Implementing References as Chains of Links

Julien Maisonneuve, Marc Shapiro, Pierre Collet

▶ **To cite this version:**

Julien Maisonneuve, Marc Shapiro, Pierre Collet. Implementing References as Chains of Links. Second International Workshop on Object Orientation in Operating Systems - IWOOOS 1992, ieeecs, Sep 1992, Dourdan, France, France. pp.236–243, 10.1109/IWOOOS.1992.252975 . inria-00444623

HAL Id: inria-00444623
https://inria.hal.science/inria-00444623v1

Submitted on 20 Feb 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Implementing References as Chains of Links

Julien Maisonneuve
Marc Shapiro
Pierre Collet

INRIA, Projet SOR*†
B.P. 105, 78153 Rocquencourt, France
E-mail: Julien.Maisonneuve@inria.fr, Marc.Shapiro@inria.fr

Abstract

The goal of this work is to provide uniform transparent access to objects, be they local, remote, persistent, or mobile. In this way, we facilitate distributed programming and persistence management.

An object (the *target*) is accessed through a *reference*. A reference retains its meaning as it is copied, stored, passed in messages, and as the target migrates. A reference is used to invoke a procedure (or *method*) of the target object. References support standard single-space targets, as well as fragmented objects [9]. The cost of using a reference to a local object is comparable to the cost of accessing the object through a pointer. In the implementation, a reference is a *chain* of *links*. Each link embodies a small piece of functionality. A chain may be composed of an arbitrary number of links.

They allow the target to be in another context, on another machine or even stored on disk. Fragmented and migratory objects are also addressed.

The cost of using references is close to the cost of a normal invocation in the local case, and is low in the case of local inter-context invocations. A references is a chain of links from a source to target, each link performing a specialized function, as fast as possible. The number of those links is variable according to the desired functionality of the chain.

This work was motivated by the lack of flexibility and scalability of most of today's identification and invocation schemes.

1 Introduction

Distributed systems programming has to deal with different programming paradigms to achieve distributed cooperation. A transparent, scalable invocation and designation scheme greatly simplifies the task of the programmer. Moreover, the ability to perform some work prior to every invocation if needed can provide useful language extensions.

To address these goals, we propose the concept of *References*. References are used like normal object pointers to invoke methods. The difference is that

*Julien Maisonneuve and Pierre Collet are also affiliated with Laboratoire MASI, Université Paris 6, 2 place Jussieu, 75005 Paris.

†This work is supported in part by the Ministère de la Recherche et de la Technologie

2 Previous Work

Distributed object identification has often been implemented with location-independent, system-provided unique identifiers (UIDs [8]). We see a number of problems with UIDs: they are costly to dereference, they do not conveniently support fragmented objects [2], and they do not scale well. UIDs imply knowledge of a global state that we would like to avoid in the design of a scalable system. Furthermore, UIDs differ greatly from ordinary pointers, forcing programmers to deal with two very different models.

Ports, used by some systems [10, 16] present similar problems. Also, they group a number of functions together that are not always needed at the same time, thus adding additional cost to elementary operations. Those functions are locating, naming, queueing, scheduling and communication protocols. Also, they hide distribution in a system layer that makes multiple policies hard to design due to lack of information and control.

In some languages, such as C++, it is possible to

devise user-defined pointers with special dereferencing operators (the overloaded -> or * in C++) yielding a raw pointer. Distribution, persistence, and other similar models may be hidden in such "smart pointers" [15, 4] using an appropriate dereference procedure.

An example is the permPtrs to persistent objects in SOS [13]. A permPtr contains the UID of the target persistent object and a raw pointer that is initially null. On first access, the dereference procedure maps in the target (if it's not already there), sets the pointer to its address, and returns the pointer. Subsequent use goes directly through the raw pointer. The main drawback is that a single policy is wired into the permPtr; experience with SOS proved that it was not always the most appropriate one. For instance, migratory or fragmented targets are not supported. Supporting multiple policies is possible in principle, but would require variable-length smart pointers, not easily accommodated by C++. Our references extend this model to support different policies in a more flexible way.

