Sharing objects in a distributed, single address space environment

Gianluca Dini, Lanfranco Lopriore*

Dipartimento di Ingegneria della Informazione: Elettronica, Informatica, Telecomunicazioni,
Università degli Studi di Pisa, via Diotisalvi 2, 56126 Pisa, Italy

Received 3 February 2000; received in revised form 5 June 2000; accepted 5 June 2000

Abstract

With reference to an object type defining the two basic operations, read and write, we present solutions to the object sharing problem, classified according to the migration and/or replication of the shared objects. We refer to a memory management system supporting a single address space view of storage in a distributed environment. Our system defines a small, powerful set of primitives that allow processes to explicitly control the allocation and deletion of the virtual pages in the physical storage as well as the page movements across the memory hierarchy. By using real programs, we demonstrate that these primitives make it possible to integrate an object sharing algorithm within the implementation of the given object at little programming effort. The discussion takes a number of salient issues into consideration, including network costs and a conceptual framework for actual implementation of the memory management system. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Distributed system; Migration; Replication; Shared object; Single address space

1. Introduction

The two classical concepts, a tightly coupled multiprocessor featuring a shared memory accessible to all processors and a distributed multicomputer composed of nodes (computers) interconnected through a high speed local network, have been recently combined into the concept of a distributed shared memory [13,15,20] in which the shared memory paradigm is implemented on a physically distributed memory system.

Implementation models for distributed shared memory can be classified according to the migration and/or replication of the shared objects [19]. Migration means that a given shared object moves to the node where it will be accessed next, replication means that multiple copies of the shared object may coexist at one time. In the central server algorithm, both migration and replication are forbidden. A single copy of the shared object is present throughout the system. The node storing this copy is called the central server. Both read and write accesses to the shared object are carried out by a process in the central server. In the migration algorithm, we have a single copy of the shared object, and this copy migrates to the node where it will be accessed next. Thus, this algorithm operates according to a single reader, single writer paradigm. In the read replication algorithm, the shared object can be replicated in different nodes. However, only one copy can be accessed for write, according to a multiple readers, single writer paradigm. Finally, in

* The work described in this paper has been carried out under the financial support of the Italian Ministero dell’Università e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the MOSAICO Project “Design methodologies and tools of high performance systems for distributed applications”.

* Corresponding author. Tel.: +39-050-568665; fax: +39-050-568522.

E-mail addresses: dini@iet.unipi.it (G. Dini), lopriore@iet.unipi.it (L. Lopriore).
the full replication algorithm, the shared object can be replicated in different nodes, and each object copy can be accessed for both read and write, according to a multiple readers, multiple writers paradigm.

The four algorithms exhibit very different behaviour and performance. Of course, a central server is likely to become a system bottleneck. A solution is to distribute the server responsibilities for different shared objects among the nodes, but this approach is viable only provided that applications are given the ability to locate the server of each object. Migration can be advantageous if the cost of moving an object across the network is low in comparison with the cost of a remote object access. Read replication has a potential for a form of parallelism in the read accesses to a shared object, but each write implies invalidation of all the object copies. Finally, full replication is characterised by a high flexibility, but consistency must be maintained between the multiple object copies. This result can be achieved by propagating the writes across the network.

In a classical distributed shared memory system, a single object sharing algorithm is used in all references to remote objects [10]. Excess network traffic follows if the algorithm does not match the object access pattern of the application programs. In a different, application-controlled approach, the memory management system includes a set of mechanisms allowing application programs to manage virtual memory explicitly. To this aim, each program will take the internal structure of the shared objects as well as its own pattern of object reference into account [23].

The advantages derived from a distributed shared memory implementation are enhanced by a single address space environment where all processes share access to a single virtual space, and this virtual space is both uniform and persistent [18, 21]. In an environment of this type, the meaning of an address is unique, and is independent of the process issuing this address [14]. Persistence eliminates the differentiation between primary and secondary memory. Uniformity removes the separation between local and remote storage. Thus, in the presence of a single address space, the address of a given object identifies this object for its entire lifetime, irrespective of its position in the network. In this paper, we present the following.

