A Remote Execution Mechanism For Distributed Homogeneous Stable Stores

Alan Dearle[†], John Rosenberg[¥] & Francis Vaughan[†]

al@cs.adelaide.edu.au, johnr@cs.su.oz.au, francis@cs.adelaide.edu.au

[†] Department of Computer Science
University of Adelaide
GPO BOX 498
Adelaide 5001
South Australia
Australia

[¥] Basser Department of Computer Science
University of Sydney
New South Wales 2006
Australia

Abstract

Persistent languages and systems provide the ability to create and manipulate all data in a uniform manner regardless of how long it persists. Such systems are usually implemented above a stable persistent store which supports reliable long-term storage of persistent data. In this paper we consider the issue of distribution of the persistent store across nodes. A number of existing persistent languages with support for distribution are described in terms of a taxonomy of distributed stores. It is shown that there are considerable difficulties with these systems, particularly in terms of scalability. A new mechanism based on the exportation and remote execution of procedures is then described. A key feature of this mechanism is that an exported procedure may dynamically bind to data in the remote store. It is shown that the mechanism alleviates most of the problems of existing systems and provides considerable flexibility. The paper concludes with some examples of practical use of the proposed mechanism.

1. Introduction

Persistent languages and systems provide the ability to create and manipulate all data in a uniform manner, regardless of how long it persists. Thus data which lives longer than the program which created it may be constructed. Such systems do not require any flattening process since data persists in its original form. This results in considerable savings in terms of programming, debugging and testing [ABC83].

Orthogonal persistence means that <u>all</u> data may be persistent and that data may be manipulated in a uniform manner regardless of the length of time it persists. In other words, the right for data to survive for a long (or short) time is independent of the type of data. Similarly, programs manipulating data do so in a uniform manner, whether the data is short or long lived.

Persistent systems are usually implemented above a *stable store* which supports the long-term storage of persistent data. Such a store is stable in the sense that, following a crash or system failure, it can always return to a consistent state [lam81]. This may be implemented either by a checkpointing mechanism [lor77] or by a low-level transaction facility [HR83]. Several stable stores have been designed and constructed and are being used to support experimental persistent languages [bro89, mos89, ros91].

In this paper we consider the issue of distribution by examining the motivation for distributed stable stores. We then give a taxonomy of distributed persistent store mechanisms and two major models of distributed persistent systems are examined in detail. This is followed by a description of some existing languages and their support for distribution. Restrictions imposed by these languages are discussed and we propose a new language mechanism specifically oriented towards supporting multiple distributed stable stores. The paper concludes with some examples of the use of the proposed mechanism.

2. Distribution Models

The power of processors has been increasing at a rapid rate while at the same time their cost has been decreasing. Coupled with a corresponding decrease in the cost of secondary storage devices this has resulted in a proliferation of personal computer systems or workstations. It is now common to find a powerful workstation, with significant storage (gigabytes), on an individual's desk. In such an environment it would be sensible, for performance reasons, for the individual to have their own stable store on a local disk. This is analogous to having a local file system in a Unix environment.

Much of the time it may be possible to work only within this local store. However, for some applications it will be necessary to access shared data. For such applications it is desirable to support the notion of a distributed store, where parts of the store may reside on different machines, possibly in geographically separated locations. Distribution in this sense on Unix systems is supported by special protocols integrated with the file system, such as NFS.

It is logical in a persistent system to integrate distribution into the persistent store. However the best model for a distributed stable store is not clear. In order to categorise the different approaches we suggest the following stable store taxonomy.

Distributed persistent systems may be partitioned into one of two categories. In the first of these, attempts are made to hide distribution; in such systems, the user views the universe of discourse as one single large persistent address space. We term this the *one world model*. By contrast, in the *federated model* the user is aware that the universe consists of independent stable stores that may be stabilised separately.

When examining each of these approaches there are three issues that must be addressed. These are: protection, security and the propagation of pointers; in this paper, due to space limitations, we will concentrate on this last issue, the first two issues have been discussed elsewhere [RK90].

2.1 One World Model

It is tempting to consider the stores on all machines in a distributed environment to be part of a single large store; this characterises the one world model. Using this approach, pointers located in a stable store on one machine may freely reference remote objects. Such an approach makes distribution completely transparent and means that existing applications need no modification to operate in a distributed environment. This approach fits in well with the ideals of orthogonal persistence which state that all physical attributes of data should be abstracted over. However, there are some difficulties with this approach.

First, in a persistent store each object is usually given a unique name or address. If the entire network is considered part of a single store then these addresses would have to be large enough to uniquely identify any object on any machine in the network. Although some researchers have concentrated their efforts on building hardware with direct support for very large virtual addresses [RA85, coc89], such machines are not readily available and most systems are therefore constructed above conventional architectures. Some stores attempt to work around this problem by utilising contextual naming mechanisms [mos89]; typically with these designs, programs only manipulate short local addresses. Non-local addresses are stored in a special import table with import table addresses being distinguished from local addresses in some way. Such schemes are similar to the techniques used to implement the first generation of persistent object systems [CAC84] and have similar drawbacks. These include that they are difficult to manage, are not supported by most conventional architectures and, without hardware support an efficient implementation is difficult to achieve.

Second, the resulting store would be potentially huge. The management of very large stores is difficult. Issues which create difficulty in large flat distributed stores include allocation of free space, naming new objects to ensure uniqueness and the construction of appropriate navigation tools. Other difficulties are discussed in [mos89]. As with the problems faced with address sizes, the application of contextual naming schemes do help matters somewhat; however, the authors believe that in the final event, these schemes will not scale.

