A Remote Invocation Mechanism For Smalltalk and C++

Abstract

The object oriented programming paradigm with its notion of synchronous message passing between distinct objects maps neatly onto the client/server model of distributed systems. In this model, servers can be said to behave like objects with the notional message passing actually happening between nodes. When programming an object oriented system across a network of nodes it may be desirable to use differing languages for their best advantages, e.g. the simplicity and reactiveness of Smalltalk and the efficiency and ubiquity of C++. A generic model for building remote invocation mechanisms is developed and then applied to Smalltalk and C++. This project is an attempt to provide a system whereby different object oriented languages may be used together in a distributed system.

Contents

1 Introduction

There are several models of distributed programming, the most common and easy to understand is the client/server model. The client/server model is where a server process (or processes) provides a service for several client processes on (possibly) remote nodes. A generic mechanism known as Remote Procedure Call (RPC) can be used to provide transparency to the programmer.This is so that the remote service can be accessed and used as if it were a set of procedures in the local program. The underlying system translates these RPCs into messages that can then be sent across the communications network to the node where the server resides. The server then performs the request and sends a reply to the client.

The client/server model shows several similarities to object oriented programming model where (at least notionally) synchronous messages are passed between objects, an appropriate method is invoked and a reply returned to the originating object. It can be useful to combine these two models to provide a single *Distributed Object* model. In such a model we map the server onto an object and pass messages to it. A mechanism similar to RPC can be used, *remote method invocation*, to provide transparency to the programmer.

There is now a plethora of object oriented languages and systems and the characteristics of these may not make them all equally suitable for use in the client or server roles. Smalltalk-80, for instance, is a highly reactive language that is tightly bound to its user environment and is this highly suitable for the use as the client side interface of the client/server model, however its large and complex environment, and the difficulty in separating programs from this environment makes Smalltalk unsuitable for the use in building stand-alone server. C++ [Stroustrup 86], on the other hand, does not have such a highly integrated environment and can easily be used to build stand-alone applications thus making it a suitable language for building servers. The characteristics of the two languages makes it suitable to use Smalltalk as a client side interface for servers written in C++.

2. Motivation

As has been stated in the introduction, the object oriented programming model has much in common with the client/server model of distributed systems. In this section we will look closer at these models and the similarities between them.

2.1 Object-Oriented Programming

Objects are abstractions that appear to a programmer as encapsulated combinations of data and functionality. Objects of the same basic structure and functionality can be grouped together by using *classes* (c.f. prototyping languages such as Self []). Classes describe the structure and behaviour of objects. Classes may either be produced by the programmer or be included from class libraries. An object's state can be represented by a set of *instance variables*.

The functionality of an object is provided by a set of methods that operate on the state of that object. It is not possible to access the state of an object except through the defined interface of the object. This ensures encapsulation of the objects state. These methods can be invoked by sending a message with a set of arguments to the object. The receiving object then performs the operations described in the invoked method, this may involve further message sends to other objects, and returns a result to the originating object. A running object-oriented program can therefore be viewed as a network of objects communicating by message passing.

2.2 Distributed Systems - The Client/Server Model

The large number of current distributed systems are based on the client/server model. In this model the client process sends messages to a server process, which may be located on a different node, requesting it to perform operations on the clients behalf, these requests may be passed to the server by RPCs. The server may be a client of other servers and may use these other servers to perform parts of the requested operation (e.g. a file server may use a directory server to look up file names and a disk server to retrieve blocks from disk). The clients have no direct access to the internal state of a server and can only access the server via a defined server interface. The state of the server can thus be said to be encapsulated within the server process. Such a distributed system can be viewed as a network of client and server processes communicating by message passing.

2.3 Combining the Client/Server and Object Oriented Models

As stated above, both the object oriented and client/server models can be viewed as networks of distinct and encapsulated entities communicating by message passing. It is thus desirable to combine these two models into one distributed object model. In this model the servers of the client/server model can be viewed as objects with their internal state encapsulated in the object. The messages that are passed between clients and servers can then be viewed as method invocations across process boundaries.

The combined distributed objects model requires a mechanism to convert the notional messages within a local environment (i.e. within a single node and process) into real message that can be passed between environments (i.e. between processes and/or nodes) and a system by which these messages can be resent in the receiver's environment to the intended recipient object. Such intra-environment messages have to be in a machine and language independent since we intend to pass them between differing environments - in particular we will look at passing intra-environment messages between Smalltalk and C++ later.

Within each environment it is desirable for any extensions allowing distributed programming to maintain a consistent style with the standard language or environment. Different environments, such as those provided with Smalltalk and C++, have different ways of representing method names and objects. A language independent mechanism for representing and describing such concepts has to be developed. This will allow messages to be passed between different environments whilst being used in the different environments as if they had been generated locally. With a dynamic language like Smalltalk it is easy to generate messages at run-time and send them to an appropriate object. However, with a statically compiled language, all possible messages must be known at compile time along with a mechanism to select the appropriate message.

To define such a machine independent interface we can create an *interface definition language*. In this language we can specify the interfaces to objects, thus allowing automatic generation of code that translates messages between native and machine independent forms.

3. Modelling A Generic Remote Invocation Mechanism

In order to develop a generic remote invocation mechanism it is desirable to build a coherent and flexible model of both local and remote invocation. This model should be generic enough to be applied to vastly different object oriented languages. The basis of the model will be a local invocation mechanism. This model will then be extended to cover the case of remote method invocation. These models will be developed with reference to Smalltalk and C++ which, although object-oriented, differ widely in their style of invocation.

3.1 Local Invocation

An object consists of state information and code (methods) which operate on that state. An object oriented program consists of a set of objects that invoke operations on other objects. In order to invoke operation on an object, the system needs to be able to *locate* objects, *marshal* messages into a suitable format, *dispatch* the message to the destination object, and *activate* the method corresponding to the message. After the object has performed the required operations, any reply value has to be returned to the originating object, the receiving object de-activated and the originating object re-activated.