Another popular object identification abstraction uses stub objects. A stub has the same interface as its target, but all the operations are redefined to send messages along a connection to the remote target that is identified in the stub's data. In our opinion, stubs alone are not flexible enough when dealing with objects that can move, or whose location is not precisely known.

Stubs can be extended to handle persistent objects, as in Amadeus [6]. In this case, a dummy, uninitialised object of the same size as the target is allocated. Its procedures are redefined to trap to the system. On first invocation, the program traps, causing the data to be read in (overlaying the dummy object) and the procedures to be redefined to their proper meanings; the invocation is then restarted. Subsequent invocations execute the correct procedures, which find the data already loaded. This approach does not provide enough flexibility for a real persistence system.

An extension to the stub idea, called proxies [13], gives access to fragmented or replicated objects. In SOS, a fragment containing the data for a replica is imported (i.e. migrated in) on first access, as detected by the compiler. Its procedures must know how to communicate with the other fragments and maintain consistency. Using this approach, arbitrary policies can be implemented. The burden is placed on the proxy code, that can become quite complex and monolithic.

The Emerald language provides references supporting a few predefined cases [7]. An Emerald reference is a pointer to a descriptor that contains the target's data, a remote address (based on a UID), or a pointer to a further descriptor that itself contains the data. Emerald references support efficient access to local data and transparent access to mobile (local/remote) objects. Persistence is not provided.

3 Chains of Links

Our references are similar to those of Emerald. They allow transparent designation and access of objects that is independent of the object's state or location. They also offer transparent inclusion of special policies for sharing, placement or persistence. Finally, they support this scheme efficiently, keeping the cost of local invocations over references comparable with normal invocations.

A reference is a *chain* of *maillons*[1], each of that is similar to an Emerald descriptor. A *maillon* is also like a smart pointer, supporting a dereference procedure : at each level of *maillon*, the procedure performs a piece of work. This mechanism supports both efficient local pointers, and arbitrary-length chains implementing complex policies.

A *maillon* is composed of two parts: a data part (usually the identification of the next link on the chain) and a pointer to its dereference procedure that knows how to interpret the data part. This dereference procedure can change dynamically to accommodate the state of the object and provide different functions.

The simplest maillon acts like a *pointer*. Its data part is a direct pointer to the target. Invocations over this maillon results in invocation on the referenced object through the dereference procedure.

Another simple maillon type acts as an *indirection* to the next maillon that permit the creation of chains, each link providing a particular service. The next maillons address is stored in the data part. The dereference procedure simply jumps into the next maillon's dereference procedure.

These maillons are useful for example to create indirections in order to share objects, or to keep track of objects in some "object table" (e.g. Edelson's "root tables" [3]). A garbage collector could conceivably remove redundant indirection maillons.

More interesting cases are references to remote, persistent and fragmented objects. They imply the use of specialized stubs for each function.

[1] The French word for a link of a chain.

3.1 Reference to Remote Object

In our terminology, remote qualifies an object in another address space, be it on the same machine or not. The actual location of the remote object is totally transparent to the client.

A remote maillon is initialised with its dereference procedure pointing to an initial binding procedure. This procedure has to establish the conditions for subsequent invocations and will be called on first invocation.

More specifically, this binding function has to evaluate the reference contained in the maillon to decide which protocol is best suited to communication, and must establish the invocation path. It then completes the invocation. Further invocations will go more directly to the stub by resetting the dereference pointer to point to a simple indirection procedure (as in the local case).

In order to have an uniform interface, an invocation on a remote object must go through a stub object with the same interface as the target. The stub address will be used by the indirection procedure to relay invocation after the binding phase.

Usually, stubs do the marshalling as well as the communication part. We felt it necessary to separate both operations to keep the invocation of the remote object transparent wherever the target may be. We therefore divided what is usually called a stub into a marshalling stub and a connection stub.

Upon invocation of the methods of a marshalling stub:

- the procedure name and arguments are marshalled in a message,
- the connection stub is invoked with the message,
- the connection stub waits for a response message,
- the marshalling stub unmarshals the response message, and
- the marshalling stub returns the result.