Finally there is the problem of stability; if the entire network is viewed as a single store then the stabilisation mechanism must capture the entire state of this store at a checkpoint. This involves coordinating all nodes and causing a synchronised stabilise to take place. The algorithms for achieving this are non trivial and are likely to be very expensive; once again, scalability is the main problem. Using two phase commit protocols over a high band width local area network, a modest number of reliable machines may be stabilised together. However, it is unlikely that such protocols would be successful when applied to large numbers of machines in geographically distributed locations.

2.2 Federated Stable Stores

For all of the above reasons, it may be prudent to partition the store into regions and make these regions visible at the language level. We term such a model the federated stable store model. Indeed, there are advantages in partitioning the store even on a single node.

Firstly, such partitioning allows for logical grouping of related data which may well improve performance in terms of disk access time, garbage collection and checkpointing overheads. This is also the motivation for contextual naming in the one world scheme and for generational garbage collection schemes [ung84].

Secondly, partitioning may also provide a level of protection where required between different uses of the store. For example, in a multilingual environment it is necessary to partition the store in order to ensure type security.

The main difference between this approach and some of the hybrid one world schemes (such as contextual naming) is the ability of the application programmer to perceive distribution. The motivation for this seemingly undesirable feature is to give the programmer clean semantics for failure. Using the one world approach, individual stores may crash and consequently disappear or, perhaps even worse, appear to travel backwards in time with respect to other stores. With the federated approach, the programmer is aware of store boundaries and therefore local data may be expected to behave differently from non local data.

Although we do not wish to address the problems of heterogeneity in this paper; it would appear that the federated approach also offers advantages over one world models with respect to heterogeneous store architectures.

The federated approach does have its disadvantages however; the most undesirable of these is the potential loss of referential transparency. This refers to a situation where two roots of a graph are independently saved in a store and these refer directly or indirectly to a common sub-graph. In a system which maintains referential transparency, only one copy of the common sub-graph is retrieved when either or both of the roots are restored. Federation by definition does not allow pointers to span stores therefore copies of data structures must be propagated between stores. Such copying may (and often does) violate referential transparency.

3. Language Approaches to Distribution

Before designing a distributed persistent system, one of the models of distribution must be chosen. Orthogonal to this decision, three other issues dominate the considerations which need to be made. These are:

1. how and when other stores are named,
2. how and when bindings are made to values in other stores and
3. how and when type checking is performed.

Using the one world model it may be impossible to identify individual stores which comprise that world. However, even with this model some notion of locality may be visible at the language level. With the federated stable store model the naming of other stores is one of the cornerstones of the programmers perception of the domain of discourse. The language mechanisms described in this section adopt different solutions to the problem of naming other universes.

It has been argued elsewhere [AM88] that the mechanisms used to establish bindings are especially important in persistent systems. Bindings may be made to locations or values and may be established dynamically or statically. In a distributed environment the binding mechanisms and their interaction with stability are subtle and complex. For example, it is hard to see how a static binding to some location in another store may be made without forcing the stores to stabilise together for as long as that binding exists.

Many database programming language designers consider strong typing to be essential. The motivations for this are obvious – long term data stored in a database is usually more valuable than data which has been created during a program invocation. The safety of this data cannot be compromised when accessed remotely, therefore type checking between stores becomes inevitable. Using the one world model, type checking may be transparent to the user whereas using the federated model the user must have a clear understanding of when and how type checking is performed.

Distribution also influences the algorithm used for checking the equivalence of two types. In a federated system the type equivalence algorithm must use structural equivalence. The one world model permits more freedom since it is logically equivalent to a single persistent store.

In the following sections we examine several different linguistic approaches to distribution and classify each according to our taxonomy. We then evaluate each system focusing on the three issues discussed above.

3.1 Conventional RPC

One of the most popular models for supporting distribution is the Remote Procedure Call [nel81]. In this model, a server machine exports a fixed set of services and a client may invoke these services via an interface that appears to the application programmer like a conventional procedure call. This has been a successful model and has been used in a number of distributed systems.

The rationale behind RPC is to provide a convenient and transparent mechanism for the programmer to execute requests on a remote machine, and hence provide an easy method of building distributed applications. The approach chosen by Birrell and Nelson [BN84] is to force the programmer to design a fixed interface of services that are invoked by a simple protocol. The interface is specified with a separate definition language, which when compiled produces stub code in the target systems language (e.g. Mesa in the Cedar system [xe81], C in RPCGen [sun88] and MIG [DJT88]).

The stub code provides both the client and server interfaces. The client interface contains code that builds the request message, copies all the parameters into the message block and sends the message to the server. The server code receives the message, unpacks the parameters and invokes the selected routine with the parameters from the message. The stub code is bound to the programmers code by the system linker.

The RPC approach is limited in a number of important respects; by necessity, parameters to the RPC are call by value, and this is enforced by the stub generator. In most RPC systems, the stub generator will not generate code that knowingly passes pointers between systems. However, this protection is achieved at the cost of another, separate, definition language. Furthermore the stub generator approach is not type secure. It is trivial to subvert the type checking in the stub generator, or indeed to generate RPC requests without the use of the generator at all. Such requests may contain arbitrary data structures including pointers. Only disciplined programming maintains store security.