Often the activation/de-activation/re-activation process can be thought of as a thread of program execution. In this model objects are considered to be passive (i.e. are activated on the request of other objects and then automatically de-activated once the request has been performed).This is illustrated in figure 3.1. Active objects, sometimes called *actors* can remain active after the invocation has completed, thus allowing parallel threads of execution. Neither Smalltalk nor C++ implement active objects, although Smalltalk implements the orthogonal concept of *Processes*.

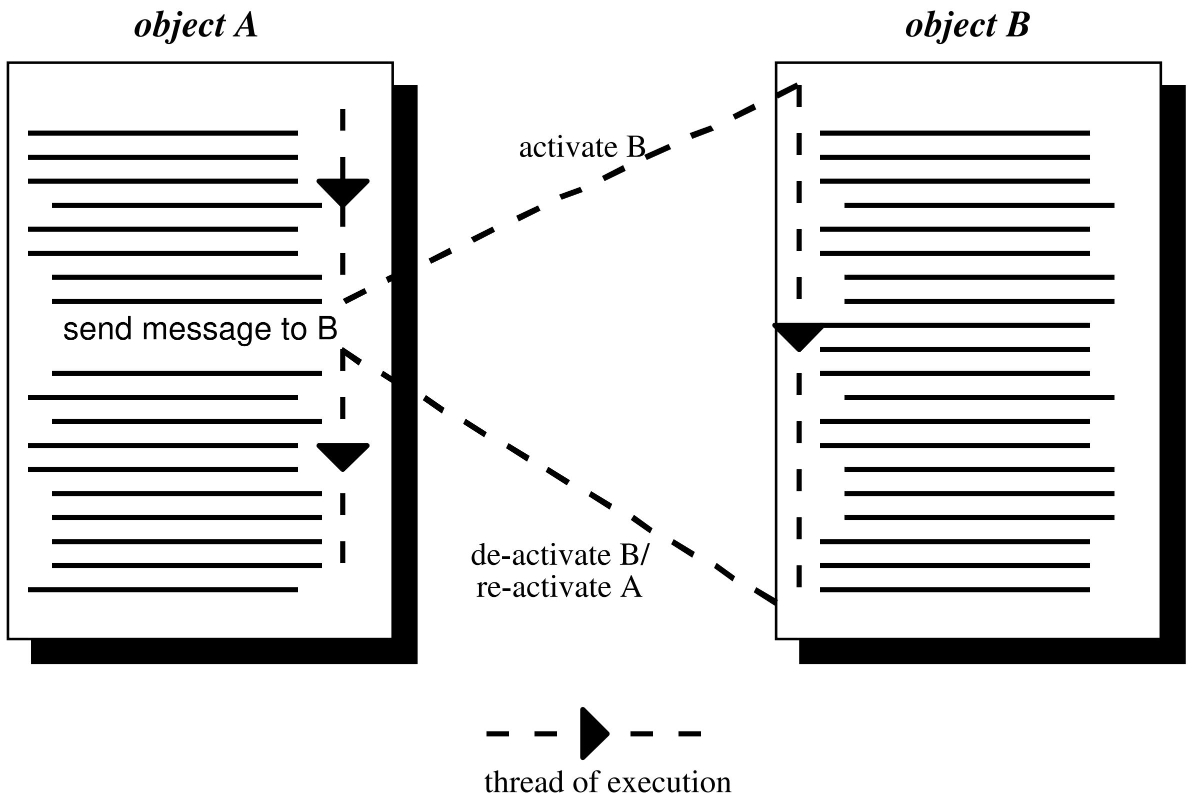

Figure 3.1

3.1.1 Locating Local Objects

The location of an object for local invocation is a vital element of an object oriented environment. This location must be both efficient and flexible. Efficiency is required since invocations are frequent, flexibility is required since objects may move around within an address space (due to such factors as garbage collection). Objects must be represented by an identifier which is unique within the environment to avoid naming conflicts. These identifiers are often implemented as pointers and are often known as *Objects Oriented Pointers (OOPS).*

OOPS may refer directly to an object's location or indirectly through an *Object Table*. Direct reference to an object's location is highly efficient, but makes it difficult to relocate an object, since all references to the object need to be modified to reflect the object's new location. Indirect reference solves this problem, at the expense of an extra level of indirection, by having all object references being an index into an object table that contains the actual location of the object.

3.1.2 Marshalling Local Messages

Messages have to be converted, *marshalled*, into a format that the receiving object will understand. On most systems this is a fairly simple process involving merely pushing the message arguments, normally OOPS, onto an invocation stack The receiving object, when invoked, then just pops the arguments from the stack. A system which involves an actual message send, a *real message system*, would have to generate a message that the receiving object would then parse to generate the arguments. Such a message maybe be a text stream or a binary representation of the message and its arguments.

3.1.3 Message Dispatching and Object Activation

Once a message has been marshalled into a suitable format then the message can be sent to the receiving object. Notionally this involves a real message send, but in most systems this can be optimised into a process very similar to a procedure call. Once the arguments have been pushed onto the invocation stack then a call can take place. This will involve saving the return address on the stack, creating a new environment for the called object, and then jumping to the appropriate method. This mechanism will also activate the receiving object, i.e. the flow of program execution will have been transferred from the caller to the callee.

On systems that do an actual message send the caller will have to block until the callee has finished execution. The callee needs to be scheduled to run once a message has been received and unmarshalled. To do this the message needs to be parsed and the appropriate method invoked. The scheduling can be optimised by the use of a *hand-off scheduler*.

3.1.4 Replying and Re-activation Of Caller

After the callee has completed the requested operation, a reply needs to be sent to the caller, the callee de-activated and the caller re-activated. On a true message based system this will involve marshalling the reply, sending the reply message to the caller, de-scheduling the callee and unblocking the caller. On most systems this can be optimised to pushing the reply onto the stack and, retrieving the return address of the caller from the stack and doing the equivalent of a procedure return. This will de-activate the callee and re-activate the caller.