- A small set of virtual memory primitives is presented, that forms the application program interface of a memory management system operating according to the single address space paradigm in a distributed system context. These primitives support a uniform, persistent vision of storage as well as forms of application-controlled memory management. They make it possible to move the virtual pages across a two-layer memory hierarchy consisting of the resources of primary and secondary memory distributed among the network nodes.
  - With reference to the usual notion of an object type defined in terms of a set of values and a set of operations accessing these values, a remote call primitive is introduced that allows a program running on a given node to execute the operations of an object stored in a different node.
  - Four solutions to the object sharing problem are illustrated with reference to an object type defining the two basic operations, a read operation returning the object value and a write operation modifying this value [23]. Each solution corresponds to a different object sharing algorithm. The network costs of each algorithm are analysed in terms of the number of remote calls and the number of page copies generated by the execution of this algorithm.
  - A conceptual framework for an actual implementation of our memory management system is presented. A system prototype is briefly described that has been developed to evaluate the potentialities of the virtual memory primitives from the point of view of application program writing.

Of course, a relationship exists between the distributed shared memory paradigm and the single address space paradigm. In a distributed shared memory environment, the meaning of a memory address is independent of the node storing the corresponding information item. We have a form of transparent distribution causing the primary memory resources of all the network nodes to resemble a local primary memory. In a single address space environment, address independence from storage location is extended by the properties of uniformity and persistence to secondary memory as well. These considerations suggest that it should be possible to implement the distributed shared memory paradigm within the framework of a single address space system. This paper demonstrates that this result can be actually achieved, provided that adequate support is given by the application program interface of the memory management system.
2. Single address space

Let us consider a distributed system architecture where the nodes share access to a single virtual space logically broken into pages. The virtual address of a page is given by the address of the first storage location of this page. The virtual address of a given information item is the result of the addition of an offset to the address of the page containing this information item.

2.1. The virtual memory primitives

The memory management system supports a page-oriented, single address space abstraction. The system interface consists of a number of primitives, the virtual memory primitives, which allow application programs to manage the virtual pages. Fig. 1 shows the definition of the page type in terms of a C++ [22] class called Page. The virtual memory primitives are the public member functions of this class. Of course, the lifetime of a page is independent of the lifetime of the process making this page active. It follows that pages are dynamic objects, that are referenced by using pointers of type Page*

In each given node, the secondary memory is logically broken into blocks and the primary memory is logically broken into frames. Blocks and frames have the same, fixed size, which is equal to the page size. The primary memory and secondary memory devices of the network nodes give physical support to the virtual space, as follows. A given page can contain valid information only if it is active in a node. The new primitive makes a new page M active and returns the address of this page. This address will be used, for instance, to initialise a page pointer P as follows:

\[ \text{Page}^* P = \text{newPage}; \]

Let THIS_NODE denote the node issuing the call to this primitive. Execution reserves a free block \( \beta \) for storage of \( M \) in the secondary memory of THIS_NODE (execution fails if a free block is lacking in THIS_NODE).

The processor of a given node \( N \) can successfully reference an information item stored in active page \( M \) only if this page has been opened in \( N \). This effect is obtained by issuing the \( P->\text{open}(N) \) primitive (this primitive provides a default argument whose value is THIS_NODE). Let \( \beta \) be the block reserved for \( M \) in the secondary memory of the node where \( M \) is active. Execution reserves a free frame \( \varphi \) in the primary memory of \( N \) and copies the contents of block \( \beta \) into frame \( \varphi \) (execution fails if a free frame is lacking in \( N \)). If \( M \) is already open in \( N \), the primitive has no effect. If \( M \) is already open in a different node \( M \), then a page migration occurs, from \( M \) to \( N \). This means that the contents of \( M \) are copied from the frame \( \varphi \) reserved in \( M \) for \( M \) into \( \varphi \), and \( M \) is demoted to replicated in \( M \) (the concept of a replicated page will be introduced shortly).

