DISTRIBUTED OBJECT ORIENTED PROGRAMMING: MECHANISM & EXPERIENCE

Douglas Steel*
Department of Computer Science,
Queen Mary and Westfield College,
Mile End Road, London E1 4NS
Email doug@dcs.qmw.ac.uk

ABSTRACT

This work is motivated by the desire to ease the process of writing distributed programs. We have implemented a mechanism whereby we can make an existing Smalltalk environment support distributed programming. We also present our experience of using this mechanism to implement both a program designed to be distributed from the outset and a single user application that has been modified to run on our distributed Smalltalk system.

1. Introduction

In distributed systems research we are looking for a programming model that allows us to express distributed programs clearly. Object-oriented programming, with its encapsulated objects and the concept of messages being passed between these objects fits neatly onto the client/server model of distributed systems. Current implementations of object-oriented environments do not directly provide support for such a model of distributed systems [Dollimore 91]. Our aim, in the first instance, is therefore to add remote invocation to an existing environment, in this case our local implementation of Smalltalk - *BrouHaHa* [Miranda 87]. The initial goal of this research into remote invocation is to provide access to a persistent object store [Dollimore et al 90], but it was found that it is a useful mechanism in itself. Eventually it is hoped to use the lessons learned in this approach to the design of a new environment to support distributed object oriented programming.

In our initial prototype we use a mechanism whereby we can add remote method invocation at the Smalltalk level, rather than modify the virtual machine to recognise when an object is not local. This approach may not be as efficient as providing the mechanism in the virtual machine, but is preferable for an investigative approach since it allows us to arrive at a solution iteratively. We do, however, make use of some network support that was already available in BrouHaHa since it allows Smalltalk programs access to the underlying operating system - in this case Unix.

To see how useful this remote invocation scheme is, several applications and tools have been implemented or ported. Among the first programs to be written specifically for this mechanism was a naming service intended to provide a tool for allowing users to select which remote objects they wish to interact with - the design of this program will be discussed as an example later. Another way to test the usefulness of the remote method scheme is to "distribute" an existing program. The Room Booking Service application discussed later is an example of such a program that has been distributed.

2. The Remote Invocation Mechanism

We aim to provide a flexible scheme for distributing object oriented programs. It is desirable to maintain a programmer's view consistent with conventional Smalltalk systems as this will allow for existing programs and programmers to be integrated easily [Bennet 87, Bennet 90, Schelvis & Bledoeg 88]. Essential to this task is the notion of transparency which means that the details of distribution are hidden from the user. It is not, however, desirable to constrain a program (or programmer) only to existing programming techniques, so we may wish that sophisticated programmers can manipulate the underlying system to enhance performance, security or functionality. We provide a simple mechanism which allows for *both* transparency and the manipulation of the mechanism [McCullough 87]. In Smalltalk (and other object-oriented systems) messages are sent to objects which perform operations on private, local data and then return results, these messages are called *method invocation*. For a distributed system we provide a similar mechanism, *remote method invocation*, which sends a method invocation message to a (possibly) remote object known as the *host object*.

2.1 The use of proxies for transparent remote method invocation

A proxy (or encapsulator) object performs a task for another (unavailable) object [Decouchant 86]. In Smalltalk it is possible for a proxy to imitate another object. All objects are notionally referred to by object-oriented pointers, or *OOPS*,

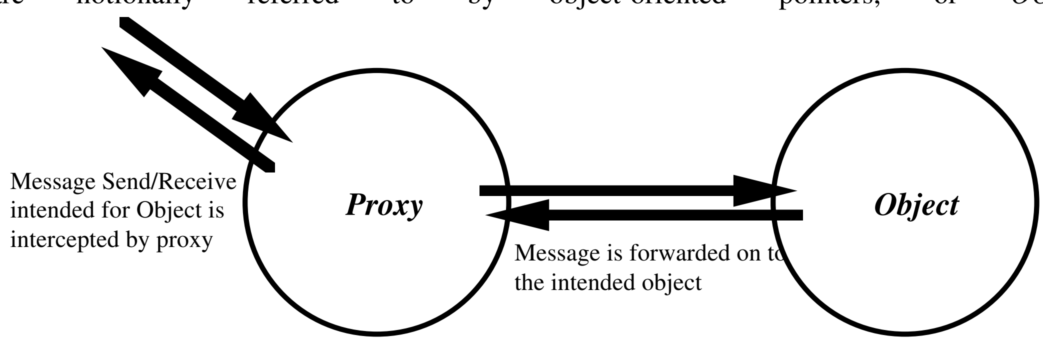

Figure 2.1 Message Forwarding

and by manipulation of these we can change the object that an OOP refers to. This means that all messages intended for the original object are now sent to the impersonator. It is also required that this impersonator can in fact do something with the messages that it receives.