3.2 Remote Invocation

Similar problems to those of the local case arise in the remote case. Namely, we need to be able to:

- locate remote objects
- marshal messages
- de-activate the caller
- perform the message send
- dispatch the message
- activate the callee
- return a reply
- de-activate the callee and re-activate the caller

Several new problems also arise, as well as the more complex communications between address spaces, if we are dealing with systems which fake the actual message send for efficiency reasons. The location of an object is also a more complex issue since there may be several possible address spaces to contend with as well as the objects within each address space.

Remote invocation is the invocation of a method of an object in a different address space. This requires a real message to be sent between the caller and callee. It would be possible to modify the run-time system of a real message system to detect when a remote object is referenced and then redirect the message to the remote object. However, since most current systems use an optimised message send mechanism, as described above, we need to have some way of trapping one of these *fake* message sends and generating an inter-address space message. This message can then be sent to the remote object's site where the fake message is regenerated and sent onto the intended object.

3.2.1 Locating A Remote Object

In local invocation an object has to have a unique identifier within its environment. A remote object also requires a unique name, but in the remote case the environment is much more complex, consisting of several disjoint local environments. An identification mechanism must therefore maintain uniqueness across different naming environments.

One solution to this is to attach an environment specifier to a remote objects identifier. This leads to a hierarchical identifier space, where an objects identity is dependent on both its current environment and its identity in that environment. Such a naming scheme is sufficient when objects cannot migrate between environments, however, this scheme will lead to an object changing its identity on migration.

Another solution is a flat naming scheme. In this scheme objects retain there identity even when migrated into other environments. To avoid naming conflicts between environments, the objects must have a separate set of identifiers from local objects (although a remote object may have a local alias). One way of achieving this is to have a central naming authority which will create a new global object identifier on request. Such a central authority may lead to innefficiencies and bottle-necks, especially if a truly global system is envisioned. It is becomes impossible to create new global objects if the central authority is unreachable for some reason (e.g. network failure).

A partitioned naming space is a variant on the flat scheme which allows a local environment to give newly created global objects a global name. This can be done without reference to a central authority and without causing naming conflicts between environments. Each local environment is given an environment identifier (as in the hierarchical scheme) and attaches that to a global object's local identifier forming a long identifier, sometimes known as a global object-oriented pointer or GLOP. This long identifier will refer to the object wherever it is located. The environment identifier in such a scheme only refers to where an object was created, not to where it currently resides, it is only used to avoid naming conflicts with objects created in other environments.

Most object oriented systems are optimised with a fixed OOP size, and thus cannot directly refer to a GLOP. It is possible to have a local alias for a global object so that a pre-written program need not be modified to cope with GLOPs. A local environment containing aliases for global objects needs to be able to realise when such objects are referenced. It is possible to modify a message dispatch mechanism so that it can recognise aliases by tags in the OOP. This requires reserving space in the standard representation of an OOP for such tags, and checking on every reference to see whether the tag is set or not. Such a mechanism involves a high overhead for references to local objects, so is therefore not an efficient solution when the number of local references is higher than the number of remote references.

Another mechanism for recognising when a remote object is referenced is the *Proxy*. It is possible for a special object class to be created which will allow instances of the class to realise that they are not the final destination for messages and rather can then forward on any messages received. This proxy can be treated as a normal object and therefore does not result in any overhead for local object reference. A proxy may catch messages sent to it by two main mechanisms :

- understanding all possible messages and then forwarding the message to the intended object.
- understanding no messages and relying on the languages error trapping mechanisms to raise an exception that the object can then deal with by forwarding the message.

For a proxy to understand all possible messages, it is necessary to develop a proxy class which contains an implementation for all methods. This is a complex process and can best be done by proxy generator which parses a class interface description to generate a the proxy class. This approach is suitable only for static languages where the class interface remains fixed during program execution. Such a mechanism can be very efficient since a fixed interface allows static marshalling and errors due to invalid messages or arguments can be caught by the proxy, thus saving an expensive remote message send. Each method may have its own optimised marshalling and forwarding strategy.

When a proxy understands nothing (it must, however, understand how to deal with exceptions) it only needs to catch exceptions due to messages that it cannot understand. The exception handler is then responsible for marshalling and forwarding of the message. This mechanism can be less efficient than the static mechanism since it cannot be known until runtime what messages or arguments the proxy will receive. Not all language environments provide such an exception mechanism (e.g. C++), so this mechanism is not applicable such environments.

It is possible to combine the static and dynamic mechanisms to gain the flexibility of the dynamic mechanism and most of the efficiency of the static mechanism. This can be done by providing static proxy methods to handle the most common cases, with the exception mechanism trapping the messages that are not explicitly handled. To be able to do this the environment will require both the ability to raise exceptions on undefined messages and a proxy generator, and hence this mechanism it not suitable for all language environments.

3.2.2 Marshalling

In systems that fake local message sends, the message send is simulated using a procedure call or equivalent and arguments are passed on an argument stack. In this case, since the message send is contained within one environment, there is no possibility of the calling and the called object using different argument formats, they both use the same format. Hence no actual marshalling of the arguments is required. When a real message send is involved within a single environment all that is required is for the two objects to have an agreed standard for in memory representation of method selectors and arguments, and hence no great effort is required to marshal messages.

However when a message is sent between two environments it is normally sent as a byte stream[1]. Different environments may have different ways of representing arguments, e.g. they may have different byte-orders, floating point representations and OOP (pointer) sizes. Considerable effort is therefore required to make sure that the two environments can understand the same messages. It is necessary to define a standard representation of inter-environment messages[2].

This standard representation needs to define representations for such commonly used types as integers, strings and real numbers in machine and environment independent formats. Remote invocation specific entities such as GLOPS and inter-environment method selectors will also need defined.

