Implementation of Distributed Trellis

Bruno Achauer

Computer Science Department, Telecooperation Group
University of Karlsruhe, D-76128 Karlsruhe, Germany

Abstract. DOWL is an extension of the Trellis language supporting distribution. It allows programmers to transparently invoke operations on remote objects and to move objects between the nodes of a distributed system. A few primitives permit the programmer to take full advantage of distribution and to tune performance; most notably by restricting the mobility of objects and specifying which objects should move together. This paper describes the implementation of these extensions: the object format, communication system and the mechanism to invoke operations on remote objects. Performance figures are also presented.

1 Introduction

Object-oriented systems are well-suited for distributed processing: in the object-oriented paradigm, flow of both control and data is performed by sending messages between objects. Provided that there is an appropriate addressing mechanism, these messages can be sent over a network; thus, operations on objects can be invoked regardless of their actual location. Moreover, location-independent invocation allow movement of objects at run time, providing opportunities for dynamic reconfiguration and load adaption.

In contrast to remote procedure call (RPC), the distributed object-oriented paradigm is not bound to client/server architectures and provides full syntactical and semantical equivalence between local and remote operation invocation: in both cases, arguments can be passed as object references.

However, complete distribution transparency is not desirable. For several reasons, there have to be mechanisms to make distribution visible. First, distributing an application obviously requires means for object placement. Second, even though there is no semantic difference between local and remote invocation, there are differences in performance. A mechanism to keep related objects closely together is therefore essential to keep the communication overhead tolerable. A related topic is parameter passing. Depending on the intended usage of arguments passed to an operation on a remote object, it might be beneficial to migrate arguments to the object instead of simply passing object references. Finally, there is a need to restrict the mobility of some objects, either because they are communicating heavily with entities outside the application or because they are bound to local resources.

This paper describes the implementation of DOWL, an extension of the Trellis language [12] to support transparent distribution. Rather than inventing a

new language, we wanted to extend a strongly typed, commercially available language to support distributed applications. One of our primary goals was upward compatibility; existing Trellis applications should run on the extended system without any changes.

Specifically, we have chosen to support fine-grained distribution because coarse-grained distribution models (e.g. by clustering objects at the language level) tend to destroy the uniform object model. Moreover, experience has shown that coarse-grained distribution usually introduces an additional abstraction level, which can create problems for the application programmer. For example, in the Argus system [10] there are two different object models, guardians and clusters, with different semantics: Clusters are data types living entirely within a guardian; they are accessed using local procedure calls. Guardians are similar to virtual nodes; they are accessed by remote procedure calls (with different parameter passing semantics). While building a distributed application based on Argus, Greif and others noticed that the semantic differences in this model can force the programmer to use a guardian where a cluster might be more appropriate [8].

The Trellis extensions to support distribution are: [1]

- An addressing mechanism allowing object references to span node boundaries and to invoke operations on remote objects. This is invisible at the language level; the only difference between a local and a remote invocation is a difference in performance.
- The (logical) nodes of the distributed system are represented by **`$Node`** objects. Besides containing information peculiar to the runtime system, they can also answer inquiries about the topology of the distributed system.
- An object's **`$location`** component contains the object's current location. Its initial value is the node on which the object was created, and it changes whenever the object moves.

  Assignment to the **`$location`** component causes the object to move (*migrate*) to the assigned object's location; thus, two objects can be brought together by simply specifying the first object's location to be the second object. Migration is possible even if the moving object currently has one of its operations activated; in this case, the operation is suspended and the operation's context moves along with the object to the target location where it is resumed. Finally, operation results are transmitted back to the caller of the operation.
- To prevent migration of an object, it can be *fixed* at a node. There are two mechanisms available: instances of a type declared with the **`$fixed`** attribute cannot migrate at all; their location is always the node on which they were created. Any attempt to unfix them will result in an error. The inverse attribute, **`$mobile`** states that instances of this type are able to migrate (this is the default).

  An object's **`$fixed_at`** component contains the node at which the object is fixed or the constant **`Nil`** if the object is able to migrate. For instances of a fixed type, this component cannot be altered. For others, assigning a node

to it atomically unfixes the object and migrates it to the specified location where it becomes fixed again. Assigning `Nil` unfixes the object.