Open page \( M \) can be saved by issuing the \( P->\text{save()} \) primitive. Let \( \beta \) be the block reserved for \( M \) in the secondary memory of the node where \( M \) is active, and let \( \varphi \) be the frame reserved for \( M \) in the primary memory of the node where \( M \) is open. Execution copies the contents of frame \( \varphi \) into block \( \beta \) (execution fails if \( M \) is not open). Page \( M \) can be closed by issuing the \( P->\text{close()} \) primitive. Execution releases frame \( \varphi \) (execution fails if \( M \) is not open). Finally, page \( M \) can be deleted by issuing the delete \( P \) primitive. Execution releases block \( \beta \) (execution fails if \( M \) is open or is not active). Consequently, the page contents are lost forever, and it will never be possible to access these contents again.

A page open in a given node \( N \) can be replicated in a different node \( M \). For page \( M \), this effect will be obtained by issuing the \( P->\text{replicate}(M) \) primitive (this primitive provides a default argument whose value is THIS_NODE). Execution reserves a free frame \( \varphi' \) in the primary memory of \( M \) and copies the page contents from the frame \( \varphi \) reserved for \( M \) in \( N \) into \( \varphi' \) (execution fails if \( M \) is not open, or a

```cpp
class Page {
    char c[PAGE_SIZE];
public:
    Page();
    void* operator new(size_t);
    void open(Node N = THIS_NODE);
    void save();
    void close();
    void replicate(Node M = THIS_NODE);
    void remove(Node M = THIS_NODE);
    void operator delete(void*);
    ~Page();
};
```

Fig. 1. A C++ class defining type Page.
free frame is lacking in \( M \). If \( \Pi \) is already replicated in \( M \), the primitive updates the replica, by taking the new page contents from \( \varphi \). If \( \Pi \) is open in \( M \), the primitive has no effect. Replication makes \( \Pi \) accessible to the processor in \( M \). However, in contrast to an open page, the contents of the replica cannot be saved back to the secondary memory. The replica can be \textit{removed} from node \( M \). This effect is obtained by issuing the \( P->\text{remove}(M) \) primitive. Execution releases the frame \( \varphi \)' reserved in \( M \) for the replica (this primitive provides a default argument whose value is \texttt{THIS\_NODE}; execution fails if \( \Pi \) is not replicated in \( M \)).

A page can be open in a single node at a time, whereas no limit is imposed on the number of page replicas. However, at any given time, a node can reserve at most one frame for the given page. This means that if a page is open in a given node, it cannot be replicated in this node, and two replicas of the same page cannot coexist in the same node.

2.2. Objects

We shall refer to the usual notion of an object type defined in terms of a set of values and a set of operations accessing these values. In the virtual space, a given object type \( T \) is supported by a memory area, the \textit{type descriptor} \( \Delta_T \), reserved for storage of the machine code representation of the operations of this type. For small-sized operations, the descriptor occupies a single page or even a portion of a single page, for large operations the descriptor spans several pages. An object \( \xi \) of type \( T \) is supported by a memory area, the \textit{object descriptor} \( \Delta_{\xi} \), reserved for storage of the internal representation of this object. The object descriptor may be as small as a single page or even a portion of a single page, whereas, for a large object, the descriptor spans several pages.

In the following, we shall refer to a type \( T \) featuring the two basic operations, a \texttt{read()} operation returning the object value and a \texttt{write()} operation modifying this value (Fig. 2). These two operations are representative of two different classes, the class of the operations causing accesses to the object for read only and the class of the operations causing accesses for both read and write. The operations in the two classes may be arbitrarily complex, and may even involve a substantial amount of data processing. To simplify presentation, we shall hypothesise that both type descriptor \( \Delta_T \) and object descriptor \( \Delta_{\xi} \) occupy a single page, and that these pages are entirely used for the descriptors. This means that the migration or replication of a descriptor never leads to the migration or replication of other information items. In this hypothesis, the page address \( d_T \) of type descriptor \( \Delta_T \) can be obtained by converting the address of any operation of type \( T \) into type \( \texttt{Page}^* \). This result can be achieved by using a cast, as follows:

\[
\texttt{Page}^*\ d_T = (\texttt{Page}^*) &T::T;
\]

An object \( \xi \) of type \( T \) is allocated by defining a pointer \( pX \) of type \( T^* \) and executing the \texttt{new} operation, as follows:

\[
T^*\ pX = \text{\tt new } T;
\]