Smalltalk provides for this by giving every object one method as an error catching mechanism. The method **doesNotUnderstand:** is invoked when the message sent to an object does not refer to any method that the object understands. Using this mechanism it is possible to catch all messages sent to an object and deal with them as required. A proxy object catches messages in this manner and forwards them to the object that they were originally intended for, this is illustrated in Figure 2.1.

---

*This work is supported by the Esprit High Performance Workstation Project

A proxy used in this simple way is effectively an extra level of indirection on the message send. If some extra processing is performed it is possible for this mechanism to be applied to several areas including remote forwarding and object faulting*. There is a difficulty in providing the extra processing required within the proxy as this would require the proxy to understand more messages than **doesNotUnderstand:**, this gives rise to the problem of *aliasing*, i.e. for whom is the message intended - the proxy or the original object ? This problem is solved by moving any extraneous functionality from the proxy into another object known as the *Forwarder*. All the proxy now does is pass every message onto the Forwarder for further processing. The proxy now need not concern itself with how messages are actually passed to the original object, which can be located in another computer accessed via a network, it has abdicated all responsibility to the Forwarder.

The counterpart of the forwarder is the *receiver*, this is a process which listens for incoming messages from forwarders and passes them on to the host objects. The overall architecture is illustrated in Figure 2.2. Several proxy/host object pairs can use one forwarder/receiver pair. The proxy informs the forwarder who it is, it is then the forwarder's responsibility to decide where the message is intended to go and how it is to get there.

A unique identifier is required for each object that takes part in a remote operation. This is so that the system can unambiguously decide which object a message is intended for no matter where on the network that an object is located. It is possible to create such a globally unique identifier, the *GLOP*, using the object's original network address and its original OOP. These two values together are sufficiently unique since there is only ever one machine with a particular network address and any particular OOP will not be reused within a reasonable time. A GLOP is an extended form of a LOOP (long object pointer) [Kaehler & Krasner 82].

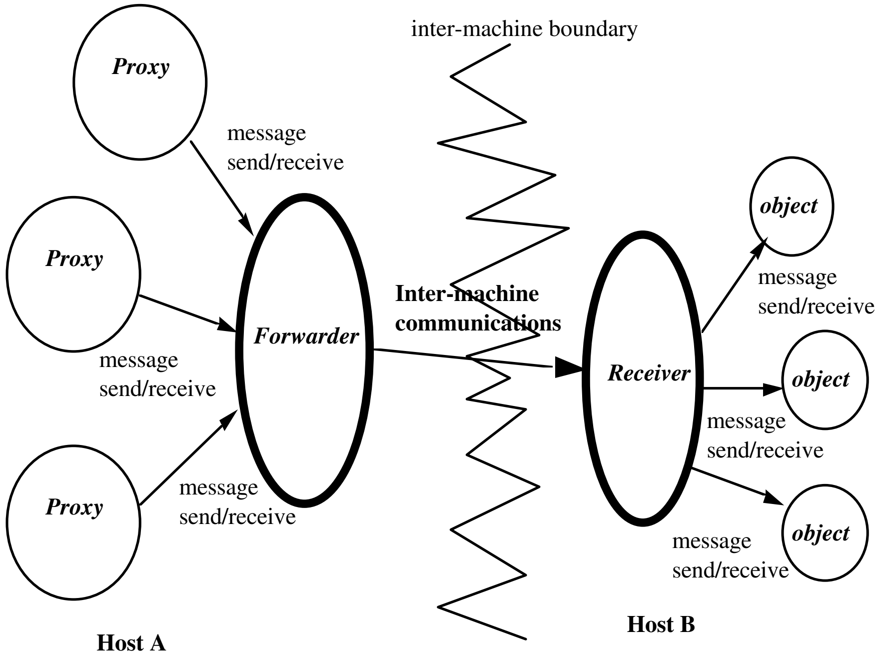

Figure 2.2 Overview of system architecture

* Object Faulting is a mechanism whereby objects can be loaded into an image on reference

Since objects are allowed to migrate freely across machine boundaries an identifier independent of the object's current network location and local oop is required.