- Although both local and remote operation invocation have identical semantics, they have different performance: Using today's technology, a local operation is several orders of magnitude faster than a remote one. It is therefore crucial to keep related objects closely together to reduce communication costs. DOWL provides two mechanisms to achieve this: explicit migration requests using the **`$location`** components, and attaching objects.
  To specify which objects should move together, objects can be *attached* [9] to other objects. Whenever an object migrates, all mobile objects attached to it will also move. Attachment is transitive; any object attached to a moving object will move also. However, attachment is not symmetric; an attached object can migrate without affecting any objects to which it is attached.
- Normally, arguments to an operation are passed as simple references to objects. In addition, DOWL offers two alternative mechanisms: *call-by-move* and *call-by-visit* [3], causing argument objects to migrate either permanently or temporarily to the location where invocation takes place.

Finally, there are two low-level primitives allowing which allow the programmer to violate distribution transparency: the **`$replicated`** type attribute forces instances of a (mutable) type to be replicated, and the **`$local_operation`** attribute requests an operation to be performed on the local representative regardless of the real object's actual location. These attributes are DOWL's equivalent to the **`asm`** statement in C; a programmer should use them only if he knows exactly what he is doing.

Prototypes of DOWL run on DEC's VAX and MIPS architecture workstations, and experience with the Trellis Programming environment (a large, non-distributed application [5]) suggests that we have achieved a high degree compatibility: Adapting the programming environment to DOWL required only minor changes to existing code (most of them were necessary to tell the compiler about the special treatment required for builtin types). In particular, there was only a single operation which required call-by-move.

The rest of this paper describes how these extensions are implemented: The next section presents DOWL's object format and the mechanisms supporting object references across node boundaries. Communication in DOWL is achieved by sending objects between the nodes; section 3 details the mapping of this communication model to facilities provided by the supporting operating system. Section 4 shows how remote object invocation and the parameter passing mechanisms work. Section 5 gives performance information. We conclude with a comparison with related work.

2 Object Format

Trellis distinguishes between constant and mutable objects. Constant objects cannot change their state once they are created and thus can be distributed by

simple replication. In contrast, mutable objects can change during their lifetime. These objects can be used to share data; it is therefore important that only a single copy of a mutable object exists, to ensure that changes to the object made on one node are seen by all nodes. DOWL ensures this by distributing mutable objects as *proxies* [2, 7, 11]. A proxy is a (local) representative for some object residing on a remote node; it behaves exactly like the object it represents: invoking an operation on it is trapped by the runtime system and forwarded to the real object, which carries out the computation. Any results returned or exceptions raised are then transmitted back and delivered to the (local) caller.

Internally, an object is represented by a pointer to an object header, followed by several slots containing the object's instance variables (cf. figure 1). There are two kinds of slots: *Refs* are references to other objects, and *bytes* hold any kind of data that is not to be interpreted as an object reference (e.g. the individual characters of a string).

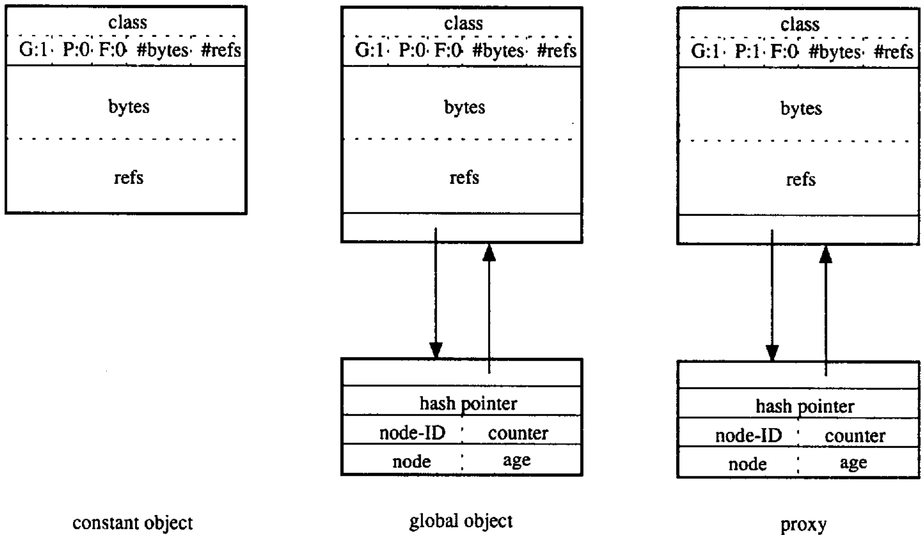

Fig. 1. DOWL object formats.