Using RPC, the problem of binding context is trivially solved by binding the procedure instance to the server statically. However, such an approach causes problems in a persistent context. Although the kind of static binding does not imply a one world approach, no system evolution can take place. In particular, the RPC mechanism is unable to take advantage of changes within the server store that occur after it is built.

Once a service has been implemented it is impossible to extend its functionality without rebuilding the entire server. Furthermore it is difficult to comprehend how an optimal RPC interface could be constructed. In an effort to build one that is of maximum use, either a large number of atomic services could be provided causing a large communications overhead, or a service with an extremely complex interface could be designed with its own attendant problems. The inherent inflexibility of RPC is a serious weakness.

3.2 Distributed Shared Memory

A more recent and increasingly popular model for supporting distribution is Distributed Shared Memory (DSM) [li86, LH89, WF90, HR91]. The idea is essentially an extension of virtual memory to encompass a network. Each object is located on a particular node via a unique (network-wide) virtual address and all objects may potentially be addressed from any node. The system transparently copies remote objects on the first access and maintains a coherent view of the data at all machines.

This approach, which has been used in a number of persistent systems [HR91, KSD90] clearly falls into the one world model and has a number of advantages from a language point of view.

Since the network is fully transparent, the issue of naming stores need not arise, although it is desirable to provide a facility for specifying the machine on which a new object should be created. Binding and type checking are non issues since there is logically only one store. However, as we discussed earlier, the one world model does have major disadvantages associated with both the address size required and stabilisation which make the resulting system unscalable. An additional cost is the maintenance of coherency between the nodes; many systems use a single copy protocol with the more recent ones using a multiple-reader-single-writer protocol, this may result in considerable network traffic.

3.3 Argus

The Argus system [lis85] was an experiment in distributed systems which support long-lived data. The Argus approach is the epitomy of the federated stores model. In Argus, each store is protected by a software entity known as a *guardian*. Each guardian is an abstraction of a stable store and provides access procedures known as *handlers*: for every handler call, a process is spawned to serve it. Guardians may call handlers contained within other guardians as part of distributed transactions. To the user, a guardian appears as a programming data structure similar to an Abstract Data Type. For example, a mailer guardian may be defined as follows:

```
mailer = guardian is create
                  handles send_mail, read_mail, add_user ...

send_mail = ...

read_mail = handler( user : user_id, msg : message ) ....

add_user = ...
```

No global naming facility was implemented in the Argus system. However, Guardians may pass the names of other guardians as parameters to handlers and it would therefore be possible to construct such a facility using Argus. In addition to this mechanism, Argus also supports a *catalog* (sic) which registers guardians and handlers according to their type. Type checking in Argus is strong and static; at compile time, the compiler checks abstractions used by the compilation unit to ensure that they are consistent with their specification.

The passing of objects in handler calls result in copies of data being installed at the receiving ends. These copies of data are considered to be separate copies and referential integrity is not preserved. Semantic consistency amongst objects is therefore the responsibility of the applications programmer.

3.4 REV

Stamos and Gifford [SG90] present a model which permits the execution of code at another machine. Their work was motivated by the desire to enrich a conventional programming environment and to allow programmers to take advantage of specialised hardware (such as array processors) seamlessly within the programming environment. They present a generic model that they term REV (Remote EValuation) and a proposal for the integration of the model with the language CLU [LSA77]. Many of the ideas presented within the context of the REV work can be seen to have originated in Argus.

The REV model allows the programmer to select a server by specifying a set of attributes that the store must have. This is achieved using a language mechanism called *service* that specifies the names of abstract data types that are matched against those provided by servers. A run time call returns a server that contains the required set of ADTs. Specific naming of servers is not addressed although it could be achieved by providing each server with a unique service.

For example a service supporting matrix operations is statically specified as follows:

ArrayProcessorService = **service is** matrix **end**

REV requires that the compiler be able to statically resolve all the bindings required for a routine that will be remotely executed. It uses the list of ADTs specified in the server list to identify those references that will be resolved in the server. The processs of encapsulating a REV routine is termed *encoding*. An encapsulation includes all of the locally defined routines needed to satisfy the binding requirements of the REV; it is a compile time error if all these bindings cannot be resolved.

To make encoding tractable, REV explicitly prohibits the use of first class procedures or other constructs that would make it impossible to statically determine the REV routine's call graph. To prevent problems in concurrent execution and to ensure that the code portion of a REV is self contained, REV routines may not contain any free or own variables.

Since an REV request only ever sends the minimum code needed, and never sends code for services that exist within the server, requests which are a single procedure call to a server reduce to a conventional RPC.

At run-time, the predefined routine *Service* is called to provide a binding to a server with appropriate functionality and therefore the correct types. If no suitable server exists then a run time exception is raised. For example, a binding may be made to a server providing the matrix operations specified earlier as follows:

ap: ArrayProcessorService := Service[ArrayProcessorService]$Any()

The locally defined procedure called *exp* may be executed on the server bound to *ap* as follows:

answer: matrix := **at** ap **eval** exp(m, power)

In the form presented, the run time binding to the server and the actual REV request occur as two separate operations. It is assumed that the functionality of a store cannot change between the call of *Service* and the execution of a remote procedure.

In a persistent system this would result in the two stores becoming interdependent. The REV model as presented by Stamos and Gifford therefore falls into the one world category. However, the approach may be modified by combining the server binding and the REV request into a single operation. Such an approach would permit a federated approach, perhaps at some efficiency cost.

