The Implementation of a Distributed Smalltalk

Marcel Schelvis and *Eddy Bledoeg*

Océ Nederland, P.O.box 101
5900 MA Venlo, the Netherlands
+31-77-594036

Abstract

This paper describes DistributedSmalltalk, which consists of a number of cooperating Smalltalk virtual machines distributed over a network, that provide complete distribution transparency to the image level, including transparent message passing across machine boundaries. As a result no modifications are necessary at the image level and e.g. the standard Smalltalk debugger can be used for system wide debugging. Transparent I/O is provided by means of a concept called "home objects". The performance degradation is acceptable, due to replication and the home object concept. Replication is transparent and replication consistency is guaranteed, so e.g. for replicated class objects no compatibility checking is needed. Distributed garbage, whether containing cycles or not, is collected incrementally without any synchronization being necessary.

Key words and phrases

Smalltalk, distributed processing, distribution transparency, remote procedure call, incremental distributed garbage collection, replication.

1. *Introduction*

This report describes the design and implementation of a distributed programming system called DistributedSmalltalk, which is completely distribution transparent. Distribution transparency implies that programmers writing distributed applications, such as multi-authoring document systems, e-mail or calendar systems, need not worry about object access, network location, replication, concurrency control,

etc. DistributedSmalltalk is based upon an existing implementation of Smalltalk (Berkeley Smalltalk) and is implemented on a network of Sun workstations running Berkeley UNIX. It will serve as a vehicle for further research in distributed object-oriented computing environments and applications.

Object model in distributed systems

Distributed systems and applications are inherently more complex to program than non-distributed ones. To reduce complexity, much research work has been focused on tools that assist in the construction and programming of distributed systems. For example, message based operating systems like the V-system [11] and Mach [12] provide the communication means, but the programmer has to deal with message packaging/unpackaging and locating message targets.
With a remote procedure call facility like Sun's Rpc [10], the programmer can describe his application as a set of interacting modules via procedure calls.

Object-oriented languages like Smalltalk-80 [15] and Argus [13] can make life more easy. With these languages, the programmer can construct his applications in terms of communicating objects.
We believe that objects are an excellent way to structure a distributed system because they provide a means for data encapsulation. Data encapsulation is a powerful mechanism for controlling access to shared data.
Objects are the units of programming and distribution. Furthermore, an object can be seen as a computational unit, so therefore it has the potential for concurrent computation as exploited in POOL [14].
Regarding reliability, damage resulting from an error can be confined within an object. This property of objects is vital for fault tolerant distributed systems.

Smalltalk as a distributed system

The Smalltalk programming system can be seen as a set of objects that communicate with each other and with the user in a well defined way.
Until now, Smalltalk is a non-distributed system. Distribution of this system implies that the set of objects is divided in subsets, each subset residing on a different host. In order to call the resulting system a Smalltalk system, the semantics of the interaction between objects and between the user and the objects must not change. In other words no functionality must be changed.
The way functionality is implemented however will have to change and will depend on the location of the interacting entities (objects, users).

2. *The Smalltalk system*

Smalltalk is a programming system that provides functions like storage management, display handling, text and picture editing, compiling and debugging. The Smalltalk system consists of a virtual image and a virtual machine.

The virtual image

All components of the Smalltalk programming system are represented by objects. The set of all the objects in the system is called the virtual image. An object is a representation of a real world entity e.g. a display screen or an abstract entity e.g. a number. An object consists of some private data and a set of operations. The private data of an object can be manipulated only by its own operations. An object can be manipulated by other objects through its set of operations.
Objects communicate with each other by sending messages. A message is a request for an object to carry out one of its operations. A message specifies the name of the receiver object (receiver), the name of the operation (selector) and a list of object names as arguments. A message only specifies which operation has to be performed. The receiver of the message determines how the operation will be carried out.

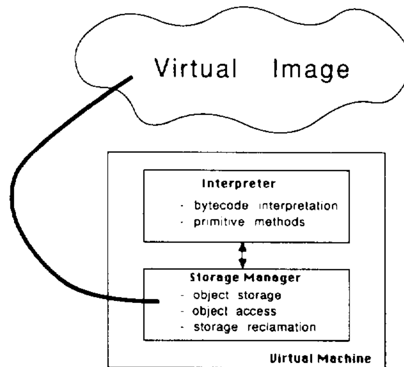


Figure 1. The Smalltalk system

A class is a set of equivalent objects. A class itself is an object. It describes the private data and the set of operations of its instances. Furthermore, a class provides the operation of creating its instances. Every object in Smalltalk is an instance of a class.
The private data of an object is described by its instance variables. An instance variable is a name which refers to one object, called its value.
Each operation of an object is described by a method. A method specifies manipulations of the object's instance variables and manipulations of other objects by means of message passing. A method specification is expressed in the Smalltalk programming language. A method specification is compiled into a bytecode representation, which is executable by a runtime kernel, called the virtual machine.
A small subset of the methods in Smalltalk are not expressed in the Smalltalk language. These are called primitive methods. Primitive methods are built into the virtual machine. Primitive methods allow the underlying hardware and virtual machine structures to be accessed.
A class can inherit methods from one other class, called its superclass. There is a class in Smalltalk,

called Object, which is the ultimate superclass of all classes.

The virtual machine

The virtual machine consists of a storage manager and an interpreter.
During the lifetime of the system objects are created, live for a while and die. The storage manager provides storage for the objects that make up the virtual image and reclaims the storage occupied by objects, that have died (garbage collection). The storage manager also provides access to the memory fields of objects.
The interpreter provides a stack oriented execution environment. It accesses the following objects found in the object memory: processes, compiled methods, contexts and classes. The interpreter fetches and executes bytecodes found in compiled methods. The state of execution of a compiled method is represented by a context. Contexts correspond to stack frames or activation records. The interpreter finds the appropriate compiled method to execute by searching message dictionaries of class objects.
Some compiled methods refer to primitive methods. If a compiled method indicates a primitive method, the interpreter does a dispatch and executes the primitive. There are about a hundred primitive methods that perform arithmetic, storage management and I/O operations. Process scheduling routines are also performed by primitive methods.