The run time support is responsible for activating a pool of pages in each node and reserving these pages for the heap. Allocation of the new object will use one of these pages for storage of object descriptor \( \Delta_{\xi} \). The page address \( d_X \) of \( \Delta_{\xi} \) can be obtained by converting \( pX \) into type \( \texttt{Page}^* \), as follows:

\[
\texttt{Page}^*\ d_X = (\texttt{Page}^*)\ pX;
\]

An operation of type \( T \) can be executed in node \( N \) only if type descriptor \( \Delta_T \) and object \( \Delta_{\xi} \) occupy a single page, and that these pages are entirely used for the descriptors. This means that the migration or replication of a descriptor never leads to the migration or replication of other information items. In this hypothesis, the page address \( d_T \) of type descriptor \( \Delta_T \) can be obtained by converting the address of any operation of type \( T \) into type \( \texttt{Page}^* \). This result can be achieved by using a cast, as follows:
2.3. Remote calls

Let us refer to a program running on a given node \( C \) (the client node) and issuing a call to an operation involving an object presently open in a different node \( S \) (the server node). This remote call causes execution of a stub (piece of code) linked to the calling program in the client node. The stub is responsible for carrying out a marshalling activity converting the arguments of the call into a message form suitable for transmission across the network \[1\]. At the server site, a stub linked to the object is responsible for an unmarshalling activity leading to reconstruction of the arguments. The operation is now executed in the server node, and then a reply containing the operation results is sent to the client node \[4\]. Thus, the semantics of the remote call is that of a synchronous interprocess communication. The client program is blocked in wait for the results of the call coming from the server node after execution of the operation has been accomplished.

Our system supports the concept of a remote call at the object level. Let \( \xi \) be an object of type \( T \), let \( \text{pX} \) be a pointer to \( \xi \) and let \( \text{pf} \) be a pointer to an operation \( f() \) of type \( T \). If object descriptor \( \Delta_\xi \) is open in node \( S \), a call to \( f() \) involving \( \xi \) can be issued from a different node \( C \) by using the remote call primitive. This primitive has the form \( \text{remote} (\text{pf}, \text{pX}, \text{arg1}, \text{arg2}, \ldots) \), where \( \text{arg1}, \text{arg2}, \ldots \) are the arguments of \( f() \). Execution of \( \text{remote}() \) causes client node \( C \) to determine the name \( S \) of the server node. Then, the call is transmitted to \( S \), where it generates a local call of the form \( (\text{pX}"->\text{pf}) (\text{arg1}, \text{arg2}, \ldots) \). On termination, the result of the execution of this local call is returned to node \( C \), where it forms the result of \( \text{remote}() \). In the rest of this paper, to simplify presentation, we shall use notation \( \text{pX}"->f(\text{arg1}, \text{arg2}, \ldots) \) to indicate the call \( \text{remote}(\text{pX, pf, arg1, arg2, \ldots}) \).

If \( \text{remote}() \) is issued in a given node and the object involved in the remote call is stored in the same node, then the call is converted into a local call, and this conversion takes place at run time. In this respect, a local call is a special case of a remote call. Of course, performance losses follow from the run time conversion. If the programmer (compiler) can determine that a given call is always local, usage of the local call construct is more appropriate.

3. Sharing objects

In our implementation of the object sharing algorithms, an object of type \( \text{OwnerT} \) is associated with object \( \xi \) of type \( T \). This object is called the owner of \( \xi \). It is responsible for allocation and deallocation of \( \xi \), as well as for co-ordination of the concurrent accesses to \( \xi \). In particular, the owner enforces the required degree of mutual exclusion and synchronisation on these accesses. Fig. 3 shows the definition of type \( \text{OwnerT} \). Constructor \( \text{OwnerT}() \) allocates \( \xi \) in the same node as its owner; however, as will be made clear shortly, \( \xi \) may well migrate to a different node, as is required by the object sharing algorithm selected for \( \xi \). Besides destructor \( \sim\text{OwnerT}() \), the interface of type \( \text{OwnerT} \) includes a number of other operations, that are algorithm-specific, and will be introduced later.