The Forwarder has to have at its disposal all the information required to locate and send the message on to the host object. The information required includes the GLOP of the object, the last known network address, the strategies for packing and unpacking parameters, and the method for actually performing the communication, all this is stored in the *remote object table*.

The proxy mechanism as described above gives full distribution transparency, i.e. the programmer does not know where a proxy is located and cannot tell (except by measuring response time) whether the object is a normal local object or a remote object.

To increase the flexibility of the proxy mechanism it is possible to tailor proxies to specific objects. This allows the proxy to perform some processing on behalf of its host object. Such processing could add additional functionality to the proxy and/or reduce communications overhead by only passing messages on to the host object when absolutely necessary.

2.2 *Marshalling of a message*

When an object wants to send a message to another object in a remote machine, it gives the message to the forwarder. The forwarder is responsible, among other tasks, for the packing and unpacking of messages and replies.

Many RPC systems based on more traditional imperative languages use a stub generator or interface compiler which statically compiles marshalling code into the executable program [Tanenbaum & van Renesse 88]. Several such strategies for the marshalling of messages are possible and have been used in previous systems. Smalltalk presents us with some additional problems due to the late dynamic binding of arguments. We cannot tell what the arguments will be until the message is actually sent, thus ruling out any form of statically bound marshalling mechanisms. In other words, we have to perform all message marshalling at run time.

The vast array of possible object types in Smalltalk also presents us with the problem of deciding the most efficient amount of data to transfer without affecting the semantics of a message. In "local" Smalltalk all arguments are passed by object reference (*OOPS*) to which the receiver may then send further messages and so on to the arguments instance variables thus traversing parts of the object graph. To pass an argument by value we have to recursively descend the Smalltalk object graph for each argument, thus ending up sending an excessive amount of data across the network. We then have to send the object back after the method invocation if it has been modified in any way. This method has an advantage when we know that the arguments will not be modified in any way, such as simple *immutable* objects, however, as well as being different from local message sends, it is very difficult to predict which objects will be modified within a connected set of objects such as commonly constructed in object oriented programs.

Another possible solution is the passing of arguments by object reference. This allows the objects referenced by the arguments to be modified as they will always be residing on their original node. However, this method can suffer in other ways, for instance, a SmallInteger is actually smaller than the corresponding GLOP. A more serious disadvantage is that this method suffers from a very poor worst case when the object is referred to repeatedly at the receiving end - every access to the object has to be resent back across the network.

Passing arguments by object migration [Jul et al 88] is similar to passing by value except that the object need not be sent back to the originator after method invocation. The originator keeps a proxy for each object in the migrated collection and refers to the object via this proxy. One problem with this method involves knowing how far down the object graph we need to go before we stop migrating all the object's children. One solution to this is only to migrate the top level of the object and migrate its children on demand at a later stage. We are currently investigating this method for possible future implementation.

Another similar method is passing by object replication. In this method we would pass a copy of the object to the receiving object, again trying to determine the most efficient depth to descend the object graph. This requires a method for keeping replicated objects consistent with each other which is a research topic that we are currently persuing.

We provide a mechanism which is a hybrid of the pass by value and pass by object reference mechanisms [McCullough 1987]. Here we have distinguished a set of object types known as *immutable*. These objects can be sent over the network by value, since it is known that their values cannot change, examples of such objects are numbers, characters, symbols and special forms of character strings. All other objects are passed by reference. This mechanism results in good performance characteristics in the average case. In the cases where we use object reference worst case characteristics can still arise, however we have observed that this rarely happens.

A special case arises when trying to send character strings to our persistent object store. These are sent by value as if they were immutable objects. We do this because of efficiency reasons and also because the persistent object store requires these objects to “really” be sent to the machine that the store is running on. Such strings can have any size, although the current implementation limits strings to 2^{32} characters.

When the message arrives at the other side, it is unpacked (with the creation of any proxies as required) and sent to the receiving object, as if it were a message generated locally in the machine itself. The object performs the message and returns a reply (an object) to the Communicator (see below). The reply is packed in the same way as arguments and sent back to the “sending machine”. There, it is unpacked and received by the Forwarder (see below).