It also needs to be decided what types of entities will be passed between remote objects. It would be possible to either send the whole object graph that a parameter is the root of. This may lead to excessive communications since an object graph can potentially be huge, and also if any changes to any object in the graph needs to be kept consistent at all sites where such a replicated object resides. Another possibility is to only pass references to parameter objects, however if a parameter is referred to often by the called object, then this will also result in a heavy communications load. Hybrid schemes have been developed, and it is possible to make use of the fact that certain classes of objects are small enough to be atomic immutable objects which can be passed around freely without the overhead of

---

[1]shared memory is another method for passing data between environments

[2]It is possible for an environment to send messages in its native format, and make the receiving environment responsible for translating this into its native format, but since we need to marshall messages onto byte streams, the extra overhead in marshalling to a standard format is minimal.

keeping replicated objects consistent [Steel 91b]. In such a scheme, all parameters that are immutable objects are passed as is and all other objects are passed as references. This cuts down on the overheads associated with both of the other schemes, whilst still retaining a large degree of flexibility.

3.2.3 De-activation of Caller and Inter-environment Message Sends

In a local invocation with fake message sends, the caller waits for its request to be performed by simply transferring its thread of control to the called object. This transfer of control is achieved by the equivalent of a procedure call, and thus serial, synchronous execution is achieved. With real message send the caller needs to be suspended until the callee has finished execution, it can do this by waiting on a semaphore.

Achieving the same semantics on a remote method invocation requires some form of suspension of the caller's thread. If a real message send system is being used, then its built in mechanism can be used. How this is achieved in a fake message send system depends on the inter-environment message send/receive mechanisms provided by the environment.

Inter-environment message send/receive primitives can either be blocking or non-blocking, some environments can provide either or a mixture (e.g. non-blocking send with blocking receive). If blocking primitives are used then we automatically gain the local message send semantics, since the message send/receive will automatically block the caller until the callee replies. This simplicity is gained at the expense of concurrency within the environment[3].

Non-blocking primitives require extra overheads in dealing with the suspension/resumption of the executing thread. When the caller sends an inter-environment message it will wait for an reply by waiting for a semaphore. Non-blocking primitives may be polling or event-driven[4], but this can be hidden by the use of user-level semaphores. Event driven semaphores may be more efficient and provide a faster response time since they don't have to constantly poll the IO system or contend with other threads of execution when an event occurs.

3.2.4 Message Reception and Dispatch

An environment whose objects are available for remote invocation need to be listening for inter-environment messages.directed to these objects Using the message send/receive primitives discussed above it is possible to listen for and receive such messages. It is desirable in for such an environment to use non-blocking primitives to enable it to deal with multiple messages from both remote environments and its own local objects.

---

[3]blocking primitives may block the whole environment rather than just the executing thread

[4]or a mixture, e.g. an event signals that an IO operation is ready to proceed and the IO system is then polled to see which operation is ready

When a remote invocation message is received it needs to be decoded (demarshalled) and then dispatched to the appropriate object. The demarshalling is the reverse of the marshalling procedure, from the received byte stream, the receiver needs to extract the identity of the actual object that the message needs to be sent to, the message selector and the list of arguments. The extracted identity is in the form of a GLOP which the the receiver will lookup in a table of GLOP to local OOP mappings. The receiver the dispatches the message to the local object.

The message dispatching may take several forms depending on what facilities are available in the language environment. If the system can perform real message send then the message is formatted into the local message send structure and sent normally. If the environment supports run-time message generation (e.g. Smalltalk) then it is possible to construct a local message based on the remote message and (notionally) send that (even if message sends are faked).

On static systems which do not allow run-time generation of messages (e.g. C++) its is necessary to select the appropriate message from a precompiled list of messages and then perform the invocation. This selection may be performed by either a set of case statements or a table of pointers to methods.

The local object is scheduled for processing using the environments normal object invocation mechanism. No extra thread control is required and the listening/receiving thread is suspended until the task has been completed. If the listening/receiving is event driven rather than blocking then it may be necessary to provide concurrency control between objects which are accessed form different invoking environments. If the listener/receiver is blocking and single threaded then no concurrency control is required since conflicts cannot arise.

After the local objects has performed its task, the listener/receiver is reawakened using the environment's normal mechanism), and receives the reply from the object. This reply needs to be sent back to the originating system. This is done by the inter-environment message send/receive mechanism.

The originating environment then receives the reply from the remote object. If a blocking receive was used, then the act of receiving the reply unblocks the sender and it proceeds and returns the reply to the caller. If event-driven communications were used then a receive event is signalled which then signals the appropriate semaphore to unblock the caller and return the reply to it.

3.2.5 Other Remote Invocation Issues

Other issues arise when a local invocation mechanism is extended to encompass remote invocation. In the above discussion about the location of remote objects, it was assumed that the GLOP of a remote object was known and that the site where this object resides knew that this object may be accessed remotely. These issues can be resolved by the use of a name server or *binder*.

A binder is a service that knows what objects are available for remote access and associates human-readable names with them. An environment that wishes to enable an object for use by remote environments first registers this object with the binder and assigns it a name, service type, interface description (if applicable), the objects GLOP and current location.

When an environment wishes to use a remote object that it does not already know about from previous interactions, it will contact the binder[5]. A binder should have several modes of interaction allowing environments to find remote objects in several ways. A user, may for instance, decide to have access to any object that implements a specific service type (e.g. a printer service) or to a specific instance of such a service (e.g. a specific printer), or may just want to see what type of services are available.

3.3 Guidelines for Implementing The Generic Mechanism

Some of the issues for implementing the generic remote invocation mechanism have been raised in the preceding discussion. Before describing actual implementations of the mechanism in Smalltalk-80 and C++ in the following sections, some general guidelines for implementation will be established.