The object header consists of several subfields describing the object's type and its precise memory layout (i.e. the number of byte and ref slots present). Three flag bits contain information related to distribution:

- The *G-bit* distinguishes *global* objects (which can be referenced across node boundaries) from *local* objects (which are distributed by replication). Its value is determined by the object's type attributes; it never changes.

The other bits are significant for global objects only:

- The *P-bit* indicates whether the structure represents a resident object or a proxy; it is set and cleared as the object enters and leaves the node.

- The *F-bit* determines whether the object is currently fixed. For instances of a fixed type, its value cannot change; for others, it is set and cleared when the object is fixed and unfixed.

Finally, the slot part of a global object is followed by a pointer to a *global object descriptor*, a structure containing information required to implement inter-node references:

- The *global object identifier (GOID)* uniquely identifies the object. It consists of two parts: an identifier for the node that created the object and the value of a counter which is incremented whenever a new GOID is fabricated.
- A *forwarding address* [9] indicates the location of non-resident objects. It consists of two parts: an identifier for the node on which the object is believed to reside, and a counter showing how often the object has migrated so far. The counter is used to disambiguate conflicting forwarding addresses; a greater value indicates more recent location information.
- Each node maintains a hash table mapping GOIDs to object descriptors. A *hash pointer* chains all descriptors hashing to the same table slot.
- Finally, an *object pointer* points to the structure allocated for the object.

Descriptors only exist for proxies and resident objects that are referenced from remote nodes; all other objects may have a `NULL` descriptor pointer, meaning that no external references to the object exist. The descriptor is created only when the object or a reference to it leaves the node for the first time. On the other hand, object structures are only needed when the object is actually referenced; either locally or remotely through its descriptor. There is no need to retain the object for proxies that are not referenced locally; it is sufficient to have the descriptor which can forward all incoming requests. Eventually, the address information on nodes referring to these lone descriptors will be updated, and the descriptor can be reclaimed by the garbage collector.

The real object denoted by a proxy is located by following a chain of forwarding addresses until the object is found. This process is explained best by an example (cf. figure 2):

An object originally created on node A migrated to node B and then finally to node C. Both A and C still have local references to the object, but there are no more references to it on B and the storage allocated for the object structure has already been reclaimed by the garbage collector. Accessing the object from A now proceeds as follows:

1. The P-bit is set, identifying the object structure as a proxy. Consulting the object descriptor gives the object's global identifier (`A-4711`) and the information to look on node B for it.
2. B's descriptor table contains an entry for object `A-4711`, but there is no associated object structure (the object pointer is `NULL`). This identifies the descriptor as part of another proxy. (If the structure were still present, the proxy nature would be revealed by a set P-bit). Since forwarding addresses are consistent (`B-0` vs. `C-1`), the search continues on node C.

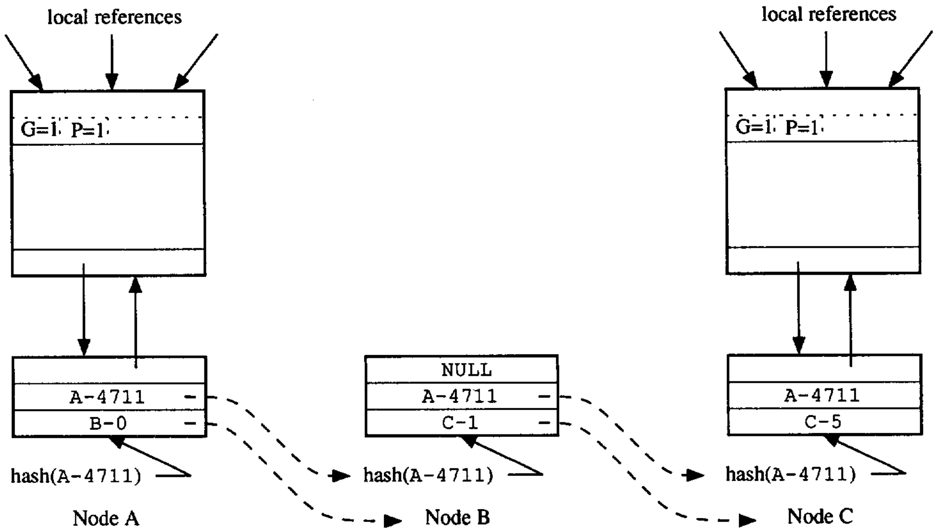

Fig. 2. Forwarding addresses.