3. *Distributing Smalltalk*

Smalltalk-80 is a single user programming system. Multiple Smalltalk programmers can exchange objects only by writing objects (or source code) into a file, transferring the file over the network and reading the file at the destination.
There has been a number of attempts to make shared access of objects easier [1][2]. They all provide a transparent message passing mechanism at the image level. The Smalltalk virtual machines (VM's) are enriched with some primitives that enable inter-VM communication, and the images with special objects that make use of these new primitives in order to send messages to each other over the network. Other attempts use a centralized data base containing the shared objects, where the database manager acts as a virtual machine and communicates with a number of VM's by means of transparent message passing [3]. Although we think the object manager of a VM indeed should support persistent storage and concurrency control, we do not believe in this centralized approach. We believe in a truly distributed system, where each VM provides this functionality. Advantages of the image level approach are:

- ☐ No substantial changes have to be made to the VM, and hence
- ☐ it is relatively easy to make VM's from different vendors work together and
- ☐ no performance is lost during local operation.

However there are a number of problems that are difficult to solve with an image level approach.

One problem mentioned in [2] has to do with I/O. When a remote VM is executing some method for me, and within this method a message is sent to the object Display, I want things happen on my screen and not on the screen of my colleague (who in his turn would be rather annoyed seeing my menus pop up out of the blue).

Then there is a problem with standard classes on different hosts. Is the semantics of "standard" classes everywhere the same? If not, things can go very wrong. On the other hand it would be almost impossible to keep consistency unless every user is so kind not to touch standard classes.

A third problem has to do with Smalltalk processes. With the image level approach, during a remote execution there are several Smalltalk processes involved during a remote execution, at least one sender and one receiver process. Therefore the standard Smalltalk debugger can not be used for remote debugging.

To solve these problems we adopted a different approach. We chose for the concept of distribution transparency [4] at the image level. This is the property provided by the VM's, that all consequences of distribution are concealed from the image, and therefore also from applications and users. As a result of distribution transparency, all objects in the system may be referenced in a uniform manner regardless of such factors as access, location, migration and replication. Since distribution transparency must be provided by the VM's, we call our approach the "VM level" approach. When a message is sent to some object, the VM knows whether the object is local, remote or replicated, and if it is remote, where or how to find it and if it is replicated, how to select the appropriate replica. If the receiver happens to be a local object, the local VM handles the message just like in an ordinary standalone VM. If the receiver is remote however, the message is forwarded to the remote VM. As a result objects on different VM's can work together as if they were on the same host. If there is distribution transparency, the Smalltalk process concept need not change at all. For remote execution process objects simply migrate to remote VM's, and the context chain may run across many machines. At the image level this is all transparent and therefore NO modification whatsoever is needed to e.g. the standard Smalltalk debugger or the user-interrupt code.

4. *The VM level solution*

In the previous chapter problems encountered when "distributing" Smalltalk are introduced. In this chapter we discuss our VM level solutions for them.

Host objects

Lets start with a number of virtual machines being capable of localizing objects and doing remote sends (and returns) and one standard Smalltalk image. In order to make a distributed Smalltalk system we can take this standard image and distribute its objects at random over the VM's (see fig.2). The resulting system however would still be a single user system, since there is only one Display object, one InputSensor, etc. and worse, they are on different machines. So the first thing we should do is making

extra objects on every VM which represent the functionality of the underlying hardware, a.o. Display, Sensor, Processor, ScheduledController. We call these objects "host objects" (a host is a VM + image). Since we may now have multiple objects associated with the same name (e.g. the name Display is associated with a set of display objects, one on every host), we immediately have to deal with a new problem. How do we select the right object? In case of a message sent to Processor or

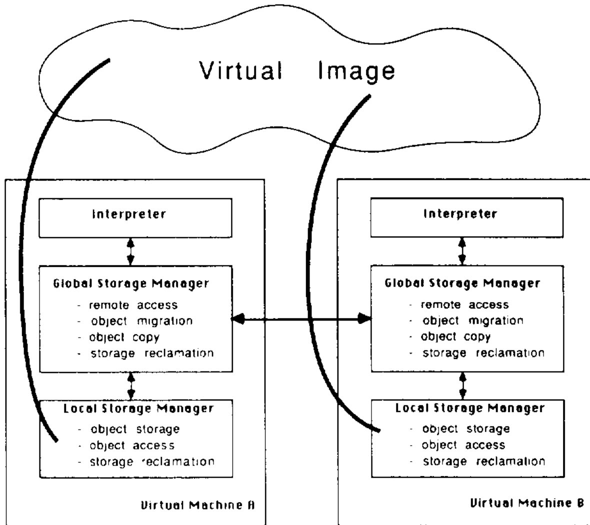


Figure 2. The DistributedSmalltalk system with two virtual machines

ScheduledControllers the local object is fine. However when e.g. Display is the receiver, the VM handling the message must find out on behalf of which user (me or my colleague?) the message is sent, and redirect it to the host of this user.

Replication

The system now can handle multiple users concurrently, but is extremely slower than its local standalone counterpart. This is because even for simple key strokes or mousebutton clicks the objects needed to handle them are scattered at random over the network. Also the method lookup process itself is very costly, as class objects, their superclasses and their method objects can reside on different hosts. To reduce the network traffic, we practically do the same as we did with the previous problem. We introduce multiple copies (replica's) of heavily used objects, one for each host.