3.3.1 Determine the Local Invocation Mechanism

The local invocation mechanism has a large impact on the implementation of a remote invocation mechanism. Particularly, whether the local invocation mechanism uses real messages or fakes message sends using a mechanism similar to procedure calls. It has been shown above that a real message system is easier to adapt to a remote invocation mechanism. However, mosts systems implement local invocation by fake message sends for efficiency reasons.This complicates adding remote invocation to such a system.

In static language environments (such as most implementations of C++) method invocations are even more restrictive and are implemented as procedure calls through function pointers. These function pointers are used only to provide inheritance. This means that the method invoked on an object needs to at least have its signature known at compile time and cannot be generated easily at run-rime. As mentioned in §3.2.4, a general run-time method dispatch mechanism cannot be implemented in such environments. All possible invocations on an object need to be known at compile time and then the correct invocation needs to be selected at run-time.

Some dynamic language environments (such as Smalltalk-80) allow run-time generation of messages. These messages can then be easily dispatched to the object, allowing a general mechanism to be implemented.

Other features of the local invocation mechanism can affect the implementation of a remote mechanism. In a real message send system, message

[5]the binder has a fixed well-known address

sends are blocking and the caller is deactivated. In a fake message system the caller does wait fro the callee, but the message send is not normally a scheduling point and execution continues in the callers thread.

3.3.2 Implementation of Proxies in Target Environment

If support is to be provided in for an environment to access remote objects, then in this model, proxies have to be implemented ion the target environment. How proxies are implemented depends on whether the environment can catch unimplemented methods when they are invoked. The catching of such exceptions makes the design of proxies very easy, since the proxy only needs to know how to pass messages that it cannot handle onto something that can. If on the other hand these exceptions are not available then proxies that have exactly the same interface as the remote object have to be implemented. In a static environment

3.3.3 Define Basic Types, Immutables and GLOPs

In §3.2.2 it was mentioned that some objects may be classed as immutable objects and can therefore be passed by value without the overhead of keeping replicated objects consistent. When implementing such an argument passing scheme it is necessary that all participants in the remote invocation environment know how to understand and communicate these objects. It is therefore necessary to define which classes of objects can be immutable and how these are to be represented in a message.

The classes that can be considered immutable must have simple and coherent object representations. Some suitable classes are integers, real numbers and strings. An immutable object can only be created or destroyed, not modified.

3.3.4 Define Interface Description Language

An interface description language needs to be flexible enough to cover a substantial set of object interfaces. Since it is only concerned with the interface to objects it does not need to include any implementation details for the class. It only needs to support a list of method names with parameter types for marshalling purposes. It is desirable to support inheritance, since a large number of objects that are made available for remote invocation may have had their classes defined in terms of inheritance. This only needs to be a simple interface inheritance since the interface description language does not concern itself with implementation details.

3.3.4 Determine Inter-Environment Communication Mechanism

It is important to establish whether the inter-environment communication mechanism is blocking or non-blocking. A blocking mechanism is the easiest to use in the development of a remote invocation mechanism since it alleviates the need for synchronization and concurrency control. However, it may be less efficient if the environment supports multi-threading or an service object is required by several client objects, since blocking on communication means that other threads cannot proceed and other clients cannot access the service.

A non-blocking mechanism does allow other threads to proceed while a remote object is processing an operation and also allows several clients to simultaneously access a service object. This flexibility comes with the cost of increased complexity, the client and service objects need to synchronise, concurrency control is required when several clients access one service, and also thread control is required in a multi-threaded environment.

It is probably sufficient in a single threaded environment or in a prototype to rely on a blocking communications protocol, since this is easier to implement. However, when multi-threading is used or multiple clients are supported then it is desirable to use a non-blocking mechanism.

4. The Smalltalk Remote Invocation Mechanism

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. A simple mechanism that allows for *both* transparency and the manipulation of the mechanism is described [McCullough 87].

4.1 Implementing Proxies in Smalltalk

A proxy (or encapsulator) object performs a task for another (unavailable) object [Decouchant 86][Shapiro 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*, 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.

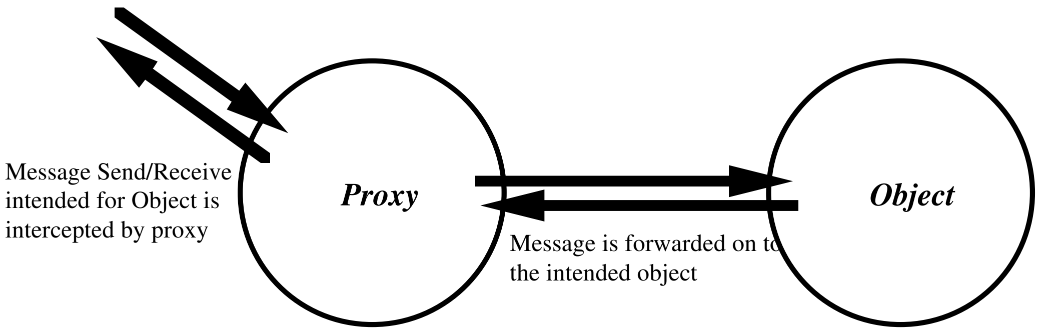

Figure 2.1 Message Forwarding

Smalltalk provides for this by giving every object one method as an exception handling 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 4.1.

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 causes 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 to 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 that listens for incoming messages from forwarders and passes them on to the host objects.

The overall architecture is illustrated in Figure 4.2. Several proxy/host object pairs can use one forwarder/receiver pair. The proxy informs the forwarder whom 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 that 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].

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.