On the other end, at the destination context, a *scion* is responsible for the reconstruction of the invocation, its delivery and the return of a reply. *Stubs* and *scions* are generic objects that perform the marshalling and unmarshalling of invocations: they are mechanically generated from the declaration of the class they represent.

Each reference potentially uses a different transport protocol depending on the location of the accessed object. The binding procedure can dynamically decide on the best protocol. If the referenced object appears to be in the same context, the maillon is transformed into a simple indirection maillon (its dereference procedure being an indirection to the local object). If it points to a different context on the same machine, a URPC or LRPC protocol [1] using shared memory can be used. Otherwise, a RPC-like protocol will be used to carry the invocation across the network. The latter two imply the creation of a stub whose interface conforms to the target object's interface. This stub will be referenced by the data pointer so that it replaces the object in the invocation.

3.2 Type Checking

In distributed environments, one may not always know the actual type of a transmitted reference, or guess wrong about it. The results of such errors can be unpredictable since scions will try to reconstruct invocations based on incorrect data specifications. To avoid that, we must perform minimal type checking.

Current compilers for well-typed languages are typically built on the assumption that the program can be type-checked statically. This is not necessarily true when we consider persistent or remote objects: we have to be able to type-check them to detect user errors. These compilers need to be extended to emit type information, that can later be used for run-time type checking [5].

The binding procedure can be responsible for limited type checking. Since objects are not supposed to change type dynamically, verification at binding time seems to be a reasonable option. The type information can be stored in the maillon itself or in a place where it can be recovered using the maillon.

A type mismatch error is usually a fatal error to the faulty process since it means that the program guessed wrong about the semantics of the objects accessed, thus making them unusable. This condition must be handled by a run-time module to produce sensible termination/recovery behaviour.

3.3 Reference to a Persistent Target

References can be used to access persistent data. We will only consider a simplified view of a persistence system, accommodating a single copy in memory and a single one on disk; the full architecture of a storage system, along with concurrency and consistency control between multiple copies is out of the scope of this paper [14].

A persistent object is stored on disk. It is copied into memory upon first access and then used as a local

object. If modified, it is copied back to disk after use.

As with remote targets, the binding procedure can be used to establish the conditions for performing an invocation. Applied to persistence, this binding function allows us to load the object before invoking its member functions.

The same remarks that were made about type checking in remote references can be applied to persistent references. The binding procedure is also the place to check for type conformance.

A maillon to a persistent object contains its target's identification (e.g. the target's name on disk) , a type field (containing the compiler-generated expected target type), and a pointer, initially null. On first access, the dereference procedure will: lock itself against concurrent updates, read in the object, check the read-in type against the expected type, set the pointer to the newly-read object, change the dereference pointer to the indirection procedure, unlock itself, and return the pointer. Subsequent accesses directly execute the indirection code. Unloading the object is relatively easy if there is a single such persistent target maillon; any aliases should be indirection maillons to the persistent maillon. To unload, atomically set the dereference pointer back to the initial load procedure and set the lock, then copy any modifications to disk and unlock.

A more realistic view of a persistence model would include a storage context interacting with user contexts. Multiple storage and consistency policies can be supported by using specialised stubs and multi-level maillons chains[14].

3.4 References to Migratory Object

Migratory objects are objects that are accessed in a single context at the same time, by phases, then in another, and so forth. The simplest policy to accomodate such objects is to move them where they are accessed, thus the name of Migratory.

A migratory object typically has several references pointing to it, a number of which are indirected through intermediate spaces. When an invocation is performed through one of the references, in the target context, the specialized migratory scion detects the access. It initiates the object migration while patching the other scions pointing to the object to point to freshly created stubs. The local references must also be patched to point to freshly created stubs, so we have to keep track of them.

On the invocation site, the object is reinstalled and scions are recreated for the incoming references of the previous location. Then the (now local) invocation can be performed. This requires again to patch the local reference.