Selection of message receivers

The receiver selection mechanism for host objects like Display is implemented in the following manner. Host objects like Display carry a flag "HOME" in their header, and Process objects have an instance variable containing the identity of the host where their process was initiated (their "home"). When a receiver is flagged "HOME" then the message sent to it is forwarded to the home of the current process. Objects that are flagged "HOME" are called "home objects".

The selection problem for replicated objects is not that urgent, in fact it seems obvious to always select the local replica in order to avoid network traffic (after all that is why we replicated in the first place), but on second thoughts it becomes clear that class objects the instances of which are of a highly interactive nature, be better selected on the host where the interaction is to take place (StandardSystemView, Debugger, Inspector, etc.) Therefore also these objects are flagged "HOME" and the same selection mechanism is used as for host objects like Display.

Relationship between host-, home- and replicated objects

The host objects are

- ☐ Display, Sensor, Transcript.
- ☐ All class variables.
- ☐ Processor, ScheduledControllers.

Before we define the relation between host- and replicated objects, we have to introduce the concept "equality".

- ☐ Two objects are equal if they reference exactly the same objects in the same order. That is if the objects are viewed as tuples consisting of references $r1,..,rn$ and $s1,..,sm$ then $n = m$ and $r1 = s1, .. , rn = sn$.
- ☐ Two objects are equal if the objects they reference are equal.

An object's graph is a directed graph where the nodes are all objects that can be accessed from the root object via a path of inter-object references (the graph's branches). Two objects are called deep equal if their graphs do not share any node, other than primitive self-describing objects like SmallIntegers (Objects containing the SmallInteger "3" in this context are considered to share the object "3"). Two objects are called shallow equal if their graphs share all nodes except the roots. If the roots are shared the objects are called identical. See also [5].

A set of objects is replicated if

- ☐ there is exactly one object on each host and
- ☐ the objects are equal except for host object nodes in their resp. graphs.

From this definition follows that host objects, although not equal, are replicated. Replicated objects are

- ☐ Smalltalk (the SystemDictionary containing all global variables).
- ☐ All host objects.
- ☐ All objects that are referenced by a non-host replicated object and that are not shared.

Non-replicated objects are all remaining objects, that is objects not referenced by a non-host replicated object.

The home objects are

- ☐ Display, Sensor and Transcript (we already explained why).
- ☐ All class variables (because they often serve as a communication medium for class' instances of one user, e.g. ParagraphEditor *CurrentSelection* and *Clipboard*).
- ☐ Class objects the instances of which are of a highly interactive nature.

Restrictions on object location

To make method lookup faster (and simpler) the VM's make sure (a replica of) the class object of methods and class-instances is locally present. These restrictions are reasonable, since before a class' instance or method can be accessed, nearly always the class object itself must be accessed first. At the moment also (a replica of) the superclass object of each class object must be locally present. This restriction will be removed.

Easy implementation

A way to understand why the replication problem and the I/O problem are best handled at the VM level is to realize that within the VM there are only a few places where a check is needed. During send-bytecode interpretation one check is needed to see if the receiver is a home object and if the message should be forwarded (and this check is combined with the check for remote receivers, so there is no additional cost). In the same way there is only one place in the VM code, where a check has to be done on writing in a replicated object (this is also combined with other checks). With the image level approach checks would be necessary on numerous places. Furthermore replication by the VM can be easily implemented as an atomic action.

5. *Addressing and storage management*

Every VM manages its local object space, which is organized the same way as the Berkeley Smalltalk object space. In DistributedSmalltalk objects within a local object space are uniquely identified and addressable by means of their *oop* (object-oriented pointer). Most of the time objects are addressed

directly, and oops are pointers to the object (instead of indices in an object table).

Forwarding objects

Sometimes objects are addressed indirectly. In this case the oop is a pointer to a "forwarding object", a Smalltalk object that contains the actual pointer. Forwarding objects are not visible from within the Smalltalk image, but for the VM they are ordinary objects. A forwarding object is used as a temporary indirection to some local object that moved to another address (and therefore changed its oop) during garbage collection or because it needed more space. In this case the forwarding object is located at the object's old address.
A forwarding object is also used to reference an object on a remote host. This kind of forwarding object is also called "proxy" [16]. A proxy contains a host identification and an index in a table on this host. This table contains the local oops of remotely referenced objects. When remotely referenced objects change oop (because of garbage collection or growing), only their table entry has to be updated.
When a forwarding object is accessed, the VM recognizes this by means of a "FORWARDING" flag in the object header and accesses the "forwarded" object instead. In case of local forwarding, the reference to the forwarding object is shortcircuited to the forwarded object.

Object spaces

A VM's object space consists of several subspaces. One subspace contains all replicated objects (ReplicaSpace), and the others contain objects according to their age (OldSpace, NewSpace, and SurvivorSpace). SmallIntegers are self-describing, their oop is at the same time their contents. All other objects have real oops that point into one of the before mentioned spaces.
Newly created objects reside in NewSpace, old objects in OldSpace. SurvivorSpace is an intermediate between New and OldSpace.

ReplicaSpace

The ReplicaSpace is the only space in DistributedSmalltalk, that does not exist in Berkeley Smalltalk. The ReplicaSpace is like OldSpace, except that it contains equal objects in the same order on every host. Since replicated objects are in a separate space, they can be recognized by their oop. Since ReplicaSpaces start at the same local address, a replicated object has the same oop on every host. For this reason remote procedure call (de)serialization routines need not translate pointers to replicated objects.
The image can be seen as a directed graph, with the Smalltalk SystemDictionary as root. ReplicaSpace is a subgraph which covers the top of the image graph. The leafs of this subgraph are either primitive data like SmallIntegers, or oops to objects in other local spaces. Host objects contain oops to local objects, that are not shared by other hosts. The remaining leafs of the graph are objects shared by all hosts. Take for example some non-replicated class object, then the Smalltalk SystemDictionary of the

host where this class object resides, contains an association (which is a replicated object), the value part of which is an oop to the class object in some other local space. All other hosts have the same association except for the value part which is an oop to a proxy, which points to the same class object.
The leafs of the ReplicaSpace are the roots of the other spaces.