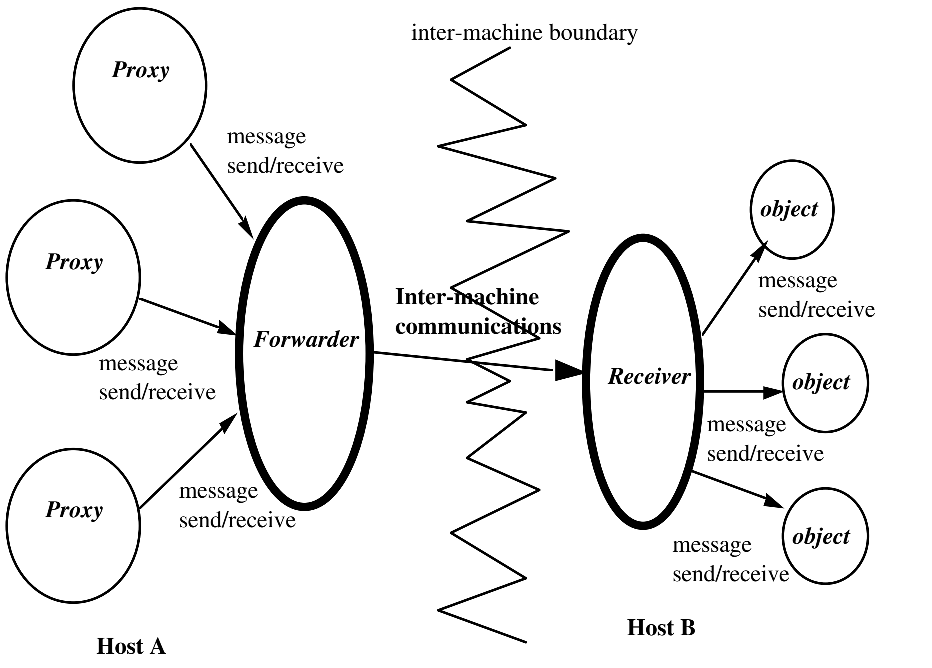

Figure 2.2 Overview of system architecture

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 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.

4.2 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 that 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 argument's 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 that 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 that is a research topic that we are currently persuing.

Provided here is a mechanism that is a hybrid of the pass by value and pass by object reference mechanisms [McCullough 1987]. Here a set of object types known as *immutable*. is distinguished 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 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).

4.3 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 4.3.

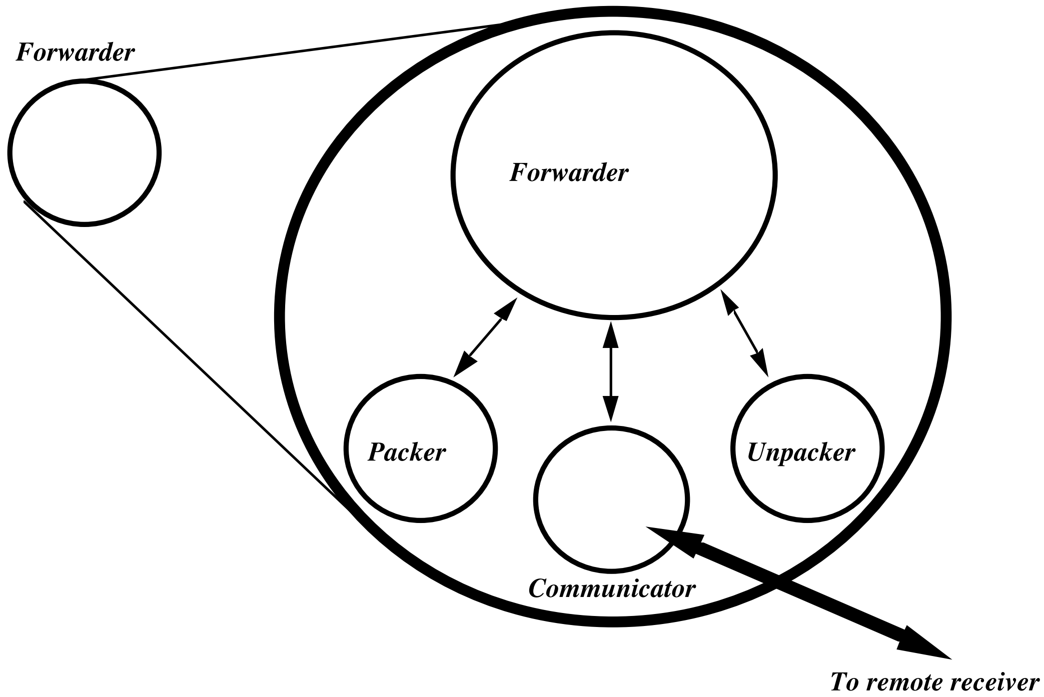

Figure 2.3 The Structure of A Forwarder

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 that 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 use 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). We therefore allow the object to 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.

4.4 Current Status and Performance

A working prototype of the system described above has been built on top of a local version of Smalltalk - *BrouHaHa*. Although the virtual machine was not modified to facilitate the implementation of the prototype, the use is made of several non-standard features provided by BrouHaHa. These features give access to the underlying Unix system calls for network access. The prototype works on Sun and

Apple A/UX workstations, and allows programs to be distributed across these different architectures.

The proxy mechanism has been fully implemented and support provided for specialised proxies. A simple GLOP derived from the machine address and local oop of an object has been used. The initial implementation of the Communicator and Receiver used Unix sockets and UDP. A simple call by value packer and unpacker were implemented using the Smalltalk storeString mechanism. A more reliable TCP oriented protocol for the Communicator and Receiver was tried, but it was found that the extra reliabilty is not always required and imposes an extra overhead.The current marshalling strategy is the hybrid scheme described in section 4.2. Several tests on the performance of the mechanism have been performed and are this system has shown to be better than comparable systems. Comparable results were published in [Bennet 87] and this method invocation is around 3 times faster when using TCP and 4 times faster using UDP (admittedly Bennet was using Sun2's and this test used Sun3/75's). Schelvis & Bledoeg use a different method of remote invocation that involves modification to the Virtual Machine. Their modifications impose a cost of around 10% to local invocations, this mechanism doesn't, also this mechanism has comparable performance for remote invocations (using UDP), their measurements were performed on equivalent hardware (i.e. Sun 3/75's). The main overheads in this implementation are marshalling and the creation/scheduling of processes to handle the messages. Communications overhead is reasonably low and it is not thought that a more optimised communications strategy will gain much performance. Even so this marshalling scheme has a far better performance than either Bennet's or Schelvis and Bledoeg's.The results are summarised in table 4.1. Note that the literal and array value are for immutable types in this & Bennet's figures.