3.5 Reference to Fragmented Object

A fragmented object (FO) is a distributed object made of fragments (local objects) located in different address spaces. In many cases the fragments in a particular address space can be known early and instantiated statically. With respect to the reference system, a static fragment behaves just like a stub, and the same techniques described in the previous section are used. In the general case however, the fragment must be instantiated dynamically by a "fragment factory". In this case, a mixture of the techniques described for persistent and for remote targets is used. When a reference to a FO is initially created, such as by receiving it in a message, in a migrated object, or in persistent store, it is in the *unbound* state [2]. An unbound reference has the following characteristics:

1. it identifies some remote fragment, but not necessarily the one that will ultimately be used;
2. the location of the remote fragment may not be precisely known
3. it does not support the full interface of the target, but only the single operation bind;
4. it must be bound before use.

Binding at first use is ensured by setting the head maillon's dereference pointer to bind.

bind is a generic procedure, independent of the target's expected or actual type. It sends a request message to the fragment indicated by the unbound maillon, passing identification of the invoker (taken from its context) and the expected type (stored at compile time inside the maillon). That fragment responds with the type of the caller's local fragment and with initialisation data, including a single *bound* reference, i.e. one with a precise location, and assigned to a precise interface of its target. The type and the initialisation data are passed to the language-specific runtime, to check against the expected type, and to create and/or initialise the local fragment, the address of which is returned. Finally, the maillon is set to indirect to the new local fragment. After this, the local fragment will communicate with its peers over the bound reference retrieved by bind.

It should be noted that fragmented objects are the general case. The mechanisms for local, remote, and persistent targets, specified in the previous sections, are transparent optimisations for special, restricted cases.

4 Distributed Reference Management

This section will review the general mechanism of the distributed references along with the invocation protocol and the garbage collector.

In order to be able to use a reference when it is transmitted to another space, we must register the fact that the referenced object is accessible from outside. This is done by creating a scion pointing to the referenced object. The scion's identification becomes the external identifier for the object , that will be transmitted to the outside world.

Thus, in order to create the scion when a reference is exported, references must be detected when passed in a parameter or as a parameter in a message. The automatic generation of stubs must generate the proper code when a reference is detected.

If this reference is again passed to another space, a scion-stub pair is formed in the intermediate space that is a simple indirection. That way, indirect chains are created between the source of the reference and its target. A way to avoid the marshalling-unmarshalling step in this case should be provided to make indirections cheaper, although this is an optimisation.

4.1 Shortening Chains

Liberal use of indirect reference chains allows a reference to be passed cheaply in messages while retaining the guarantee of reachability. However, when considering communication, references are harmful: not only because of the overhead and poor locality, but more fundamentally because sending a reference along an indirect chain creates yet another indirect chain. If this same reference is returned as a result, things deteriorate further. It is necessary to have a way of shortening unnecessarily long chains to avoid unacceptable remote communication delays. This work is performed by the invocation protocol itself.

In the initial binding phase, the initial connection message used to perform the type checking is sent to the destination. The reply to this message contains the actual location of the target (or rather of the terminal scion pointing to it); the reference is then updated and the invocations will follow the new short path.

When the binding step is over, the target object object can move as a result of object migration. The chain is reconstructed as explained before. The next invocation over the bound chain will result in a location exception being returned from the migrated space to the invoker with the identification of the next scion in the chain. The caller can then reset the destination field to point to the new location and restart the invocation.

4.2 Garbage Collection

Garbage collection is useful in centralised objects systems, but it is even more important in distributed objects systems. It allows us to avoid the problems associated with the tracking of remotely referenced data in an environment where they necessarily flourish.

Distributed references allow an easy integration with a distributed garbage collection system because we already keep track of outgoing and incoming references. The addition of a local garbage collector and of a specific protocol allows us to have a workable distributed garbage collector.

The distributed garbage collector uses the connection stubs to track external references and the scions to track incoming references. It also needs a small number of data structures for itself. Every *scion* contains an indication of the space that points to it. Also, to improve the fault-resistance of the protocol, timestamps are used and have to be included in the messages and in the stubs and scions.

The local garbage collector traces references from the local root and the set of all local scions. An unreachable stub is garbage and can be collected. The scion-stub pairs that become useless, i.e. through chain shortening, are automatically reclaimed whenever it is safe.