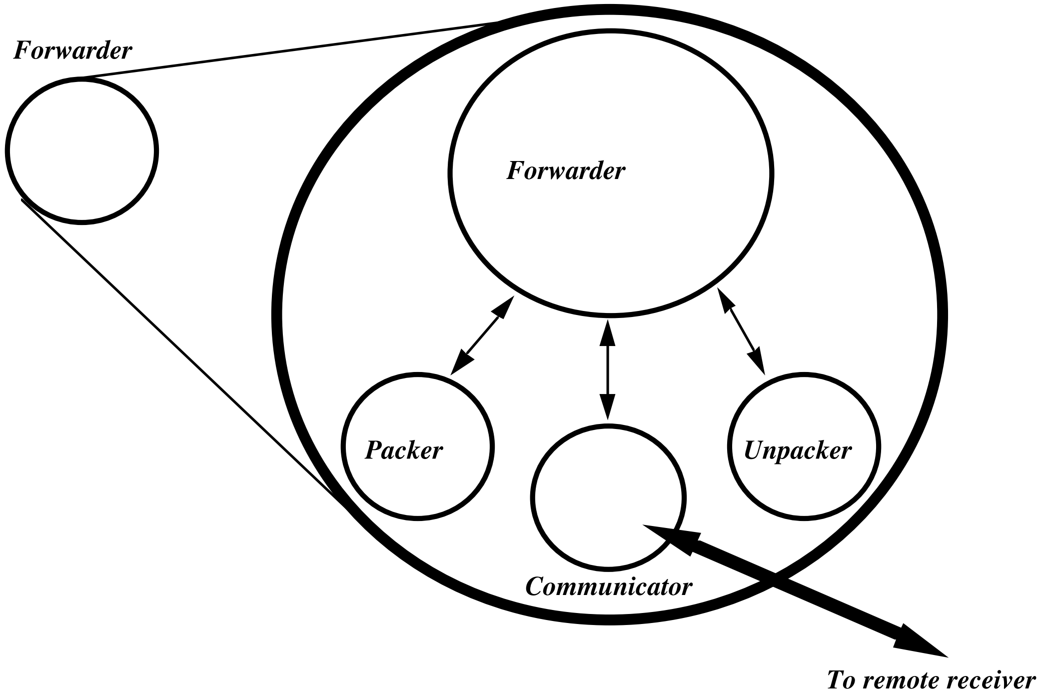

Figure 2.3 The Structure of A Forwarder

2.3 The Message Forwarding Mechanism

When a proxy receives a message it is passed onto a Forwarder for further processing. The Forwarder has to know how to marshal the message into the correct form for transmitting to a remote receiver, how to actually transmit the message and how to decipher the reply. The functions are carried out by the *Packer*, the *Unpacker*, and the *Communicator*. Their relationship with the Forwarder is depicted in Figure 2.3

The Forwarder obtains which packer, unpacker and communicator to use from the *ProxyInformationDictionary*. This dictionary is a list of all local proxies and the associated methods for forwarding. The forwarder then packs the message (see section 2.2), passes the packed message to the Communicator which sends it to the remote machine, and then unpacks the reply for passing back to the proxy. Using this approach it is possible for proxies to utilise different communication strategies depending on where the host object resides. For example it would be possible for an object to reside on a node that supports many different possible communication schemes (e.g. streams and datagrams) and choose the communicator appropriately for efficient communication with the proposed message traffic. In a heterogeneous environment this choice may be limited when the object wishes to communicate with a node that only provides a few possible communication strategies.

2.4 Current Status and Performance

We have built a working prototype of the system described above on top of our local version of Smalltalk - *BrouHaHa*. Although we have not modified the virtual machine to facilitate the implementation of the prototype, we have made use of several non-standard features provided by BrouHaHa. These features gave us access to the underlying Unix system calls for network access. Our prototype currently works on Sun and Apple A/UX workstations, and allows programs to be distributed across these different architectures.

We have fully implemented the proxy mechanism and have provided support for specialised proxies.