3.5 DPS-algol

Wai has designed and implemented a version of the persistent language PS-algol [ps87] called DPS-algol [wai89, wai90] which implements a superset of PS-algol supporting distribution. Wai argues that the ideal of orthogonal persistence demands that the programmer should have no knowledge of where objects are resident in the network; Wai is therefore an advocate of the one world approach.

DPS-algol provides three important extensions to vanilla PS-algol. The first of these is a process mechanism. The second is the provision of an RPC mechanism. Both these facilities are orthogonal to distribution and may be used in a non distributed context. The last extension is that machines (stores) are given symbolic names of type *locality*. New localities may be added to the system using a mechanism external to the language.

An RPC is initiated by specifying a process handle and an entry point within the process. For example, suppose that a server process called *server* exists with an entry point called *register* which takes an integer as a parameter, an RPC is effected using the following syntax:

```
    server@register( 3 )
```

In Wai's system, the semantics of parameter passing using RPC is the same as vanilla PS-algol, namely call by value.

The final extension to PS-algol made by Wai is the distribution mechanism. In DPS-algol both type checking and binding are resolved dynamically. DPS-algol permits a symbolic name (a string) to be associated with a process running in a locality. This name may be used to identify a process running in a remote locality. Type checking is performed by specifying the expected name and type of an entry point within a remote process. This is best illustrated by an example. Suppose that the server process shown above has the symbolic name "dataDictionary" and is running on a remote locality called "remoteStore". A distributed call of the RPC shown above could be instantiated as follows:

```
for server =         "dataDictionary" at remoteStore
            with register : entry( int ) do server@register( 3 )
```

If a process called *dataDictionary* is running on the machine called *remoteStore* and it has an entry point called *register* with the appropriate type, a one time binding will be made and no further binding or type checking is required for the duration of the associated clause (in this case the RPC call).

Since the semantics for local and remote RPCs in DPS-algol are the same, it is possible for pointers to leak across store boundaries. Thus there may be dependencies between machines and the corresponding stores must be stabilised together in order to guarantee consistency. As we have argued earlier, it is not clear how such a distributed checkpointing mechanism can be made scalable.

3.6 Emerald

The Emerald programming language [DHJ86] is a non-persistent object-oriented language for programming distributed applications. Emerald objects are mobile, that is they may be moved from one processing node to another with programmers choosing to ignore or exploit the concept of location. In [BHJ87] it is claimed that objects must be able to invoke other objects in a location independent manner. Such reasoning would suggest that in the context of persistent systems a one world approach should be advocated.

In the Emerald system, objects communicate via the invocation of other objects. Like CLU [LSA77] and Smalltalk-80 [GR83] the parameter passing mechanism used is call-by-object reference. However, since objects are mobile, referenced objects may be moved to the location of the callee.

Which objects are moved is determined by one of two mechanisms. The first of these is type dependent information; small objects such as integers and immutable objects are obvious candidates for automatic relocation by the run-time system. The other mechanism is a novel, programmer specified parameter passing mode termed *call-by-move*. Using call-by-move, objects are eagerly moved to the remote site prior to procedure invocation and may be moved back upon completion.

It is argued that such an approach increases performance of the system (which no doubt it can do). However, this approach also implies a one world model where (at least) the communicating stores must be intimately bound with respect to store stabilisation. If the stores did not stabilise synchronously, the semantics of failure in a persistent system would be unpredictable.

4. Napier88

In order to illustrate our communication mechanism we will use the language Napier88 [MBC89] as a framework. However, the mechanisms described in this paper may be incorporated into any programming language which supports at least first class procedures, parametric polymorphism and some global contextual naming facility. In the context of the Napier88 programming language, these facilities are provided by the Napier88 types **proc** (which may be quantified) and **env**. We discuss these types in this section.

4.1 The type proc

Napier88 supports procedures which are first class data objects. Consequently they may be stored, passed as parameters and be returned as the result of procedures. Like other Napier88 data types, procedures are declared using the reserved word “let”. The identity function over integers may be declared as follows:

```
let intId = proc( x : int → int ) ; x
```

the type of this procedure is written:

```
proc( int → int )
```

Napier88 supports parametric polymorphism, for example it is possible to write a polymorphic version of the identity function shown above as follows:

```
let id = proc[t]( x : t → t ) ; x
```

which is the identity procedure for all types; it has the type $\forall t.t \rightarrow t$. To call the procedure, the programmer may write,

```
id[ int ]( 3 )
```

which will return the value 3. It is important to note that this form of type polymorphism is completely statically checkable.

4.2 The type env

Objects of type environment [dea89] are collections of bindings, that is name-value pairs – they belong to the infinite union called **env** of all labelled cross products of typed name-value pairs. An empty environment may be created by calling the predefined procedure *environment*, which is of type,

```
proc( → env )
```

For example the programmer may write,

```
let e = environment()
```

which will create a new environment bound to the name *e*. The programmer may then write,

```
in e let a = 7
```

which will create a constant L-value binding and place the binding in *e*. To use the binding the programmer may write,

```
use e as a : int in
   writeInt( a + a )
```

which dynamically binds the name, type, constancy and L-value to the environment expression. If the environment contains at least that name, type and constancy tuple then the binding succeeds and the name is available in the following clause. The binding, which occurs at run time, and is therefore dynamic, is similar to projecting out of a union. The difference here is that only a partial match is required; other bindings not mentioned in the use clause are invisible in the qualified clause and cannot be used.