3. Now, C has both a descriptor and a structure for object `A-4711`. Moreover, the structure's P-bit is clear, indicating that the object actually resides on C. Thus, the desired access can be performed and C sends the result directly to the original requester, A.
4. A receives both the result and a new location for the object which is more recent than its current address (`C-5` vs. `B-0`). Consequently, it updates the object's descriptor to point directly to node C. If updating the descriptor destroys the last reference to the descriptor on B, it will eventually be reclaimed by garbage collection.

The primary drawback of this object structure is the extra space required for proxy structures; each proxy must provide enough space to accommodate the real object just in case the real object should migrate to a node which has a proxy. This is not a severe problem since Trellis objects are rather small (the average object size is less than 40 bytes); moreover, for permanently fixed objects even this space can be saved because these objects will never migrate and thus can never replace one of their proxies.

On the other hand, representing both proxies and resident objects by the same structure increases efficiency; no indirection is required to access a slot of an object and there is also no need to convert between pointers to proxies and pointers to resident objects or to find and update any pointers. Furthermore, both local and remote objects are implemented by the same object layout, which allows the compiler and runtime support routines to deal with only a single kind of object structures.

A final advantage is the separation of object structures (manipulated by compiler-generated code and a few runtime support routines) and object de-

scriptors (maintained entirely by the runtime system); this allows separation of local garbage collection (reclaiming object structures within a node) from distributed GC (reclaiming object descriptors).

3 Communication

Communication in DOWL is based on message passing. The communication system is implemented as two layers to hide the ultimate transmission method provided by the supporting operating system: a lower layer maintains connections between nodes; it provides functions to send or receive byte streams to or from any node in the system. The upper layer sends objects (more specifically, object graphs) to receivers denoted by arbitrary objects (usually proxies), using the lower layers's functionality for actual transmission. Communication between the layers is achieved by several FIFO queues to allow concurrent operation of both layers.

Besides converting messages between the object graph and byte streams format, the upper layer maintains the object descriptors (notably forwarding addresses and the P- and F-bits) and implements the attachment functionality. It is implemented as two procedures: linearization and delinearization.

Linearization is invoked by sending an object; it traverses the object graph to convert it into a byte stream that will finally be handed to the lower layer for delivery. For each object encountered during the traversal, it decides whether the object has to be included in the byte stream or whether it is sufficient to include the object's GOID and addressing information: Objects to be included are the root object, all local objects and all attached objects which are both resident and mobile. A side effect of the traversal is to allocate and initialize an object descriptor for each global object encountered (unless there is already a descriptor).

Encoding of an object consists of the the object's class, its storage layout and all slot values. Object references are encoded as offsets into the generated byte stream. For global objects, both the GOID and forwarding address from the object descriptor are also included. If the object itself is to be included into the byte stream, the forwarding address in the descriptor is updated to point to the the target node, and the age counter in the passed object is incremented.

A few objects are known to the runtime system and must be present on every node; examples are constants like `True`, `False` or `Nil`, types and methods. These objects are encoded specially and will be mapped to their remote counterparts.

Attached objects are recognized by *attachment maps*. The compiler creates one map for every type and every operation processed. Type maps identify those slots in the object structure containing attached objects, and operation maps show the location of attached variables in the operation's activation record.

Finally, the linearized structure is handed to the lower layer for actual transmission. Additional parameters (determined from the original object's type and from the way linearization was invoked) shows how the message must be dispatched on arrival:

- *Invocation messages* are sent by a thread trying to invoke an operation on a proxy. They are handed to the target node's *distribution server*, which will spawn a new thread to perform the operation.
- Results of remote operation are returned in *result messages*. They are passed to the thread that invoked the operation.
- All other messages are caused by object migration. Depending on how linearization was invoked, the object either becomes fixed or remains mobile.