A connection stub associated with a marshalling stub contains a *locator* composed of two parts, called *strong* and *weak*. Each part points to a *scion* and consists of the identifier of the space containing the *scion* and the valid *scion* name within that space. The strong part identifies the *scion* that matches the associated stub. The weak part identifies a scion that, while part of the same chain of strong indicators, may be closer to the target object. Weak parts are used for communication and location, that rely on the invariant that the indicated scion will not be collected, being protected by the strong chain.

Distributed garbage collection relies on the invariant that (in the absence of failures) there is always an uninterrupted chain of stubs' strong parts and scions (hereafter called a "strong chain") between the source and target of a remote reference.

A scion may exist for which the corresponding stub no longer exists. This may cause the local garbage collector to believe that there is a remote reference to a local object, whereas in fact, none exists. For this reason, garbage collection is somewhat conservative.

The protocols ensure that eventually the unreferenced scion will be reclaimed. Then, the objects reachable only from that scion become garbage and may be reclaimed by the local garbage collector. In our scheme, all unreferenced scions will eventually be reclaimed. Thus, all local and acyclic distributed garbage will eventually be reclaimed (to the extent that the Local Garbage Collector is itself exhaustive).

The weak locator chain is used for invocations [2] and the strong chain is lazily short-circuited in a safe fashion as described earlier. Obsolete indirect stubs and scions will be cleaned up later by the garbage collector.

Application code sends and receives messages using a low-overhead "presentation-layer" protocol, with scions and stubs created or updated automatically as needed. Messages, in particular from the garbage collector, can be piggybacked to offer better performance.

A detailed account of the distributed management of references can be found in [12].

5 Reference Scope

In order to keep the system scalable, the scheme described above is valid only within a certain domain. Between domains are gateways that are responsible for translating protocols and naming conventions. Also, references must remain valid in the domain where they ares sent or be translated to remain so.

The gateways hosts scions that look terminal with respect to the garbage collector and the shortening protocol, but that are only specialised translators to other remote objects in another domain. That way references can span several domains but also be translated to interact with other protocols.

6 Implementation

This section gives a few comments about the implementation in progress. It is by no means a implementation report.

The references were designed without being specifically targeted to a precise language. However, we needed an object-oriented language, one versatile enough to allow us a flexible and efficient implementation. It turned out that C++ seemed a good option: it had most of the features we needed, we had some knowledge of it, and we had a (local) garbage collector in sight for C++.

The first design was very close to the original model, but not very well suited to C++. The maillons had (at least) two fields: one contained a `code` pointer and the other a pointer to a data part. The nature of the function pointed to by `code` determined the nature of the data part. The idea was to jump to our `code` function without modifying the stack, thus keeping the arguments for the real invocation. In the case of the local indirection, the function just had to perform the invocation. However, it was not easy to reconstruct the invocation from in code in certain cases (particularly dealing with virtual functions).

Our second try was based on two observations: first reconstructing the C++ calls by hand was a pain; second the code we had to generate to perform this invocation was excessively long because we had to guess what the compiler knew so well. It was not really slow, but we found another much cleaner way with comparable cost, and improved clarity.

The second implementation was a bit different: we let the compiler build the invocation itself. We redefined the `operator ->` to call our code function, itself returning the address of the actual object (or stub). It was also inlined so that `ref->foo(1)` became `(ref.(*code)())->foo(1)`.

The maillons are typed objects (actually C++ templates) that contain a code pointer and a pointer to an object of the referenced class, (or rather of the referenced abstract class, since we mostly care about interface conformance and do not want to inherit data). The procedure pointed at by `code` returns the address of a local object that can either be a real object or a stub.

As references are copied or passed as parameters, they must revert to an unbound state. This is why we use a copy constructor to reset the `code` pointer to the bind procedure. That way, we ensure that a reference chain is unique and that stubs are not shared, an essential property for garbage collection.

The stub and scions generation has to be done by a preprocessor (in fact almost a real C++ compiler, since it has to generate them from class declarations). The stub generation is still manual, but we're studying C++ precompilers for automating it.