Replication consistency control

In order to keep the ReplicaSpaces mutually consistent, every VM checks whether a store in an object affects this consistency. In case of store operations in non-host replicated objects, the modification is replicated over all other hosts by means of a broadcast message. Concurrency control is done by optimistic time-stamp ordering using Thomas' Write Rule (TWR) [6]. Creation of new replicated objects or existing ones that change oop (because they must grow) is handled in much the same way. The difference here is that no rejections are possible. Where according to TWR a modification request (for a store in a replicated object) comes "too late" and therefore is rejected, in the latter case the new object (or the object to move) simply has to be inserted between the ones with timestamps just lower and higher.

6. *Garbage collection*

The problem of garbage collection is that of reclaiming space occupied by "dead" objects, which is data that has become inaccessible. All data (objects) in a heap oriented system form a graph structure of objects pointing to one another. This graph contains some root objects, which are accessible by definition. Objects live when they are accessible via a path of pointers starting from a root. Otherwise they are dead.

Generation Scavenging

In DistributedSmalltalk (and Berkeley Smalltalk) the garbage collection algorithm is based on the lifetime of objects, and is called "Generation Scavenging" [7][8]. Newly created objects are stored in NewSpace. When NewSpace is filled up, NewSpace and SurvivorSpace are garbage collected with a breadth first copying graph traversal called scavenging. The roots of this graph are the set of New and Survivor objects referenced from OldSpace, ReplicaSpace or remote hosts. This root set is dynamically updated by checking on stores of pointers to New spaces. The objects in this graph are moved to a new SurvivorSpace, except for old enough objects, which are moved to OldSpace. At the end of a traversal NewSpace is empty. Since most new objects die soon (e.g. method contexts), OldSpace fills up relatively slowly, and therefore garbage collection of the much bigger OldSpace and ReplicaSpace is necessary much less frequently. This is the reason Generation Scavenging is very cheap compared to more conventional methods like mark&sweep, mark© and reference counting. Because the graph of living new objects is relatively small, scavenging NewSpace can be done atomically without the user noticing an interruption (non-disruptiveness).

Collecting old garbage

In Berkeley Smalltalk the OldSpace is garbage collected on user-request using a conservative mark&sweep algorithm with a file as temporary space. This is called a "reorganization".
For DistributedSmalltalk we designed a concurrent scavenging algorithm. Scavenging is done within a background Smalltalk process, that each time it is active copies a few living OldSpace objects from one side of the OldSpace to the other. The old copies become forwarding objects. Roots of the graph are the Old objects referenced from ReplicaSpace, NewSpace, SurvivorSpace and remote hosts. Objects referenced from ReplicaSpace or remote hosts are dynamically kept in a table.

Collecting distributed garbage

Figure 3 (left) shows 4 hosts containing local (cyclic) garbage, distributed non-cyclic garbage, distributed cyclic garbage and a distributed living structure (objects painted on the edge of a host are roots). Figure 3 (middle and right) shows the same hosts after a few garbage collections. All figures are snapshots from an interactive simulator called "GarbageEditor" designed to aid in the design and testing of our distributed garbage collection algorithm. It allows you to graphically construct hosts, objects, pointers and it gives visual feedback during the garbage collection of the simulated system.

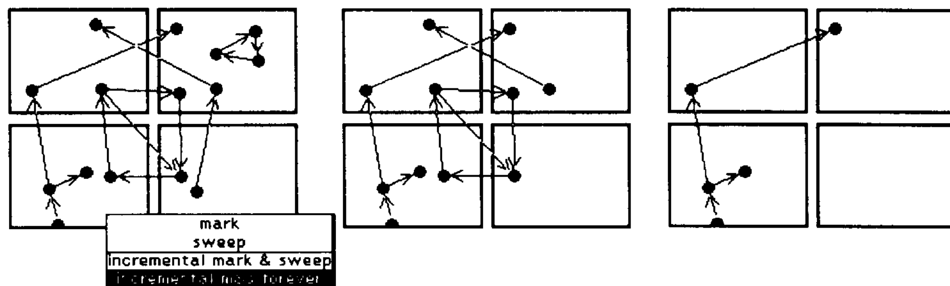


Figure 3. Four simulated hosts before, during and after some garbage collections

In order to discover dead objects in a distributed system, you should check all hosts on having pointers to a particular object. This can be accomplished by a system wide mark and sweep algorithm. The graph of living objects is traversed, the objects accessed are marked, and at the end the space of unmarked objects is reclaimed or "swept". Approaches using reference counting or weighted reference counting already suffer from the deficiency of not being capable to collect local cyclic garbage, let alone distributed cyclic garbage, so these methods are not very interesting. The global mark&sweep algorithm however, although it handles both local and distributed cycles well, does not work properly when not all hosts are able or willing to cooperate, which in an average distributed system is likely to be the case. At our R&D for example the times that all Suns are in the air will be seldom.