At the target node, the byte stream is received by the lower layer and handed to the upper layer to reconstruct the object graph (delinearization). A major side-effect of this process is the updating of the descriptor table: in particular, descriptors are allocated for all unknown global objects found in the message and the received forwarding addresses are compared to the data from the descriptor table. If the received information is more recent, the descriptors are updated.

Reconstructing the object graph is done in two passes. The first pass locates all special objects and finds (reusing already existing proxies) or allocates storage for all others. The second pass initializes the slots in the non-special objects.

Finally, the reconstructed object is handed to the designated receiver: invocation messages are put on a queue from where they finally will be picked up by the distribution server. Result messages are handed directly to the thread awaiting completion of its remote invocation. All other messages request object migration; they are completely processed after the object graph is reconstructed.

The lower layer is responsible for reliable transmission of byte streams between nodes, using communication primitives provided by the supporting operating system. Its implementation in the current prototypes is based on TCP (stream sockets), which already provides reliable transmission. The actual implementation is therefore very simple. Every pair of nodes is connected by a bi-directional communication channel, which is operated by two activities on each side:

- A *write server* waits for byte streams to be put on a queue by the upper layer. It removes them and sends them asynchronously over the channel.
- The *read server* waits to receive byte streams from the channel. After receiving a message, it hands it to the upper layer for delinearization and dispatching to its receiver.

Finally, a *connection server* is used to implement system startup: A new node wishing to join the network first queries the connection server of an existing node about the topology of the system. Next, it contacts the connection server of every node, asking it to create the communication channel and to spawn the servers operating the channel. Finally, the new node creates its own server activities to access the channel and joins the system.

All servers are written in Trellis itself, backed up by a few builtin routines to access operating system services. Communication within a node is based on FIFO queues, which are part of the predefined library.

Implementation of the **`$location`** and **`$fixed_at`** components takes advantage of the migration primitives and the fact that invoking an operation on an object takes place at the object's actual location:

```
TYPE_MODULE Object

        COMPONENT ME.$location: OBJECT
        GET IS (Local_Node($Node))
        PUT IS BUILTIN ("Object_PutLocation", "trellis$image");
```

Since both operations are guaranteed to be performed at the object's actual location, the **get** operation is implemented by returning the node on which the operation takes place. Likewise, the **put** operation simply calls an entry point which checks whether the object to be moved is fixed. If so, it returns silently; otherwise, it moves its (resident) argument to the specified location.

$Fixed_at is implemented similarly[1], using a different entry point:

```
TYPE_MODULE Object

        COMPONENT ME.$fixed_at: $Node|Null
        GET IS BUILTIN ("Object_GetFixedAt", "trellis$image")
        PUT SIGNALS (is_fixed)
        PUT IS BUILTIN ("Object_PutFixedAt", "trellis$image");
```

4 Remote Invocation and Parameter Passing

The basic idea to implement operations on remote objects is borrowed from Amber [6]: Whenever there is a chance that an operation's controlling object (**self** in Smalltalk parlance) might be a proxy, check whether the operation operates on a resident object. If so, invocation can proceed; else, the operation is suspended, its context is migrated to the controlling argument's actual location where the operation is resumed. After the operation has returned, its results are sent back to the caller. Another benefit of this scheme is that objects can be moved even if they have one of their methods activated; the invocation context will follow the object as soon as it tries to access the object.

For operations on local objects, the residency check can be omitted because these objects are guaranteed to be always resident. For all others, it must be performed in three situations:

1. when invoking an operation (its controlling argument might be a proxy),
2. when returning from an operation (which might have migrated the controlling argument) and
3. after a context switch (another thread might have moved the controlling argument).

Implementing the check is cheap for the first two cases; on the VAX, it requires only two machine instructions (one of which is hardly ever executed):

```
BBC    #34,me(LP),5          ; is controlling argument a proxy?
JSB    GoOn_Remote           ; yes, trap to the run time system
```

[1] Here, the **get** operation is implemented by a builtin routine to avoid atomicity problems; builtin routines are guaranteed to be never interrupted.

In the current version of Trellis, context switches can occur only at predefined sequence points which are already covered by the other two checks, so the check for the third case can be omitted; it could be implemented by making the interrupt return sequence check for a proxy controlling argument.

