built in commands for navigating and manipulating a name server hierarchy, but these are not pertinent to the discussion and are ignored in this paper.

There are two major differences between G-shell and the shells provided by Unix, Macintosh or DOS:

1. G-shell is typed,
2. G-shell passes different types of values to applications rather than just strings.

When a Unix shell invokes an application it always passes strings to the application being invoked. Conventionally two parameters are passed to a Unix application *argc*: a count of the number of arguments to the application and *argv*: an array of strings containing the arguments. As a consequence, every application must check that the number of parameters being passed is correct and must decode the strings and reconstruct the appropriate values. During the last operation type checking is performed implicitly, for example the string "z23" cannot be decoded into a decimal integer.

In systems such as Unix, applications often present more than one interface, for example, they may present one interface to the shell and another to the network. Typically the code used to perform interprocess communication with the application will be generated from a specification written in RPCL [16] or some other specification language and will be type checked by programming language compilers. Furthermore, the values that are passed across the network to the application are generally more complex than strings.

In the approach described in this paper we are advocating the use of a single, type checked mechanism for specifying all interfaces to applications. Once such a mechanism has been established, it is natural to provide type checking in the user shell. In addition, it is natural for the shell to supply the application with values drawn from a richer domain than strings. By allowing type checking to be performed dynamically by the G-shell, the intellectual burden on the programmer to perform type checking in each constructed application is reduced.

4.1 Type Checking

We will continue to use the Napier88 example introduced in Section 3.1 to illustrate the techniques. Suppose that a user of G-shell wishes to run the (interpreted) Napier88 program called *boot.out* contained in the name server called *Nfiles*. When the following command is entered at the G-shell prompt, the G-shell performs the actions described below:

```
N88.run 'Nfiles/boot.out
```

The first part of the line describes a command to be executed. In this case the user has specified that the *run* operation contained in the container denoted by *N88* is to be executed. Since the name *N88* does not refer to an intrinsic G-shell command, the name server associated with the shell is inspected to see if an entry with the name *N88* exists. Since it does, the type of the *run* operation must be ascertained in order to type check the invocation. This is achieved by querying the name server to obtain the capability associated with the *TYPE* attribute of the *N88* entry. The type representation for the *N88* interface may be obtained from the type server using this capability. The type of the *run* operation (if any) may be extracted from this type representation using library functions and in this case will yield the following type representation:

```
Status run_program( in CapRef program )
```

From this type representation, the expected types of the operation's parameters may be extracted and used to type check the command line arguments. However, before doing so, the actual parameter list is parsed and a list of values and types is constructed. In this example, a single command line argument "'Nfiles/boot.out" has been supplied. The single quote symbol informs the shell to treat the next word (*Nfiles*) as a name and look it up in the current name server. When the shell parses the remainder of this argument and finds the slash symbol it will assume that the name *Nfiles* denotes a name server which it will verify from its *TYPE* attribute. Finally, the name *boot.out* is looked up in the name server denoted by *Nfiles* to yield a *CapRef*. Once the actual parameter list is constructed it may be compared to the expected type of the parameter list found from the IDL specification. In this case the comparison is trivial and the types would be found to match.

4.2 Dynamic Mapping of Invocation Code

As highlighted in the previous section, the code necessary to invoke a container could be constructed statically, and the client code mapped into the client container statically or dynamically and executed. The G-shell follows this course with name servers and type servers. In both cases, the shell knows in advance of the existence of these types and how values of these types are to be used.

However, the G-shell has no knowledge of types such as *napier_system* and must therefore perform all type checking dynamically as described above. Furthermore they do not and cannot contain the code necessary to invoke instances of types such as *napier_system*. For this reason, the final approach described in the previous section is used, in that once the parameters are type checked, the client code is mapped dynamically, the parameters placed onto the run-time call stack, and the mapped code executed.