Incremental distributed garbage collection

Therefore we looked for an "incremental" algorithm based on a very loose way of cooperation between hosts as described in [9]. Again the problem with this incremental approach was the impossibility to collect distributed cyclic garbage. However we succeeded to solve this problem.
As in [9], garbage collection is an activity which hosts are free to apply to their local spaces whenever they want. During such a garbage collection no remote hosts are involved, only remote references are gathered. Afterwards these references are sent to the appropriate hosts. Unlike in [9], in our algorithm there is some information associated with each reference concerning the graph structure it is part of. It is not necessary that every remote host picks these references up immediately. It is even allowed that they are not received at all, e.g. when the receiving host is currently down (see Host method *sendAccessPathsAndSweepEntrance;* during *receiveAccessPaths:from:time* the exception *HostDown* is raised).

```
markAndSweep
      self unmark; markLocalLiving; sweep; sendAccessPathsAndSweepEntrance

markLocalLiving
      entrance do: [ :dgcInfo | dgcInfo prepareForDGC].
      "traversal with local roots"
      roots do: [ :aRoot |
            aRoot markLocalRecursivelyRoot: aRoot mark: #hasLocalRoot
                  backTrackWhen: [:mark | mark == #hasLocalRoot]
      ].
      "traversal with remote roots"
      entrance do: [ :aDGCInfo |
            aRoot <- aDGCInfo object.
            aRoot markLocalRecursivelyRoot: aRoot mark: aRoot
                  backTrackWhen: [:mark | mark == #hasLocalRoot | mark == aRoot]
      ]

sendAccessPathsAndSweepEntrance
      | received hostsDown |
      hostsDown <- Set new.
      accessPaths keys do: [ :aHost |
            received <- true.
            ([aHost receiveAccessPaths: (accessPaths at: aHost) from: self time: self time]
                  except when: #HostDown
                        do: [received <- false. hostsDown add: aHost]) value.
            self cleanupAccessPathsAt: aHost received: received.
      ].
       "sweep entrance"
      entrance do: [:dgcInfo | dgcInfo collectGarbageWithHostsDown: hostsDown]
```

Figure 4. Some Host methods for garbage collection

The sets of references to local objects that a host has received from remote hosts are used for the next garbage collection. When a set is not up to date this means it may refer to objects that in reality are not referenced anymore. As a result some garbage can be kept alive until some future garbage

collection. In fact, as long as some host is down or inaccessible, distributed garbage part of which is on this host will not be collected. The sets are guaranteed to include every remotely referenced object, so no living objects will be garbage collected as a result of incomplete received information.
Since the information exchange is the only interaction necessary between hosts (apart from some information exchange necessary when pointers move between hosts; we managed however to incorporate this in the remote send operation itself to avoid extra network traffic) and since there are no rules prescribing some time order or any other dependency between the garbage collection activities of different hosts, no synchronization problems had to be tackled.
The simulated hosts collect local garbage using mark & sweep depth-first graph traversal as opposed to DistributedSmalltalk, where the generation scavenging technique as mentioned earlier is based on copying breadth-first graph traversal. When during graph traversal a remote reference is detected (and added to a collection of other references to the same remote host), the algorithm backtracks, so there is no remote access during traversals.
A host's *entrance* is the indirection table via which the host's objects are accessed by remote hosts. Entries in this table contain of course a pointer to the remotely referenced object, but also information for distributed garbage collection purposes (*DGCInfo*). Before a garbage collection graph traversal (see Host method *markLocalLiving*), the information received in the past from other hosts about the structure of the distributed graphs is combined and results in what is called an *AccessPath* for each remotely referenced object. This is done in *prepareForDGC*. The most interesting feature of the AccessPath mechanism is that it leads to the detection of distributed garbage whether containing cycles or not. It is also capable of handling "moving" roots in a living graph (see fig. 5). A more detailed discussion of the AccessPath mechanism is beyond the scope of this paper.

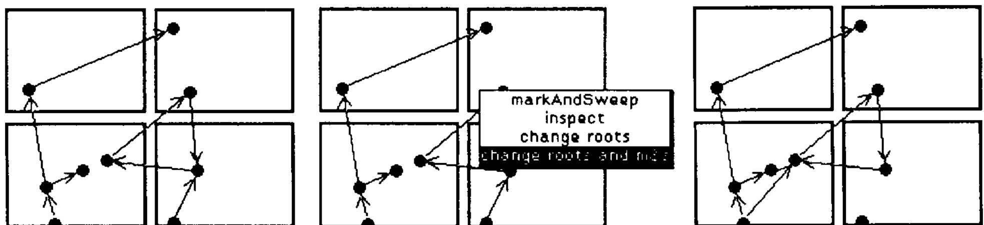


Figure 5. Living distributed graphs may have changing roots

After the AccessPaths are sent to the appropriate remote hosts, the entries in entrance that indicate "*garbage*" are removed by *collectGarbageWithHostsDown:*, that is, only if all remote hosts accessible from a particular entry received the AccessPath associated with that entry. The local objects accessible only from these removed entries will be collected the next time. Although garbage entries are detected already before the traversal (by *prepareForDGC*), they are not removed until after the traversal,

because remote hosts accessible from these entries first should receive the garbage-indicating AccessPath. Absence of this information would only indicate *"not referenced (anymore) from host X"* and it would take more time than necessary for the remaining garbage to be collected (in fact the algorithm would grow a factor N in time complexity, where N is the number of remotely referenced objects in the distributed graph).

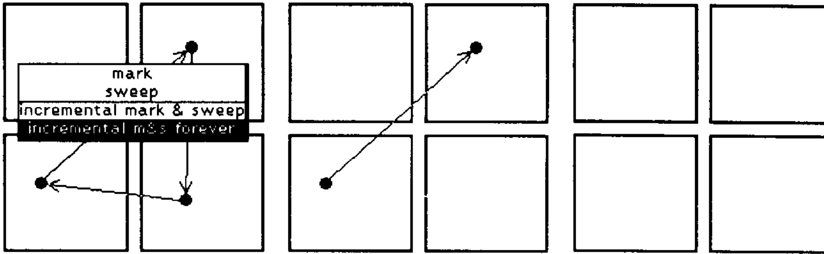


Figure 6. Simple distributed cyclic garbage before and after some garbage collections.