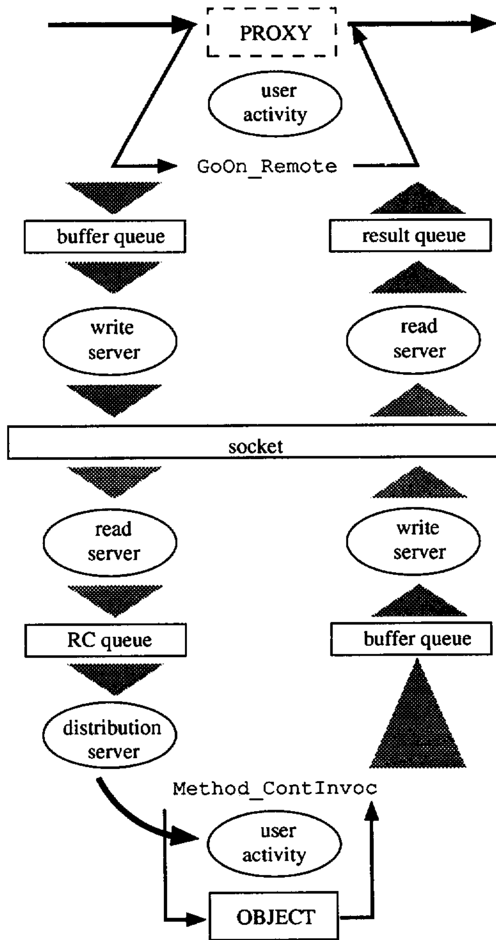

Fig. 3. Remote object invocation.

After a failed residency check, a run time support routine (`GoOn_Remote`) takes control. This routine causes the interrupted operation's context to migrate to its caller before being resumed. Specifically, the following steps are performed (cf. figure 3):

1. `GoOn_Remote` encapsulates the operation context into an invocation message object which is sent to the controlling object. `GoOn_Remote` then waits for a result message to arrive.

The encapsulated context consists of:
- the thread awaiting the result of the remote invocation,
- the name of the interrupted operation,
- the exact position where the operation was interrupted,
- the values of all other registers used by the operation,
- the size and contents of the operation's activation record and
- the activity's global context (e.g. its default I/O streams).

2. On the target node, the byte stream is received, the invocation message is reconstructed and passed to the node's distribution server.
3. The distribution server creates a new thread to resume the interrupted operation. The new thread executes `Method_ContInvoc` as its first routine; its argument is the operation context from the invocation message.
4. `Method_ContInvoc` restores the context of the suspended operation from the passed context: First, it allocates stack space to accommodate the operations activation record and initializes it from the original context. Next, it restores the values of all registers in use. Finally, the suspended operation is resumed by jumping to the residency check whose failure triggered the remote invocation.
   When restoring the stack frame, `Method_ContInvoc` pushed the address of the instruction after the jump instead of the original return address. This way, the stacks looks as if `Method_ContInvoc` had called the suspended operation and it will regain control when the operation finally returns.
   Finally, `Method_ContInvoc` encapsulates the operation's result and the global context into a result message which is sent back to the original thread. At this point, `Method_ContInvoc` returns, causing its thread to terminate.
5. When the result message finally arrives, the original thread is resumed. It pops the interrupted operation's frame (which completed on the remote node), extracts its result from the message and returns it to the operation's caller.

The `$replicated` type attribute is implemented by suppressing the residency checks for all operations owned by this type. `$Local_operation` suppresses the check for a single operation only.

Call-by-move and call-by-visit are implemented by marking the associated argument slots in the activation record as attached; this ensures that these arguments are always attached to the operation's controlling argument. In addition, the compiler generates additional code to migrate the argument objects to the invocation node. For example, the following code fragments compile to (roughly) the same machine code [2]:

```
OPERATION foo (ME, arg: $MOVE some_type)
IS BEGIN ...

OPERATION foo (ME, arg: $ATTACHED some_type)
```

[2] `Colocate` is a runtime support routine that tries to migrate its first argument to the second argument's location. It returns the initial location of the first argument if migration succeeded, `Nil` otherwise.

```
IS BEGIN
    colocate (arg, me);
    ...
```