5. Discussion and Future Work

5.1 CORBA

The architecture described in this paper conforms to many aspects of CORBA [18]. The most obvious of these is the use of IDL as a specification language. Despite the fact that CORBA uses the active object model and Grasshopper the object-thread model, the way in which communication is effected is similar to CORBA. In CORBA all communication between address spaces is via an Object Request Broker. In Grasshopper this functionality is provided by the Grasshopper kernel. In both systems, the request to communicate with (CORBA) or invoke an object (Grasshopper) is independent of the object's location. This location transparency is also one of the corner-stones of persistent programming. The containers provided by Grasshopper are more powerful than CORBA objects in that, when viewed by a programmer, containers are persistent address spaces.

The name and type servers in our architecture resemble the functionality of instance and interface repositories in CORBA, however, the usage of these entities in our system differs from the CORBA repositories. For example, in our system, name and type servers may be used purely for browsing by a programmer constructing an application. Data or code may be extracted or linked to the final application, freeing it from any further dependence on the servers. This is not supported by the CORBA architecture.

In other ways, our architecture is a subset of the generic CORBA architecture. For example, only a subset of the IDL language (the interface definition part) is required to specify entities in the system. This further allows many of the kinds of tools described in the introduction to be constructed for general purpose use at the operating system level.

CORBA is designed to assist in the construction of distributed, object oriented application systems. Grasshopper benefits from this heritage, for example, a modified version of the Grasshopper IDL compiler may be used on a Unix system to generate a Unix based Grasshopper container interface. This permits Grasshopper services to be directly invoked from Unix application programs

5.2 OMOS

In the architecture described in this paper, name servers provide a mapping between names and implementations (containers), whilst type servers provide a mapping between interface specifications and code to access implementations exporting those interfaces. This system could be extended to enable a user to access multiple implementations of a particular interface depending on a specified set of requirements, as demonstrated by the OMOS system [21]. For example, in Unix, the *malloc* operation dynamically allocates blocks of memory. All applications requiring dynamic memory allocation, regardless of behaviour, use the same implementation of *malloc*, potentially retarding performance by preventing additional optimisation. Using an OMOS style linker, it is possible to request different implementations of *malloc* depending on the required behaviour. For example, one implementation could be optimised for memory intensive applications, and another for low memory usage applications.

In OMOS, a client specifies an interface and a number of desired properties for the implementation, and the linker returns an appropriate implementation. With judicious use of attributes in name and type servers, a similar style of programming could be obtained in our system. For example, multiple libraries could be implemented to support memory management routines optimised for a variety of circumstances. Each implementation could be tagged with a set of attributes denoting information such as garbage collection style, page alignment, whether free lists are used, and so on.

5.3 Hyper-Programming and Browsers

Hyper-programming has been described in various places [9,12,13], and is currently implemented in several Napier88 systems. The essence of hyper-programming is to allow a user to indicate that a program or value in the persistent store should be included directly within the source code of a program; this process is analogous to the way in which entities are linked in systems such as HyperCard [10] or Flex [23].

Using the architecture described in this paper, it is possible to construct language independent hyper-programming tools for Grasshopper. For example, a browser could be constructed to traverse name and type servers and display representations of the types of the stored entities. A simple tool could permit capabilities for discovered entities to be embedded in the source code of programs under construction. A more sophisticated program editor could directly map executable programs and data into the source code of a Grasshopper program under construction permitting the construction time binding of arbitrary program and data fragments. Furthermore, using the mechanisms described in this paper, this could be achieved in a programming language independent manner without the need to change language compilers.

6. Conclusion

The technique described in this paper is based on a number of mechanisms and conventions which have been constructed on top of the Grasshopper operating system. It was decided to incorporate the features as an additional layer on top of the existing abstractions rather than including them as operating systems abstractions in order to retain the flexibility and security offered by the basic operating system.

In the examples throughout this paper, only one type server has been shown associated with a few name servers; however, a running Grasshopper system will typically contain many name and type servers. Name servers are nested to form a hierarchy analogous to the way directories are nested in a file system. However, since capabilities are the only protection mechanism in Grasshopper, rather than a single hierarchy as found in Unix systems, each user is the owner of a hierarchy which may or may not be linked into other hierarchies. This approach lends itself to considerably more flexible and powerful high level protection regimes than is possible in systems such as Unix or NT.