The distinguished root of the persistent store graph is of type **env** and may be obtained by calling the predefined procedure called *PS* which is also of type,

```
proc( → env )
```

Environments provide structure in the persistent store. All data objects resident in a persistent store may be found by traversing the object graph from the root environment. This ability is extremely important in a distributed context since it provides crucial knowledge of other stores. When writing Napier88 programs, it is always safe to assume that at least the root environment will be in a store. This knowledge allows programs to be written which can access any data that is directly reachable from the root on a remote machine. In the system described by Stamos and Gifford (REV) no such environment is available, which considerably restricts the power of their system.

5. A new communication paradigm

Here we present a new communication paradigm which has elements of the other mechanisms described above. It borrows from the Argus, REV and DPS-algol systems. We will show that this mechanism is:

1. simple,
2. powerful, and,
3. flexible.

The mechanism we are advocating provides a single polymorphic procedure which communicates with remote stores; we call this procedure *rx*, (Remote eXecute).

The syntax of r*x* is simple:

rx: **proc**[t](data : t ; code : **proc**(t))

The procedure takes two parameters: some data of a polymorphic type and a procedure which takes a parameter of the same polymorphic type. Note that since the data is polymorphic, any data may be transmitted using this type.

The semantics of remote execute are also simple. The procedure (*code*) is executed on a remote machine with the parameter *data* as a parameter. This is represented by the pseudo code shown below.

```
let rx = proc[ t ]( data : t ; code : proc( t ) )
begin
          copy the code and data to a remote machine and
          on the remote machine apply code with parameter data.
end
```

Since we have argued that pointers should not be allowed to leak from one store to another, like Argus the parameter transmission mode for *rx* is copy semantics. Therefore the transitive closure of both the code and the data must be copied to the remote machine before execution begins.

The *rx* procedure shown above does not specify the store against which the program should be executed. Instead, there is a mapping from the name of remote stores to procedures which permit the programmer to communicate with them. In order to obtain an *rx* procedure, the programmer is required to call a generator function which specifies the desired machine. Therefore, the *rx* generator function is as follows:

rxGen: **proc**(storeName : **string** → **proc**[t](t, **proc**(t)))

This mechanism has three advantages over passing the store name each time a remote execution is required. The first is that a table of remote stores needs to be accessed by just one function: the *rxGen* function. Secondly, since the *rx* functions produced by *rxGen* all have stores bound into their closure, there are some potential implementation efficiency advantages which could be exploited. Lastly, using such a mechanism there is no need for the code returned by the generator to be the same in each case; the procedures returned by *rxGen* may vary according to the nature of the remote store and the communications mechanism.

Since the mechanism described here permits data to be transmitted using copy semantics, bindings may not be made from one store to another. Instead, all bindings are made by the transmitted code with respect to the store to which they have been transmitted. Information can be transmitted back to the originating store using the same mechanism.

Using this approach, the code and the data transmitted to the remote site do not require type checking – all type checking is performed statically. Also, this scheme does not require any inter-store type checking since no information about the remote store is released at a language level. Here, the only thing that a programmer may assume about a remote store is that it contains a root environment of type **env** which is reachable by calling *PS*. We will see later that this is not as restrictive as it might first appear since tailored environments may be installed at a remote site.

In order to illustrate the utility of this mechanism we will give three examples of the use of *rx*; these are a remote execution facility with returned data, a simple object browser and an illustration of using this mechanism as a Trojan horse.

5.1 Remote execution with returned data

The simple remote execution mechanism described above may be used to construct a remote executor which returns values from the remote store. This procedure is also polymorphic, but by necessity is quantified over two types – the type of the parameter (called *p* in the examples) and the type of the result(called *r* in the examples). Such a procedure has the following form:

rx2: **proc**[p,r](data : p ; code : **proc**(p → r) → r)

The semantics of this function are the same as those of the *rx* procedure shown above with the addition that the procedure to be remotely executed returns a result to the store which initiated the remote command. A rough algorithm for *rx2* is as follows:

1. Construct a procedure (called *wrapper* in this example) to be remotely executed which takes a procedure, some data and the name of the initiating store as parameters.
2. Encapsulate within this procedure, the definition and remote execution on the initiating machine of another procedure (called *wakeup* in this example) which has knowledge of the initiating procedure.
3. Remotely execute the first procedure.

The procedure *wrapper* requires two different kinds of knowledge of the initiating store. Firstly, *wrapper* must be supplied with the name of the store to which it must return data; this is supplied as a parameter in a structure. Secondly, it requires knowledge of what it must do when it has completed its task; this information is contained in the procedure *wakeup* which is remotely executed at the initiating store using *rx*.

The *rx2* procedure uses a temporary location into which the returned data may be placed. Since this data must be accessible from the *wakeup* procedure which will be remotely executed on the instantiating store this location must be reachable from *PS*. For simplicity, the location has been created in the root environment; in practice the data would be stored in a more sensible environment.