If arguments are passed by visit, the generated code must also remember the object's original location; thus,

```
OPERATION foo (ME, arg: $VISIT some_type)
IS BEGIN
    ...
    RETURN;
    ...
```

is equivalent to

```
OPERATION foo (ME, arg: $ATTACHED some_type)
IS BEGIN
    VAR arg_loc: Object := colocate (arg, me);
    ...
    IF arg_loc ~= Nil THEN arg.$location := arg_loc END IF;
    RETURN;
    ...
```

5 Performance

We have performed several experiments to evaluate the performance of DOWL. All figures presented here were obtained by measuring the time required to execute the following operation:

```
OPERATION time (me, x: Dummy, n: Integer)
IS BEGIN
    LOOP EXIT WHEN (n <= 0);
        foo (x);
        n := n - 1;
    END LOOP;
END; !time
```

where `foo` is a null operation which returns immediately. Overhead introduced by checking for proxies and by performing remote operation invocation was determined by varying the characteristics (mutable vs. constant) and the relative locations of `x` and `me` (the object executing the loop). All experiments were conducted on two unloaded DECstation models 5000/240 connected by our departmental ethernet.

In the first experiment, both `me` and `x` were constant objects. Thus, neither the calling loop nor the called operation contained any proxy checks and the generated code was effectively the same as that generated by plain Trellis. The time required for one iteration was 3.8 microseconds.

In the second experiment, both `me` and `x` were mutable objects residing on the same node, which added four additional proxy checks to each iteration (three

in the loop body and one in `foo`'s entry code). Executing one iteration now required 4.9 microseconds, an overhead of 29% compared to Trellis.

However, this test models the worst case possible: a very tight loop doing no useful work. For real programs we've encountered much smaller performance degradations. For instance, a program creating a large graph and rearranging it several times was found to run about 9% slower under DOWL than it did under Trellis. This difference occurs because all user-defined objects are ultimately defined in terms of primitive types provided by the system, and a substantial amount of time is spent in operations on these building blocks. Since most primitive types are either constant or replicated, no proxy checks are required.

Proxy checking overhead could further be reduced by changing the strategy employed to place the checks: Instead of placing a proxy check at operation entry and after every operation invocation to guarantee that the controlling object is *always* resident, the checks could be placed before each access to an instance variable to ensure that the object is resident only when it is *necessary*. Flow analysis could reduce the number of proxy checks even further (e.g. there is only one check required for two instance variable accesses without an intervening operation invocation). This strategy will eliminate *all* checks from the timing loop above; its effect on real programs is currently under investigation.

To determine the performance of remote operation invocation, both `me` and `x` were fixed on different nodes. The time required to perform one iteration was 14.1 milliseconds, of which about 2.4 milliseconds were spent doing the actual TCP/IP transmission. In contrast, a null remote procedure call to a multithreaded server takes only 5.5 milliseconds using DCE RPC.

We suspect that a large fraction of DOWL's remote invocation overhead is caused by the message queues and servers connecting the two communication system layers, because performing a remote invocation requires at least seven activity context switches (there are even more if a large amount of data is transferred), which have a high latency in our implementation. Providing a procedural interface between the two layers could eliminate four of them and should thus increase performance significantly.

6 Related Work

Over the past years, several new object-based languages and extensions to existing languages with support for distribution have been presented. Some well-known examples are Emerald [3, 4, 9], Distributed Smalltalk [7, 2, 11] and Amber [6]. All systems provide location-independent invocation and migration of objects at run time.

Emerald is an object-based language. Its major features include a uniform object model, linguistic support for mobility control and for expressing object relationships.

Several implementations of Distributed Smalltalk allow interaction between Smalltalk images on different nodes. None of these systems supports mobility

control or mechanisms to express object relationships; however, [2] addresses issues like access control, remote debugging and connecting heterogeneous nodes.

Amber augments a subset of C++ with primitives to manage concurrency and distribution. The system is designed to maximize performance by exploiting concurrency in short-lived applications. This is achieved by providing a network-wide shared virtual memory. There are no declarative mechanisms to restrict object mobility or to specify which objects are related; however, objects can be attached at run time. Both Amber and Emerald locate remote objects using forwarding addresses.

Most of the DOWL extensions have been inspired by Emerald. However, the appearance of the location primitives differs: Emerald uses predefined language constructs, while DOWL implements them in the standard library. Our approach allows individual types to redefine and customize these operations, e.g. to do type-specific cleanup or initialization when an object leaves or enters a node.