Local references must occasionally be modified, like during object migration. When a reference is created, its constructor stores its address in a table so that the reference can be retrieved using the address of the object it points to. This is similar to the root pointer discovery in a local garbage collector.

[2] If objects do not migrate, the weak locator is guaranteed to indicate the space containing the target object.

The cost of local invocations is low because it requires only a added call of leaf, parameter-less function consisting in a return instruction. The indirection maillon (that calls another maillon) is even faster since the dereference procedure can be reduced to a jump to the next maillon's code procedure.

The current prototype supports local, remote and URPC invocations based on the same marshalling principle. Garbage collection and distributed reference management is under implementation.

7 Conclusion

We have designed a flexible, uniform reference system for local, remote, persistent, migratory, and fragmented objects, supporting a range of different policies as well as distributed garbage collection. Flexibility derives from two key design decisions:

1. a reference is implemented as a chain of links of arbitrary length;
2. each link has its own dereference procedure.

This design is the base for a scalable reference scheme not using UIDs, supporting distributed, robust garbage collection [12] and is the basis for an object-support OS framework [11].

We have presented a preliminary design that is under implementation. The system is designed with performance and flexibility in mind, and should reflect those goals in its complete implementation. We plan to have a working prototype and experiences at the time of the workshop.

Acknowledgements

We would like to thank David Plainfossé for his work on distributed garbage collection and for many talks that helped to shape up the system. We would also like to thank Daniel Edelson for his useful remarks on this paper.

References

[1] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, May 1991.

[2] Peter Dickman. Contrasting fragmented objects with uniform transparent referencesfor distributed programming. In *Proceedings of the SIGOPS 1992 European Workshop on Models and Paradigms for Distributed Systems Structuring*. ACM SIGOPS, 1992.

[3] Daniel R. Edelson. Precompiling C++ for garbage collection. In *Proceedings of the 1992 International Workshop on Memory Management*. Springer Verlag, 1992.

[4] Daniel R. Edelson. Smart pointers: They're smart, but they're not pointers. In *C++ Conference*, pages 1–19, Portland, OR (USA), August 1992. Usenix.

[5] Philippe Gautron and Marc Shapiro. Two extensions to C++: A dynamic link editor and inner data. In *Proceeding and additional papers, C++ Workshop*, Berkeley, CA (USA), November 1987. USENIX.

[6] Distributed Systems Group. Overview of the Amadeus project. Technical report, Trinity College Dublin, 1991.

[7] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[8] P.J. Leach, B.L. Stumpf, J.A. Hamilton, and P.H. Levine. UIDs as internal names in distributed systems. In *1st Symp. on Principles of Distributed Computing*, pages 34–41, Ottawa (Canada), August 1982. ACM.

[9] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Fragmented objects for distributed abstractions. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*. IEEE Computer Society Press, July 1992.

[10] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus distributed operating system. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, Seattle WA (USA), April 1992. Usenix.

[11] Marc Shapiro. Soul: An object-oriented OS framework for object support. In *Workshop on Operating Systems for the Nineties and Beyond*,

pages 251–255, Dagstuhl Castle, Germany, July 1991. Springer-Verlag.

[12] Marc Shapiro, Peter Dickman, and David Plainfossé. Robust, distributed references and acyclic garbage collection. In *Proceedings Of The 11th Annual Symposium on Principles of Distributed Computing*. ACM SIGOPS and SIGACT, August 1991.

[13] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating system — assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.

[14] Hervé Soulard and Mesaac Makpangou. A generic FO-structured framework for persistence support in distributed settings. In *1992 Workshop on Object Orientation in Operating Systems*, September 1992.

[15] Bjarne Stroustrup. The evolution of c++ 1985 to 1987. In *Proceedings of the USENIX C++ Workshop*, pages pp. 1–22. Usenix Association, Nov 1987.

[16] Avadis Tevanian, Jr. and Richard F. Rashid. Mach: A basis for future Unix development. Technical Report 4864, Department of Computer Science, Carnegie-Mellon University, Pittsburgh PA (USA), 1987.