The name of the initiating machine and the data passed to rx2 are placed in a record. This object is used as the data to an *rx* call which is also supplied with the procedure *wrapper*. The *wrapper* procedure is executed in the remote store and first unpacks the packed data. It then defines *wakeup*; this procedure is remotely executed in the initiating store with the result of the call to *code* as a parameter. Thus, the result of the remote execution is returned to the initiating client. In order to assist the reader, in the example below, comments associated with remotely executed code are proceeded by a "**". It should be noted that, in practice like *rx*, *rx2* would be encapsulated within a generator which permits binding to a remote store. For simplicity, this has been ommitted from the following example.

```
let rx2 = proc[ p,r ]( data : p ; code : proc( p → r ) → r )
begin
        type machineData is            ! This type is used to send the
                structure( name : string ; data : p )            ! data and the machine name
        let wrapper = proc( mcdata : machineData )           !** This is the procedure which
        begin                          !** will be remotely executed
                let sendername = mcdata( name )              !** First unpack the senders name
                let data = mcdata( data )                    !** and the real data.

                let wakeup = proc( returnvalue : r )         ! This procedure will be called
                begin                                        ! in the callers environment.
                        in PS() let result := returnvalue    ! It places a remote result back
                                                             ! into the callers environment.
                end

                let remoteResult = code( data )              !** Apply original procedure.
                let returnRX = rxGen( sendername )           !** lookup return rx command
                returnRX[ r ]( remoteResult, wakeup )        !** Send result back using rx.
        end

        let self = lookup the name of this store
        let packed = machineData( self,data )                ! Pack the data & store name.
        rx[ machineData ]( packed,wrapper )                  ! Call the wrapped function with
                                                             ! the original data and store name.
        use PS() with result : r in result                   ! Lookup the result and return it
    end                                                      ! to the calling procedure.
```

5.2 Example: a simple remote store browser

An example of this mechanism is a program which informs a user if a binding exists in an environment on a remote machine. For simplicity it is assumed that a simple name check is all that is required and that we will search only the root environment. In reality much more functionality would be required; however, the example is given as proof of concept.

In Napier, a function called *scan* is provided to iterate over Napier88 environments. It takes an environment and a procedure as parameters and applies the procedure to every binding in the environment. A procedure which checks to see if a binding associated with a name exists in an environment may therefore be written as follows:

```
let nameInEnv = proc( searchfor : string ; e : env → bool )
begin
        let found = false

        ! next define a function which will look for the name

        let lookOnce =  proc( name : string ; type : typerep ; const : bool )
                        if name = searchfor do found := true

        scan( e,lookOnce )                      ! Call scan which will repeatedly call lookOnce.
        found                                   ! Return found as the result of the function.
end
```

The procedure *getremotenames*, shown below, is effectively a type adaptor, required since *rx2* is only capable of remotely executing procedures with one parameter. *Getremotenames* takes a string as a parameter and merely applies the *nameInEnv* procedure with the supplied string and the root environment as parameters.

```
let getremotenames = proc( searchfor : string → bool ) ; nameInEnv( target,PS() )
```

This procedure may be remotely executed to search for the name "abc" as follows:

```
let rx2 = rx2Gen( "some store" )

let found = rx2[string,bool]( "abc",getremotenames )
```

5.3 The friendly Trojan horse

The mechanism shown in Section 5.2 could be extended to provide a browsing mechanism such as the one described in [DB89]. However, as the amount of code that is to be remotely executed increases so too does the amount of time required to transmit code. Furthermore, it is wasteful to repeatedly send the same code to a remote site. The *rx* mechanism deals with this problem by permitting customised interfaces to be remotely installed in other stores.

A customised protocol server may be installed in a remote store by sending an installation procedure to a remote machine. Once the code is installed it may be looked up at the remote site rather than being transmitted on every use. It is expected that this approach will be the usual way in which *rx* is used.

The use of a remote browser would take place in several steps. Firstly, a browser would be sent to the remote store with a script to install the code. An outline of this code is shown below. Notice that the procedure *browser* is being sent as data in this example.

```
let browser = proc( name : string )
begin
        ! perhaps a large amount of code....
end

let installer = proc( browser: proc( string ) )         !** This installs the browser in
begin                                                   !** the remote store.
        in PS() let mybrowser = browser                 !** This makes the code
end                                                     !** reachable from the root env.

rx[ proc(string) ]( browser, installer )
```

Once this code has executed, the programmer has the knowledge that a browser had been installed in the remote store. Future interactions with the remote code may be implemented via small pieces of code which look up the installed code in the remote store. The next example illustrates how this may be achieved by executing the procedure *usebrowser* in a remote store. At the remote site, the procedure first looks up the program installed earlier, next that procedure is applied using the given parameter.

```
let usebrowser = proc( name : string )
begin
        use PS() with mybrowser : proc( string ) in        !** Lookup the browser in the remote
        mybrowser( name )                                  !** store and apply it with the string
end                                                        !** as a parameter.

rx[ string ]( "abc", usebrowser )
```

Usually, a browser will return data to its caller, this may be easily achieved using the *rx2* procedure described above.

6. Implementation

The implementation of remote evaluation must ensure that it is never possible to have two stores in disagreement about the state of a remote execution instance. As an example of the problems encountered, consider the situation when one store sends a remote execution request to a second store and then stabilises. The second store may receive the request just after stabilising. Should a failure occur, resulting in both stores rolling back to their stabilised condition, an inconsistent state would result. Clearly, even if the federated approach is taken, when two stores interact it is necessary to link the stabilisations of the stores and to provide a mechanism to ensure synchronisation of the state of the stores, should roll back occur. The motivation for this work however is to avoid the one world model of distributed stable stores. We wish to keep the inter-store dependency to an absolute minimum. As proof of concept, the following algorithm is given for the case of two communicating stores.

When a client store wishes to initiate a remote evaluation the following actions occur. First, the client store is stabilised and secondly a request message is constructed containing both the code and parameters of the request. The message is tagged with a unique sequence number. This number may be a simple counter, however it must be derived in such a way that the same number will be generated should the store roll back and the message rebuilt. Such a counter is easily constructed in a persistent environment. Finally, the client store sends the request message to the server store and the client remains blocked awaiting an acknowledgment message.