A closer examination of name and type servers as described has revealed a wide range of possible applications for this technology; one consequence of combining a CORBA like approach with OMOS style libraries and linkers is to enable applications to link transparently to code on remote (possibly heterogeneous) systems.

As described in the introduction, the AppleScript system delivers the advantages of published application interfaces to the user. However, applications which do not export an AppleScript interface may not easily be accessed by AppleScript nor any other application. By contrast, all invokable containers in a Grasshopper system support an invocation point and (potentially multiple) operation numbers. These entry points are accessible to any application regardless of whether information about them is published in any form. Furthermore, arbitrary subsets of the operation numbers may be published via different name and type servers, permitting multiple views of containers.

Finally, our scheme provides a suitable platform for constructing a variety of general purpose tools to be constructed that were previously restricted to higher level, language specific environments. It is hoped these tools will provide programmers with more capabilities.

References

1. Cheriton, D. R. “The V Kernel: A Software Base for Distributed Systems”, IEEE Software, Vol 1, No 2, pp. 9-42, 1984.
2. Chorus-Systems “Overview of the CHORUS Distributed Operating Systems”, *Computer Systems - The Journal of the Usenix Association*, Vol 1, No 4., 1990.
3. Connor, R. C. H. “The Napier Type-Checking Module”, Universities of Glasgow and St Andrews, Technical Report PPRR-58-88, 1988.
4. Dearle, A. and Brown, A. L. “Safe Browsing in a Strongly Typed Persistent Environment”, *The Computer Journal*, Vol 31, No 6, pp. 540-545, 1988.
5. 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.
6. 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.
7. Deux, O. et al “The O_2 System”, *CACM*, Vol 34, No 10, pp. 34-48, 1991.
8. Farkas, A. “Aberdeen: A Browser allowing Interactive Declarations and Expressions in Napier88”, Honours thesis, Computer Science, University of Adelaide, 1991.
9. Farkas, A. M. “Program Construction and Evolution in a Persistent Integrated Programming Environment”, Ph.D. thesis, Computer Science, University of Adelaide, 1994.
10. Apple Computer Inc. “HyperCard Reference Manual”, 1993.
11. Apple Computer Inc. “AppleScript Reference Manual”, 1994.
12. Kirby, G. N. C. “Reflection and Hyper-Programming in Persistent Programming Systems”, Ph.D. thesis, Computing Science, St Andrews, 1993.

13. Kirby, G. N. C., Connor, R. C. H., Cutts, Q. I., Dearle, A., Farkas, A. M. and Morrison, R. "Persistent Hyper-Programs", *5th International Workshop on Persistent Object Systems*, San Miniato, *Persistent Object Systems*, Springer-Verlag, Workshops in Computing, pp. 86-106, 1992.

14. Kirby, G. N. C. and Dearle, A. "An Adaptive Graphical Browser for Napier88", University of St Andrews, Technical Report CS/90/16, 1990.

15. 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.

16. Sun Microsystems "rpcgen programming guide", Technical Report Revision A, of March 1990.

17. 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.

18. OMG "The Common Object Request Broker: Architecture and Specification", OMG, Technical Report 91.12.1, 1991.

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

20. Organick, E. I. "The Multics System: An Examination of its Structure", MIT Press, Cambridge, Mass., 1972.

21. Orr, D. B. and Mecklenburg, R. W. "OMOS – An Object-Server for Program Execution", *Proc. International Workshop in Object-Orientation in Operarting Systems*, Paris, France, IEEE, pp. 200-209, 1992.

22. Rashid, R., Tevanian, A., Young, M., Golub, D., Baron, R., Black, D., Bolosky, W. and Chew, J. "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, Palo Alto, ACM Press, pp. 31-39, 1987.

23. Stanley, M. "An Evaluation of the Flex Programming Environment", RSRE Malvern, Technical Report 86003, 1986.