	Value Sent/Returned			
System		Self	Literal	Array
	Bennet	136ms	140ms	138ms
	Schelvis & Bledoeg	28ms	144ms	106ms
	Steel TCP	45ms	45ms	45ms
	Steel UDP	28ms	28ms	28ms

Table 2.1

We currently use a simple GLOP derived from the machine address and local oop of an object. We initially implemented a simple Communicator and Receiver using Unix sockets and UDP. A simple call by value packer and unpacker were implemented using the Smalltalk storeString mechanism. We moved to a more reliable TCP oriented protocol for the Communicator and Receiver, but found that the extra reliability is not always required and imposes an extra overhead. Our current marshalling strategy is the hybrid scheme described in section 2.2.

We have performed several tests on the performance of our mechanism and are very pleased with our results. Comparable results

were published in [Bennet 87] and our method invocation is around 3 times faster when using TCP and 4 times faster using UDP (admittedly Bennet was using Sun2's and we used Sun3/75's). Schelvis & Bledoeg use a different method of remote invocation which involves modification to the Virtual Machine. Their modifications impose a cost of around 10% to local invocations, ours doesn't, also our method has comparable performance for remote invocations (using UDP), their measurements were performed on equivalent hardware (i.e. Sun 3/75s). The main overheads in our implementation are marshalling and the creation/scheduling of processes to handle the messages. Communications overhead is reasonably low and we do not think a more optimised communications strategy will gain us much performance. Even so we find that our marshalling scheme has a far better performance than either Bennet's or Schelvis and Bledoeg's. The results are summarised in table 2.1. Note that the literal and array value are for immutable types in our & Bennet's figures.

3. Applications

3.1 *The Binder*

One problem that may arise in a large distributed system is that of knowing where to locate a particular service and what properties this service has. We provide a service which users can consult about such information -the *Binder*. Users can ask the binder for information about a service which has been registered with a supplied name. These services are implemented by objects and it is the *root* object of such a service that is registered with the Binder. The Binder should keep track of any service migrations so that services can be located when no longer resident at a given site (a proxy would be left behind in the original site, but this may not be kept up-to-date with further migrations, or may in fact be garbage collected if no other objects at that site require it). The Binder therefore provides a degree of name-space permanence, it does not guarantee that the service is still available, just that it is still known about.

The binder provides methods for registering a service, accessing binder information, removing a service and getting information about a registered service. The name, internet address and port number of a service, and a list of the methods it provides are stored in a service-object within the Binder.

The binder is also used to import a service into a remote machine. This functionality is aided by specialised proxies known as *Binder Proxies*. These proxies know the location of a fixed *Host Binder* which is used as the *Root* to a distributed object graph of Binders or other services.

The Binder Proxies are also responsible for creating object proxies when a service is imported from a remote site. The binder proxies also provide specialised functionality which enables a split in computational effort between the client and server machines, for example the Binder Proxies themselves relieve the Binder from having to create proxies.

Each user virtual image which wishes to use the binder mechanism has a **Binder** "global" variable (global in this case means global to the local image !) whose value is a binder proxy. The user or application then requests a service from the binder using a wildcard lookup scheme, similar to a directory search

e.g.

Binder lookup: "building*"

will return a list of all objects registered with names matching the given string. The actual binder instance will return to the binder proxy information sufficient for the binder proxy to either decide if the object is local and just return the actual object or create a proxy for the object if it is remote.

3.2 The Room Booking Service

The room booking service is an application that presents to the user a graphical representation of a list of buildings, rooms and time-slots, allowing him/her to browse through the list then place a booking in one of the free slots. The database of rooms/time-slots will need to be a shared resource. In the single user application this shared data is contained in a file, and is read in at start-up time, and the only form of sharing is sequential sharing - i.e. each user interacts with the application until finished and then writes out the new version of the data to the file. No form of concurrency control was implemented for the standard version. The distributed version will have to cope with a larger degree of sharing than the non-distributed version.

An initial version of the distributed room booker was very easy to implement, and only required the initialisation code to be modified. The initialisation code asks the binder for all available buildings, the binder proxy then creates a building proxy for each building. Buildings may be stored in several different objects or on a persistent store, and can be accessed remotely from workstations. Further information is only accessed on demand. The user will select a building and the application asks the remote building object for a list of rooms, these are returned to the application as proxies. Similarly the user selects a room, the application sends a request for time-slots to the remote room object and is returned a list of time-slot proxies.

Problems in the shared version would arise because of the interaction of several users with the shared time-slot objects. This was simply solved by using a shared object class (which serialises method invocations) for the objects that are shared, however this then highlighted another problem in that users do not always see the most up-to-date version of the shared data.