In fig. 6 hosts are numbered 1 to 4 from up-left to down-right. Each time a host has performed a garbage collection, it tries to contact one other host. Host 4 sends an AccessPath to host 3, host 3 to host 2 and host 2 to host 4. When there has been a sequence of garbage collections with successful sends such that a particular host (host 4 in this example) receives back the information it sent out in the past (after being processed by hosts 3 and 2 in this example) then this host will conclude that his object is not accessible anymore via the entrance. After the next successful send of the associated AccessPath, the object's entry in entrance is removed, including the last local pointer to the object. Hence the object's space will be reclaimed at the next garbage collection. All cyclic garbage will have been collected after a sequence of garbage collections on hosts *432443322* but also after e.g. *14u322142u14u32u1122u423u12321411213* ("4u" indicates unsuccessful send after garbage collection on host 4).

In fig. 7 the up-left host has three remotely referenced objects. Associated with each of the three remote pointers from this host is a set of three AccessPaths, since there exist local paths from each of the three remotely referenced objects to these pointers. In order to determine these different AccessPaths, some objects are visited more than once during graph traversal (see backtrack criterion for remote roots in *markLocalLiving*). In general the number of those objects is relatively small, so no performance degradation occurs.

As the number of remote references in an average distributed system is a small fraction of the total ("locality of reference"), the (work involved with) distributed garbage is a fraction of the (work involved with) local garbage. Therefore we could allow ourselves the luxury of implementing the major part of the algorithm at the image level (in relatively slow Smalltalk-80). The bulk of the work (collecting local garbage and remote pointers) is done in the virtual machine (written in C).

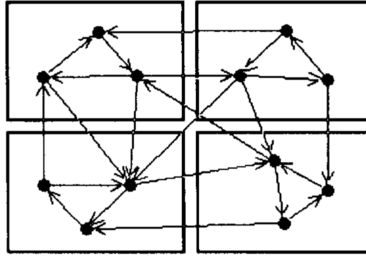


Figure 7. More complex distributed cyclic garbage

7. *Message passing*

Smalltalk objects communicate with each other by passing messages. A Smalltalk message can be considered to be a dynamically bound procedure call; the particular method to be bound is determined by the (super)classes of the receiver. In DistributedSmalltalk message passing is location transparent, i.e. any message may be sent to an object residing on the same or different host in a uniform way.

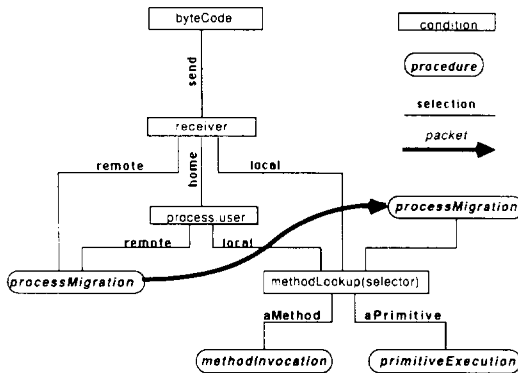
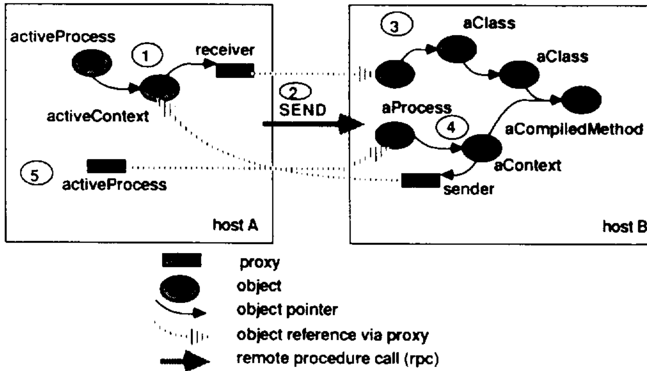


Figure 8. Send-bytecode interpretation

On a message send, if the receiver is a proxy, the host forwards the send message to the host of the remote object, that is represented by the receiving proxy. If the receiver is a home object and the host identity of the active process is not the same as the identity of the interpreting host, the send message is forwarded to the host of the active process (its home).

A send message is forwarded by means of a remote procedure call (rpc). A send rpc transfers the following objects: active process, active context, message selector, receiver and arguments. At the server

host, message lookup is performed, a new context is created and the transferred process is resumed. The new context's sender is a proxy of the active context at the client host, and the resumed process has a proxy of its home process. At the client host, the active process is suspended.



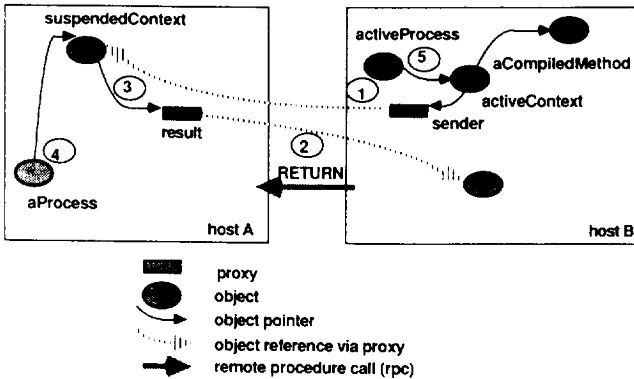
(1) if receiver is proxy (or receiver is home object ..),
  (2) do SEND(activeProcess, activeContext, send message attributes) rpc
  (3) look up compiledMethod (via class-superclass chain)
  (4) resume transferred process with new context
(5) suspend active process at client (host A), and create proxy of migrated process

Figure 9. A remote send

On a message return, if the sender of the active context is a proxy, a return rpc is initiated. A return rpc transfers the following parameters: active process, active context's sender and the result. At the server, the result is put on the stack of the suspended context. The transferred process is resumed. At the client, the active process is suspended. The same mechanism is used for remote block executions, because a block is also an object. If a block is sent as an argument to a remote receiver, a proxy of the block is transferred to the receiver. A remote block execution is in fact the result of a message send to a proxy of a block.