Upon reception of the request the server store builds a process instantiation containing the code and data from the request. This process is entered upon the dispatcher queue of the server system, but is tagged so that it is not eligible for execution. Some post processing code is added to the stabilise and roll back mechanisms to explicitly traverse the dispatcher queue and enable execution of these processes once they have finished. Once placed on the queue, the server store is stabilised and an acknowledgment message tagged with the same number as the request is sent back to the client store. Once the store has stabilised the remote evaluation is eligible to run.

If in the future the client store rolls back it will do so to the point just before the message is sent. Therefore the server store will receive a duplicate message. The server will recognise the duplicate by its tag. The server can then simply discard the duplicate message secure in the knowledge that it is already processing the request. If the converse occurs, and the server rolls back, a duplicate acknowledgment will be received by the client which can also be safely discarded.

7. Conclusions

An inter-store communication mechanism has been described which although simple is extremely powerful. We have shown how such a mechanism may provide inter-store communication which is useful yet obviates the necessity to permanently link communicating stores. We believe that any requirement to make stabilisation of communicating stores permanently dependent on one another is unrealistic, especially in a geographically distributed environment. We therefore adopt the federated stores approach and propose a mechanism which uses copy semantics for transmission of both code and data.

We have proposed a syntactically simple interface for interstore communication. It is type safe and efficient due to the power of first class procedures, parametric polymorphism and the Napier88 type system. It permits arbitrary amounts of code and data to be transmitted between stores. This in turn allows highly specialised, customised interfaces to be constructed using a basic communication mechanism. The mechanism is enhanced by the Napier88 environment facility which allows an exported procedure to dynamically bind to data in a remote store. It is this facility which makes the proposed mechanism considerably more powerful than existing mechanisms which, in most cases, force static binding. Another interesting feature of the mechanism is that no special type checking of the transmitted code and data need take place.

At the time of writing this paper, this system is still so called *slide-ware*. That is, it has not been implemented. However, an architecture which supports multiple client processes connected to a central server is currently being constructed at The University of Adelaide. We intend to implement the mechanism described in this paper as an adjunct to this architecture.

Acknowledgments

We would like to thank Tracy Lo Basso, Kevin Maciunas and the anonymous referees of this paper for their helpful comments on earlier drafts of this paper. We would also like to thank Florian Matthes for his helpful observations during the workshop.

References

[ABC83] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R. “An Approach to Persistent Programming”, *The Computer Journal*, 26(4), pp. 360–365 (Dec. 1983).

[AM88] Atkinson, M.P. & Morrison, R. “Types, bindings and parameters in a persistent environment”, in *Data Types and Persistence*, Atkinson, M.P., Buneman, P. & Morrison, R. (Eds.) {*Proc. 2nd International Workshops on Persistent Object Systems*, Appin, Scotland Sept. 1987} pp. 3–20 (Springer–Verlag, Berlin, 1988).

[BHJ86] Black, A., Hutchinson, N., Jul, E. & Levy, H. “Object Structure in the Emerald System”, *Proc. OOPSLA '86* (Portland, Oregon Sept. 1986), pp. 78–86 (ACM Press, New-York, 1986).

[BHJ87] Black, A., Hutchinson, N., Jul, E., Levy, H. & Carter, L. “Distribution and Abstract Types in Emerald”, *IEEE Transactions on Software Engineering*, 13(1), pp. 65–76 (Jan. 1987).

[BN84] Birrell, A.D. & Nelson, B.J. “Implementing Remote Procedure Calls”, *ACM Trans. Comp. Systems*, 2(1), pp. 39–59 (Jan. 1984).

[bro89] Brown, A.L. “Persistent Object Stores”, Ph.D. thesis, Dept. of Computational Science, University of St Andrews {available as Technical Report PPRR–71, Universities of Glasgow and St Andrews, Scotland 1989}.

[coc89] Cockshott, W.P. “Design of POMP – a Persistent Object Management Processor”, in *Persistent Object Systems*, Rosenberg, R. & Koch, D. (Eds.) {*Proc. 3rd International Workshop on Persistent Object Systems*, Newcastle, N.S.W., Jan. 1989}, pp. 367–376 (Springer–Verlag, Berlin, 1989).

[CAC84] Cockshott, W. P., Atkinson, M. P., Chisholm, K. J., Bailey, P. J. & Morrison, R. “POMS: A Persistent Object Management System”, *Software Practice and Experience*, 14(1), pp. 49–71 (Jan. 1984).

[DB89] Dearle, A. & Brown, A.L. “Safe Browsing in a Strongly Typed Persistent Environment”, *The Computer Journal*, 31(6), pp. 540–545 (Dec. 1988).

[dea89] Dearle, A. “Environments: a flexible binding mechanism to support system evolution”, *Proc. of 22nd Annual Hawaii International Conference on System Sciences* (Kona, Hawaii Jan.1989), pp. 46–55 (IEEE Computer Society Press, Washington, D.C., Jan. 1989).

[DJT88] Draves, R.R., Jones, M.B & Thompson, M.R. “MIG – The MACH Interface Generator”. Technical Report, Dept. of Computer Science, Carnegie Mellon University (1988).

[GR83] Goldberg, A. & Robson, D. *Smalltalk–80: The Language and Its Implementation*, (Addison Wesley, Reading 1983).

[HR83] Härder, T. & Reuter, A. “Principles of Transaction–Oriented Database Recovery”, *ACM Computing Surveys*, 15(4), pp. 287–317 (Dec. 1983).