It is possible for a user to modify a time-slot while another user is looking at it and this change is not propagated to the remote users of the application. This could be solved by either registering proxies for the user interface with the server side and propagating **hasChanged** messages to these, or by polling the object to see if it has changed. Polling across a network can impose a heavy network and server load if done too frequently.

4 Conclusions and Future Work

The remote invocation method described in this paper has been used to implement several applications from scratch and to distribute several existing single user programs. We feel that the mechanism meets the goals for which it was designed in being easy to use and flexible. We are currently gaining more experience with the system and intend using it as the access mechanism for a persistent object store.

This is an on-going research project and we will continue to improve the prototype already implemented. We have discovered limitations in the Smalltalk user model which makes sharing and concurrency either difficult or ugly. Smalltalk was not designed with multiple users in mind, but does provide a suitable platform for the rapid prototyping of ideas. In the long term one goal may be to implement our own language and environment using the experience gained with Smalltalk. Our work is remaining in Smalltalk for the foreseeable future.

We are currently integrating the idea of GLOBs with the capability mechanism used in the persistent store [Dollimore & Wang 91], this would also allow protection between active volatile images on different nodes. The migration of objects would allow us to be able to implement various possibilities for improving the efficiency of argument passing. A multi-cast mechanism would help us to provide a method for keeping replicated objects consistent, allowing us to increase the availability and reliability of volatile objects. The proxy mechanism is applicable to other object oriented languages and it would be interesting to implement it on another system (such as C++ or Objective C). The current prototype could benefit from changes in the Virtual Machine to allow access to other methods of inter-machine communications, we would like to experiment with using Smalltalk on top of a true distributed operating system such as Amoeba or Chorus [Steel 91].

Acknowledgements

I would like to thank Eliot Miranda, for invaluable help with his implementation of Smalltalk - BrouHaHa and for the single user version of the Room Booking Program, Jean Dollimore and Wang Xu for their contribution to my ideas, Sunggi Min for the original version of the Binder, and Claudio Nascimento for initial work on the hybrid marshalling scheme.

References

[Bennet 87] John K. Bennet *The Design & Implementation of Distributed Smalltalk* OOPSLA '87 Proceedings, pp 318-330

[Bennet 90] John K. Bennet *Experience With Distributed Smalltalk* Software - Practice and Experience, February 1990, Vol 20, pp157-180

[Decouchant 86] D. Decouchant *Design of a Distributed Object manager for the Smalltalk-80 System* OOPSLA '86 Proceedings

[Dollimore & Wang 91] Jean Dollimore and Wang Xu *Using Capabilities to Ensure Correct Access to Shared Objects in a Distributed Object-Oriented System* QMW CS Report No.537

[Dollimore et al. 90] Jean Dollimore, Sunggi Min, Eliot Miranda, Claudio Nascimento, Douglas Steel, Wang Xu *Communicating Distributed Persistent Objects in Smalltalk* QMW CS Report No. 504

[Dollimore 91] Jean Dollimore *Programming Language Requirements for Distributed Systems* QMW CS Report No. 529

[Jul et al 88] Eric Jul, Henry Levy, Norman Hutchinson, Andrew Black *Fine-Grained Moblity In The Emerald System* ACM Transactions On Computer Systems, Vol. 6, no. 1, February 1988, pp111-133

[Kaehler & Krasner 82] Ted Kaehler and Glenn Krasner *LOOM - Large Object Oriented Memory For Smalltalk* Xerox PARC 1982

[McCullough 87] Paul L. McCullough *Transparent Forwarding: First Steps* OOPSLA 87 Proceedings, pp331-341

[Miranda 87] Eliot Miranda *BrouHaHa - A Portable Smalltalk Interpreter* OOPSLA '87 Proceedings, pp354-365

[Schelvis & Bledoeg 88] Marcel Schelvis & Eddy Blodeog *The Implementation Of a Distributed Smalltalk* Proceedings of the Second European Conference on Object Oriented Programming, pp212-231, 1988

[Steel 91] Douglas Steel *Evaluation Report On Modern Distributed and Object-Oriented Operating Systems* QMW CS Report No. 530

[Tanenbaum & van Renesse 88] Andrew S. Tanenbaum & Robert van Renesse *A Critique Of The remote Procedure Call paradigm* Proceedings Of The Euteco '88 Conference, pp775-783