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

Table 4.1

5 Remote Invocation in C++

After producing a Smalltalk implementation of the generic remote invocation mechanism, implementing it in C++ provided several problems. Many of these problems were due to the static nature of a compiled language such as C++ when compared to the dynamic nature of Smalltalk. In this section we will look at these problems, propose solutions, and describe the design and implementation of the C++ mechanism.

5.1 C++ Local Invocation

Early implementations of C++ were really only front-end to a C compiler and produced C code as output. Local method invocation was therefore implemented as function calls (albeit indirected through a function pointer). This means that fake message sends are used and local method invocation is highly efficient. Such early implementations provided little in the way of a run-time environment or object libraries. As C++ implementations were refined may disparate environments and libraries have been proposed and created, but most have retained the original local invocation mechanism.

As mentioned in §3.3.1 such statically compiled environments have several drawbacks when implementing the generic remote invocation mechanism. One major drawback is the inability of such an environment to generate messages at runtime. Messages have therefore to be dispatched by either :

- looking up a method name in a method name to function pointer table and calling the appropriate function.
- using a case (switch) structure to select the appropriate method to call.

Another drawback of C++ is the lack of an exception facility to trap invocations of unimplemented methods, meaning that a generic proxy cannot be constructed. Proxies thus have to be tailored for each class that is to be accessed remotely.

5.2 Basic Types, Immutables and GLOPS

The marshalling strategy used in the Smalltalk implementation described above makes use of several immutable object classes (e.g. SmallInteger) to allow the reduction of pass-by-object-reference. These immutable object classes need to have equivalents in any language that the Smalltalk implementation wishes to contact. Exact equivalents can be defined in C++ or can be translated from these objects into existing near equivalent object classes provided by a standard C++ class libraries. Since it is possible for the Smalltalk implementation to send any of its plethora of standard messages to one of these remote objects it would be desirable for the C++ equivalent classes to implement these possible operations. This will either result in a large library of new object classes that implement these operations or vast modifications and additions to existing classes.

A simple GLOP can be generated in a similar manner to that used in Smalltalk. Such a GLOP can be based on a unique local identifier derived from the local OOP and the network address of the node that the program is running on. However, this is not sufficiently unique in the C++ environment since several C++ programs can be executing simultaneously on one node. To overcome this problem and make a unique GLOP for a C++ object, the GLOP is also based on the process-id of the creating environment.

5.3 Definition of An Interface Description Language

Since messages cannot be easily generated at run time, all possible messages must be known at compile time. It is possible to define an interface description language such that all possible inter-environment method selectors and the structures of their argument lists can be defined at compile time. Using such an IDL it is possible to add a stage to the compiling process that can generate these message structures and code to recognise and select between them.

This approach is not required in the Smalltalk implementation since messages can be generated and recognised at run time from their text description. An IDL and compiler pass is an equivalent approach to that used by RPC stub generators in imperative languages [Wilbur & Bacarisse 87] [Rossum 89]. The stub generator and interface definition language required here needs to be able to cope with object oriented languages. The RPC stub generation languages that have previously been defined are not sufficient (AIL however does use a class based approach to provide inheritance between interface definitions).

It is proposed that the language used here will be based closely on the syntax of C++ and should allow most C++ programs to be easily converted to the distributed object model. The IDL could also be used within the Smalltalk implementation to provide extra safety (e.g. type checking) and allow automatic generation of proxy objects for C++ servers.

The full grammar of the interface definition language is in Appendix A. In general the syntax is similar to that of C++ class definitions although much simplified to make it easy to translate for different languages. A typical example of a class interface definition is shown below:

```
class student : person {
        void            assign_to(lecturer);
        void            will_attend(course);
        supervisor      supervisor();
        course_list     courses();
}
```

In this example a student inherits the interface of a person that may have methods and state to define such attributes as name and address. This class definition defines the interface of the assign_to(), will_attend(), supervisor() and courses() methods. It is then up to the programmer to actually define these methods. It is possible to generate such interface definitions from actual C++ class definitions by ignoring the extraneous information such as any implementation details, instance variables declarations and private method declarations.

5.3 Argument Passing Strategy

In C++ arguments can be passed by either value or reference, in Smalltalk all arguments are passed by value and hence arguments cannot be changed except by sending messages to them. If we are to provide a mechanism allowing Smalltalk programs to access C++ servers we need to disallow pass by reference so that arguments can be treated as in Smalltalk. All values that are to be passed back to a calling object should be passed back as return values and not as modified arguments.

The sending of messages to arguments should not be disallowed since this is allowed in the Smalltalk model. This means that the only argument passing strategy that is needed in the IDL is pass by value (except GLOPs that are really references). C++ programs are not disallowed form modifying their arguments, however to match the Smalltalk model these modifications will not be passed back to the calling program. These modifications can be accessed from the other environment since global references may be held. It is be required that any object that wishes to be migratable between environments be responsible for its own packing and unpacking down to the immutable object/global reference level, although some helpful routines may be available.

5.4 Inter-Environment Communication Mechanism

Since C++ is based on C, and the environment being used (GNU G++ and SunOS), the standard Unix networking primitives are available. These primitives were also used in the Smalltalk implementation and it was discovered that UDP was sufficient basis for a prototype remote invocation mechanism. UDP is therefore used in the C++ mechanism. To make the networking interface to the programmer simpler and more compliant with the object oriented paradigm, a set of networking objects have been defined. The main networking objects are socket and internet address objects.