[HR91] Henskens, F.A. & Rosenberg, J. “A Capability–Based Distributed Shared Memory”, *Proc. 4th Australian Computer Science Conference*, (Sydney Feb. 1991) pp. 29-1 –29-12 .

[KSD90] Koch, B., Schunke, T., Dearle, A., Vaughan, F., Marlin, C., Fazakerley, R. and Barter, C. “Cache Coherency and Storage Management in a Persistent Object System”, in *Implementing Persistent Object Bases Principles and Practice*, Dearle, A., Shaw, G.M. & Zdonik, S.B. (Eds.) {*Proc. 4th International Workshop on Persistent Object Systems*, Marthas Vineyard Sept. 1990} pp. 103–113 (Morgan–Kaufmann, San Mateo, California, 1991).

[lam81] Lampson, B.W. “Atomic Transactions”, *Lecture Notes in Computer Science*, 105, pp. 246–265 (Springer–Verlag, New York, 1981).

[li86] Li, K. “Shared Virtual Memory on Loosely Coupled Multiprocessors”, Ph.D. Thesis, Dept. of Computer Science, Yale University (1986).

[lis85] Liskov, B. “The Argus Language and system”, *Lecture Notes in Computer Science*, 190 (Springer–Verlag, New York, 1981).

[LH89] Li, K. & Hudak, P. “Memory Coherence in Shared Virtual Memory Systems”, *ACM Transactions on Computer Systems*, 17(4), pp. 321–359 (Nov. 1989).

[LSA77] Liskov, B., Snyder, A., Atkinson, R. & Schaffert, C. “Abstraction Mechanisms in CLU”, *Communications ACM*, 20(8), pp. 564–576 (Aug. 1977).

[lor77] Lorie, R.A. “Physical Integrity in a Large Segmented Database”, *ACM Transactions on Database Systems*, 2(1), pp. 91–104 (Jan. 1977).

[MBC89] Morrison, R., Brown, A.L., Connor, R. & Dearle, A. “The Napier88 Reference Manual”. PPRR–77–89, Universities of St Andrews and Glasgow, Scotland (1989).

[mos89] Moss, J.E.B. “Addressing Large Distributed Collections of Persistent Objects: The Mneme Project's Approach”, in *Database Programming Languages*, Hull R., Morrison R. & Stemple D. (Eds.) {*Proc. 2nd International Workshop on Database Programming Languages*, Salishan, Oregon 1989} pp. 358–374 (Morgan–Kaufmann, San Mateo, California, 1989).

[nel81] Nelson, B.J. “Remote Procedure call”, Ph.D. Thesis, Dept. of Computer Science, Carnegie–Mellon University, Pittsburgh, Pennsylvania (May 1981).

[ps87] “The PS–algol Reference Manual fourth edition”, Technical Report PPRR–12 Universities of Glasgow and St Andrews, Scotland (1987).

[RA85] Rosenberg, J. & Abramson, D. A. “MONADS–PC: A Capability Based Workstation to Support Software Engineering”, *Proc. 18th Hawaii International Conference on System Sciences* (Honolulu, Hawaii Jan. 1985), pp. 515–522, (IEEE Computer Society Press, Washington, D.C., Jan. 1985).

[RK90] “Security and Persistence” Rosenberg, J. & Keedy J.L. (Eds.) {Proc. first International Workshop on Security and Persistence, Bremen, Germany, 1990} (Springer-Verlag, London, 1990).

[ros91] Rosenberg, J. “The MONADS Architecture – A Layered View”, in *Implementing Persistent Object Bases Principles and Practice*, Dearle A., Shaw G.M. & Zdonik S.B. (Eds.) {*Proc. 4th International Workshop on Persistent Object Systems*, Marthas Vineyard Sept. 1990}, pp. 215–225 (Morgan–Kaufmann, San Mateo, California, 1991).

[sun88] “Remote Procedure Call Programming Guide”, Sun microsystems, Part Number 800-1779-10, §3 (Sun microsystems, Cupertino, California, May 1988).

[SG90] Stamos, J.W. & Gifford, D.K. “Remote Evaluation”, *ACM Trans. on Prog. Lang. and Systems*, 12(4), pp. 537–565 (Apr. 1990).

[ung84] Ungar, D. “Generation Scavenging: A Non–disruptive High Performance Storage Reclamation Algorithm”, *ACM SIGPLAN Notices*, 9(5), pp. 157–167 (May 1984).

[wai89] Wai, F. “Distributed Concurrent Persistent Languages: an Experimental Design and Implementation”. Ph.D Thesis, Dept. of Computer Science, Glasgow University, available as Technical Report PPRR–76–88, Universities of Glasgow and St Andrews (1989).

[wai90] Wai, F. “Distributed PS–algol”, in *Persistent Object Systems*, Rosenberg R. & Koch D. (Eds.) {*Proc. 3rd International Workshop on Persistent Object Systems*, Newcastle, N.S.W. Jan. 1989} pp. 126–140 (Springer–Verlag, Berlin, 1989).

[WF90] Wu, K.L. & Fuchs, W.K. “Recoverable Distributed Shared Virtual Memory”, *IEEE Transactions on Computers*, 39(4), pp. 460–469 (Apr. 1990).

[xe81] “Courier: the remote procedure call protocol”, Xerox System Integration Standard XSIS-038112, (Xerox Corporation, Stamford Conneticut, Dec 1981).