Emerald represents a global object as pointers to a descriptor containing addressing information and a pointer to the object's storage area which is only valid if the object is resident. Local objects are direct pointers to the storage. This scheme consumes less memory than DOWL's object structure (the descriptor usually is much smaller than the actual data), but it is less efficient because any access to an instance variable requires double indirection.

Proxies in Distributed Smalltalk are full-fledged objects that know about the location of the object they represent. Remote object invocation is implemented based on the `doesNotUnderstand` mechanism. Conversion between objects and proxies is achieved by exchanging object table entries, an option not available to systems like Emerald and DOWL where objects are represented by pointers.

Amber's base architecture is a shared virtual memory; thus, objects are represented by their addresses which are valid on every node. Remote object invocation is implemented using residence checks on operation invocation and return; after a failed check, the invocation context is migrated to the object and the operation is resumed.

When invoking an operation on a global object, Emerald checks a bit in the object descriptor to determine whether the object is resident. If this check fails, the parameters are sent to the object's actual location to perform the operation. To allow the migration of an object which has active operation invocations, the run time system searches all stacks for activation records corresponding to invocations on the migrating object. Any activation records found are moved along with the object and copied onto a new stack at the target node. The top part of the original stack (corresponding to operations called by the migrating context) is copied to a new stack on the original node. The boundaries of the three stacks are finally modified to appear as if remote operations were performed instead of local invocations.

DOWL's scheme migrates an operation context only if it absolutely has to; in particular, moving an object temporarily away from a node (which might happen frequently if call-by-visit is employed) need not result in any overhead except for migrating the object.

7 Conclusion

We have presented the runtime system of DOWL, an extension of the Trellis language to support transparent distribution: the object format, the structure of the communication system and the mechanism to invoke operations on remote objects. We believe that the performance of the system is acceptable, even though no tuning efforts have been implemented so far. DOWL is not radically different from the (few) other known distributed object-oriented systems, but excels by several subtle features, e.g. the possibility to circumvent distribution transparency if required, an object format allowing efficient instance variable access, decoupling local from global garbage collection and finally the sophisticated type system and programming environment inherited from Trellis.

Acknowledgments

I would like to thank Lutz Heuser for many instructive discussions during the design of DOWL and Max Mühlhäuser for providing helpful comments on a draft version of this paper. Special thanks go to Wulf Becherer. He designed and implemented key components of the runtime system.

This work was supported by Digital Equipment Corp. and was performed in part at the Computer Science Department at the University of Kaiserslautern.

References

1. Achauer, B.: Distribution in Trellis/DOWL. Proc. TOOLS USA '91
2. Bennet, J.: The Design and Implementation of Distributed Smalltalk. Proc. OOPSLA '87
3. Black, A., Hutchinson, N., Jul, E., Levy, H.: Object Structure in the Emerald System. Proc. OOPSLA '86
4. Black, A., Hutchinson, N., Jul, E., Levy, H., Carter, L.: Distribution and Abstract Types in Emerald. IEEE Trans. Software Engineering SE-13, 1987
5. O'Brien, P., Halbert, D., Kilian, M.: The Trellis Programming Environment. Proc. OOPSLA '87
6. Chase, J., Amador, F., Lazowska, E., Levy, H., Littlefield, R.: The Amber System: Parallel Programming on a Network of Multiprocessors. Proc. 12th ACM Symp. Operating Systems Principles, 1989
7. Decouchant, D.: Design of a Distributed Object Manager for the Smalltalk-80 system. Proc. OOPSLA '86
8. Greif, I., Seliger, R., Weihl, W.: A Case Study of CES: A Distributed Collaborative Editing System Implemented in Argus. Programming Methodology Group Memo 55, MIT, 1987.
9. Jul, E., Levy, H., Hutchinson, N., Black, A.: Fine-Grained Mobility in the Emerald System. ACM Trans. Computer Systems 6(1), 1982
10. Liskov, B.: Overview of the Argus Language and System. Programming Methodology Group Memo 40, MIT, 1984.
11. McCullogh, P.: Transparent Forwarding: First Steps. Proc. OOPSLA '87
12. Schaffert, C., Cooper, T., Bullis, B., Kilian, M., Wilpolt, C.: An Introduction to Trellis/OWL. Proc. OOPSLA '86