Unix provides both blocking and non-blocking networking primitives and is able to signal the C++ environment when a network event occurs. Although it is desirable in a production environment to make full use of the non-blocking primitives, for the prototype implemented here, the blocking primitives are used. This limits the amount of concurrency available to the C++ programmer, but since most C++ environments have no notion of concurrency anyway, this is a small price to pay for simplicity of implementation. Future implementations should probably use the non-blocking primitives and a threads package to provide concurrency.

5.5 Implementation of the C++ Remote Invocation Mechanism

The interface description language is an integral part of the design of the C++ remote invocation mechanism. The stub generator that reads an interface description should be able to generate both proxies for a client program and the message dispatching mechanism for the server. Since one of the main aims of this project was to allow a Smalltalk program to access a server written in C++ it was decided to concentrate on this use of the mechanism first.

Handcrafting of the server method selection and dispatching allowed simple tests to be performed on the mechanism and showed what sort of C++ code the server stub generator should produce. The initial implementation was based mainly on the used of case statements to provide the method selection and dispatching. It also could only accept a limited range of object types and messages.

An initial implementation of the stub generator was developed using yacc to generate the C++ code used for selection/dispatch. This uses a set of templates that are then filled in with the appropriate information, e.g. method name, class name, argument class. A two level dispatch scheme was implemented, selecting first on the destination object's class and then on the method signature. The method signature is based on the the method name and its argument and return classes.

At the Smalltalk side, specialised proxies were handcrafted to produce inter-machine messages that would match the method signatures generated from the interface description. This allowed simple tests to be performed to show that the mechanism is feasible. All arguments are packed using the standard Smalltalk packer described in §4.2.

6. Status and Conclusions

6.1 Current Status

The Smalltalk remote invocation method described in this report has been completed and has been used to implement several applications from scratch and to distribute several existing single user programs. I feel that the mechanism meets the goals for which it was designed in being easy to use and flexible. The Smalltalk remote invocation mechanism was the subject of a paper written by the author [Steel 91b].

The C++ remote invocation mechanism is currently incomplete. Several immutable types have been implemented. This proved to be more laborious than anticipated since there is no way in C++ to provide such things as a generic Number class equivalent as can be found in Smalltalk. This results in a lot of duplication and modification of code, which is exactly what object-oriented programming is designed to prevent. The interface definition language has been designed, but the stub generator has not been completed so currently C++ stubs are generated by hand.

6.2 Conclusions

The generic model of remote invocation described in §3 is a viable way of converting a non-distributed object oriented programming environment into a distributed environment. The features provided by a language can help or hinder this conversion in several ways.

Dynamic languages, such as Smalltalk, tend to provide mechanisms for trapping messages that an object doesn't understand. This feature makes the implementation of proxies very easy. Allowing messages to be generated at run-time simplifies the message dispatch mechanism. These two features combined mean that the use of an IDL, server stub generator and client proxy generators are unnecessary.

C++, on the other hand, does not provide any mechanisms for trapping such unimplemented messages (one could be efficiently provided by the use of a protected virtual memory space that traps references to it, all of a proxies methods could then point into this protected space and hence unimplemented messages are trapped). This means a proxy generator is required. It is not possible to generate and dispatch messages at run-time and so a compiled message dispatcher is required to be built for each class that wants its objects to be remotely accessible. Thus a server stub generator is required.

Networking facilities were available in both the Smalltalk and C++ environments. Blocking and non-blocking primitives are both available in both environments. C++ does not have a notion of multi-threading and therefore the use of non-blocking primitives are not required. Smalltalk does have threads (known as *Processes*) and can therefore make full use of non-blocking primitives, however this may not be necessary since little use of multi-threading is used in Smalltalk environments.

The minimum requirements for an object oriented language environment to implement this remote invocation scheme are:

- networking facilities should be available for user level programs
- a mesage dispatch mechanism should be possible
- objects should be able to pass there messages onto other objects for further processing - i.e. proxies should be possible.

Desirable language features are:

- runtime generation and dispatch of messages
- exception mechanism that can trap unimplemented methods

Distributed systems in which multiple language environments interact are highly desirable, especially when the languages involved provide widely different facilities. In this case use can be made of C++'s lack of a builtin environment to provide lightweight services that can be run on any node on a network. These services can then be accessed from a heavyweight environment, such as Smalltalk, that need to be run on the users workstation.

It would be interesting implement the remote invocation mechanism in other suitable languages. Suitable candidates are Self, Eiffel, Objective-C and Modula-3.

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 Mobility 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

[Rossum 89] G. van Rossum *AIL - A Class Oriented Stub Generator For Amoeba* Proc. of the Workshop on Experience with Distributed Systems

[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

[Shapiro 86] M. Shapiro *Structure and Encapsulation in Distributed Systems: The Proxy Principle* Proc. 6th Intl. Conf. on Distributed Computer Systems 1986.

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

[Steel 91b] Douglas Steel *Distributed Object Oriented Programming: Mechanism and Experience* Proceedings of the TOOLS 5 Conference, pp 33-42

[Stroustroup 86] Bjarne Stroustroup *The C++ Programming Language* Addison-Wesley 1986.

[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

[Wilbur & Bacarisse 87] S. Wilbur & B. Bacarisse *Building Distributed Systems with Remote Procedure Calls* IEE Software Engineering Journal, vol 2, no 5

Appendices

Appendix A - Syntax of Interface Definition Language

The following is an extract from the yacc grammar of the interface definition language (terminals are in uppercase).

```
idl             :       EXPORT IDENT interface
                ;

interface       :       /* empty */
                |       interface class_def
                ;

class_def       :       CLASS IDENT inheritance_list '{' method_list '}'
                ;

inheritance_list:       /* empty */
                |       inheritance_list ':' IDENT
                ;

method_list     :       /* empty */
                |       method_list method
                ;

method          :       IDENT method_name '(' arg_list ')' ';'
                ;

method_name             :       IDENT
                ;

arg_list        :       /* empty */
                |       IDENT
                |       arg_list ',' IDENT
                ;
```