Smalltalk does non-preemptive scheduling for processes of the same priority. When a process migrates to a remote host, it must share the processor with other processes. In order to avoid situations where a process does not give up the processor voluntarily, periodically a Smalltalk time slicer process with adequate priority gets hold of the processor.

A process can hop through various hosts before returning to its home. It should be possible to interrupt a migrated process from its home. Therefore, after each hop (a send rpc or return rpc) the home proxy of the migrated process is updated (via an update rpc). For each particular process hopping through various hosts, there is a race condition between update rpc's. In order to solve this problem, each update rpc and each home proxy is assigned a unique timestamp according to the causal order of events. An update is performed only if the home proxy timestamp is older than the update timestamp.



(1) if sender of active context is proxy,
(2) do RETURN(activeProcess, sender, result) rpc
    (3) put result on stack of suspendedContext
    (4) resume transferred process with suspendedContext
(5) suspend activeProcess at client (host B)

Figure 10. A remote return

During message passing, various rpc exceptions e.g. "host not available" can occur due to the distributed nature of the system. If an rpc exception occurs during a send or return, the process user will be notified by means of an exception message to a home object. If the process home is not reachable, the process is considered to be an orphan and terminated.
The default exception handling in Smalltalk opens a notifier view to the user (by means of the methods *doesNotUnderstand:* and *error:*). The notifier offers the user the opportunity to proceed or to debug. This type of exception handling is appropriate for exceptions like "message not understood" which are under control of the user. But rpc exceptions have a transient nature, and therefore we have implemented a general exception handling control structure in Smalltalk (see fig.4).

Argument access (in primitives) is read-only. When a primitive argument is a proxy, a copy is made by means of a copy rpc. Most primitives, except for BitBlt, are implemented according to the rules of data encapsulation, i.e. only the receiver is directly manipulated, not the objects it refers to nor the primitive arguments. BitBlt primitives directly access the receiver's instance variables. Therefore, when an instance variable of BitBlt is a proxy, a copy is made also.

8. *Communication between hosts*

Hosts communicate with each other by means of rpc. We use two forms of rpc: a synchronous unicast and a broadcast. The synchronous unicast is similar to a local procedure call, i.e. there is one caller and one callee, and during the call the caller is blocked. A broadcast has one caller and several

callees. The rpc's are used for:

- ☐ Remote message passing (send, return, update).
- ☐ Copying read only objects (shallow copy, deep copy).
- ☐ Initializing a host. Every new host broadcasts its network address. The receiving hosts add this address and the associated host identity to a table and return this table to the new host, which after receiving the first reply ignores the others.
- ☐ Replication consistency control.
- ☐ Distributed garbage collection.

The rpc parameters that are transferred are simple C integers or graph structures representing Smalltalk objects pointing to each other. Given some Smalltalk object the following can be transferred:

- ☐ A reference to the object. If the object is replicated, just the object's oop is transferred, otherwise a proxy object.
- ☐ A shallow copy of the object. Only the references of the objects it points to are transferred.
- ☐ A deep copy of the object. The entire graph with the object as root is copied.
- ☐ An n-deep copy of the object. The graph with the object as root is copied until depth n.

Most of the time only references are transferred. For performance reasons heavily used read only objects are transferred as deep copy (e.g. Strings, Rectangles and Points). Shallow copies are made for arrays, e.g. the argument array in the perform primitive.
In order to transfer Smalltalk objects across machine boundaries, it is necessary to serialize them into a sequence of bytes suitable for network transmission. The inverse operation, deserialization, converts the object back into an accessible form in a particular host address space. We have implemented a general (de)serialization procedure, that can transfer a graph of Smalltalk objects until any depth. Cycles in this graph are handled appropriately.
We use the Sun rpc library in our implementation [10]. However we had to change the Sun rpc polling mechanism in order to avoid communication deadlock, which in our application is possible because a host can be rpc client and server at the same time. When for example host A issued an rpc request to host B and is waiting for the reply, whereas B issued an rpc to A and is also waiting for the reply, we have a deadlock. Communication deadlock can be detected by polling for incoming replies and service requests simultaneously. Deadlock is broken by serving service requests first and getting the reply later.

9. *Related work*

Our work is related to [1] [2] [16]. While in [1] [2] the mechanisms are incorporated in the virtual image, we have chosen for the virtual machine level approach, in order to provide distribution transparency for image applications as well. Because of this approach, no modifications had to be made to

existing image applications such as the debugger. As in [1] class objects can be replicated, but we have chosen for replication control to keep replicated objects consistent, instead of making run-time class compatibility checks, when instances of replicated classes have to move between machines. The I/O problem mentioned in [2] is solved with the "home object" mechanism. Our remote message passing mechanism is similar to Emerald [17]: a process moves to the destination host of the receiver of the message in order to execute the method. The problem of reclaiming cyclic distributed garbage is mentioned but not solved by [2] [16] while in [1] an expensive system-wide mark&sweep algorithm is implemented where obviously the cooperation of all machines is necessary. Instead our distributed garbage collection scheme reclaims cyclic distributed structures incrementally with a minimum of inter host communication.

1. *Current status and benchmarks*

Without any changes to existing programming tools, we now can browse on remote classes, interrupt and debug remote processes, and inspect remote objects. We promoted user interface class objects (e.g. views, controllers) to home objects in order to get an acceptable performance. Thus remote message passing is not necessary for I/O e.g. sensing input, displaying windows. For example, inspecting a remote class object, involves remote message passing for getting the model's printable representation (Strings), but not for displaying activities (menus, window clipping, text scanning). We have succeeded in keeping the system responsive.
Our incremental distributed garbage collection algorithm is implemented and tested by means of simulation, and is now being integrated in DistributedSmalltalk.
Table 1 shows the results of the micro benchmarks measured for two Sun-3/160 workstations. A comparison is made between a local send and a remote send with different types of results: self, a string, an array. A remote send is approximately 500 times slower than a local send.

Return Value	Local Send	Remote Send	(in milliseconds)
self	0.09	28	
literal	0.26	144	
array	0.20	106	

Table 1. Micro benchmarks

Table 2 shows standard benchmarks for BerkeleySmalltalk, DistributedSmalltalk with one host and DistributedSmalltalk with two hosts. In the latter case, some benchmarks (PrintDefinition, PrintHierarchy, Inspect, Compiler and Decompiler) apply to remote class objects. According to these benchmarks, a remote programming activity is 45% slower than a local one. As can be seen from table 2, additions

made to BerkeleySmalltalk for testing on each message send, has made local computing about 10% slower.

	DistributedSmalltalk	DistributedSmalltalk*	BerkeleySmalltalk
BitBLT	26.6000	28.0986	27.7083
TextScanning	39.0000	36.2791	35.4545
ClassOrganizer	19.3291	15.1081	19.6429
PrintDefinition	18.1624	6.05413*	27.2436
PrintHierachy	19.6850	9.39850*	24.3902
AllCallsOn	17.0455	12.6404	21.2766
AllImplementors	21.1039	16.1692	24.2537
Inspect	31.1644	2.33813*	35.0000
Compiler	23.9785	9.65368*	27.5990
Decompiler	18.3976	1.70893*	21.2329
KeyboardLookAhead	36.3248	30.5755	40.4762
KeyboardSingle	39.9267	34.2767	44.1296
TextDisplay	28.5714	23.1884	25.1968
TextFormatting	37.8289	36.3924	42.5926
TextEditing	39.7363	31.8731	45.0855
Performance Rating	26.4813	14.2129	29.5995

(*) applied on remote class objects

Table 2. Macro benchmarks

11. *Conclusions*

By providing a remote message passing mechanism at the VM level, we have achieved distribution transparency for users and image applications. Therefore, no changes had to be made to existing image code, not even to the debugger. Transparent I/O is provided by means of a new concept called "home objects". Performance degradation is acceptable by means of replication and furthermore by making user interface classes like views and controllers, home objects. Local programming activity is 10% slower than in a comparable standalone system. Although remote sends are 500 times costlier than local sends, according to our macro benchmarks, remote programming activity is only 45% slower. Replication is transparent and replication consistency is guaranteed, so for replicated class objects no compatibility checking is needed. Distributed garbage, whether containing cycles or not, is collected incrementally without any synchronization being necessary.

12. *References*

[1] Bennett, J. K., "The Design and Implementation of Distributed Smalltalk", OOPSLA '87 Proceedings, Oct. 4-8, 1987, pp 318-330

[2] McCullough P. L., "Transparent Forwarding: First Steps", OOPSLA '87 Proceedings, Oct. 4-8, 1987, pp 331-341

[3] Purdy A. et all, "Integrating an Object Server with Other Worlds", ACM Trans. Office Information Systems, Vol. 5, No. 1, January, 1987, pp 27-47

[4] Herbert A. J. and Monk J. (editors), ANSA Reference Manual Release 00.03 (Draft), June 1987

[5] Khoshafian S. N. and Copeland G. P., "Object Identity", OOPSLA '86 Proceedings, pp 406-416

[6] Thomas R. H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases", ACM Trans. Database Systems, Vol. 4, No. 2, June 1979, pp 180-209

[7] Ungar D., "Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm", ACM Software Engineering Notes, April 1984, pp 157-167

[8] Lieberman H. and Hewitt C., "A real-Time Garbage Collector Based on the Lifetimes of Objects", Communications of the ACM, Vol. 26, No. 2, June 1983, pp 419-429

[9] Mohamed Ali K. A-H, "Object-oriented Storage Management and Garbage Collection in Distributed Processing Systems", The Royal Institute of Technology, Dept. of Telecommunication Systems - Computer Systems, Sweden, Report TRITA-CS-8406, December 1984

[10] Remote Procedure Call Reference Manual, Sun Microsystems, Inc., Part Number 800-1177-01, Nov. 1984

[11] D.R. Cheriton, "The V Kernel: A Software Base for Distributed Systems", IEEE Software 1, April 1984, pp. 19-42.

[12] M. Accetta et al, "Mach: A New Kernel Foundation for UNIX Development", Proc. Summer 1986 USENIX Technical Conference and Exhibition, June 1986.

[13] B. Liskov, "Overview of the Argus Language and System", Programming Methodology Group Memo 40, M.I.T., Laboratory for Computer Science, February 1984.

[14] P. America, "Rationale for the design of POOL", ESPRIT Project 415, Doc. Nr. 0053.

[15] A. Goldberg and D. Robson, "Smalltalk-80: The Language and its Implementation", Addison Wesley, 1983.

[16] D. Decouchant, "Design of a Distributed Object Manager for the Smalltalk-80 System", OOPSLA '86 Proceedings, pp. 444-452

[17] E. Jul et al, "Fine-Grained Mobility in the Emerald System" ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988, pp 109-133