An application program running on node \( N \) will access \( \xi \) by using an interface object called the handle. This is an object of type \( \text{HandleT} \). It is essentially aimed at decoupling the application program from the implementation of the object sharing algorithm. The definition of type \( \text{HandleT} \) includes operations \( \text{read}() \) and \( \text{write}() \), corresponding to the homonymous operations of type \( T \) (Fig. 4). Moreover, the interface of type \( \text{HandleT} \) may include algorithm-specific operations, mainly aimed at controlling the interactions with the owner. In order to execute an operation on \( \xi \), an application program running in a given node executes the corresponding operation of the handle in this node. Thus, execution of an operation on \( \xi \) produces interactions between \( \xi \), its owner and the handle. These interactions take place by using remote calls. In the following, we shall present the form assumed by types \( \text{OwnerT} \) and \( \text{HandleT} \) in each object sharing algorithm.

```java
class OwnerT {
    T* pX;
    /* ... */

    public:
    OwnerT() { pX = new T; /* ... */ }
    /* ... */
    ~OwnerT() { delete pX; }
};
```

Fig. 3. Definition of type \( \text{OwnerT} \).
3.1. Central server

As seen in Section 1, in the central server algorithm a single copy of the shared object $\xi$ exists throughout the system, in the node called the central server of $\xi$ (Fig. 5). An application request for a read or a write is transmitted by the handle to the owner. The owner coordinates the requests from the different handles and transmits each request to the central server, where execution of the operation takes place. On termination, the central server returns the result to the owner, that forwards it to the handle. Initially, when $\xi$ is allocated, the owner resides in the central server. This situation can change later, as is required to redistribute the workload among the nodes, for instance.

Fig. 6 shows the definition of type OwnerT in the central server algorithm. The internal representation of the owner includes a semaphore called mutex. This is an object of type Semaphore. The definition of this type corresponds to the classical definition of a semaphore, and is not shown. mutex is used to implement the required degree of mutual exclusion between the concurrent accesses to the internal representation of $\xi$, that are generated by handles running on different nodes.

The OwnerT() constructor initialises mutex to 1 and allocates $\xi$. Then, the page address of type de-
class OwnerT {
  T* pX;
  Page* dT, *dX;
  Semaphore mutex;
public:
  OwnerT();
  void changeCentralServer(Node M);
  Info read();
  void write(Info value);
-OwnerT() { delete pX; }
};

OwnerT::OwnerT() : mutex(1) {
  pX = new T;
  dX = (Page*)pX;
  dT = (Page*)&T::T;
}

void OwnerT::changeCentralServer(Node M) {
  dT->replicate(M);
  mutex.wait();
  dX->open(M);
  mutex.signal();
}

Info OwnerT::read() {
  mutex.wait();
  Info result = pX->read();
  mutex.signal();
  return result;
}

void OwnerT::write(Info value) {
  mutex.wait();
  pX->write(value);
  mutex.signal();
}

Fig. 6. Type OwnerT in the central server algorithm.

scriptor ΔT is stored into data member dT, and the page address of object descriptor Δξ is stored into data member dX. Let η be the node where the owner is being executed. The changeCentralServer() operation replicates ΔT and opens Δξ in a new node M. This node assumes the role of the central server of ξ, and consequently, it will carry out the data processing actions connected with every object access. The read() and write() operations invoke the homonymous operations of ξ by using the remote call construct, as is required to deal with possible central server changes.

Fig. 7 shows the definition of type HandleT. The HandleT() constructor stores the address of the owner into data member owner. Operations read() and write() issue remote calls to the homonymous operations of the owner.

3.1.1. Observations

In a traditional, multiple address space environment, a basic problem in the implementation of remote calls is to determine the present network location of the object involved in each call. This problem is particularly complicated if the object is named by using pointers. In fact, in an environment of this type, each process references a private address space. The meaning of a pointer is confined within the address space boundaries, and consequently, this meaning is lost in a different node. Of course, similar problems may affect the arguments and the result of the remote call, if they are passed by reference, for instance.

Conversely, in a single address space environment, the meaning of an address is independent of both the process issuing this address and the node where this process is running. Our mechanism for the remote execution of object operations, embodied by the remote() primitive, takes advantage of this feature and integrates the actions required to locate the object at the memory management system level. In fact, in order to execute an operation on a given object, an application using remote() does not need to know the present network location of this object. In the central server algorithm, this location-independence property of remote() has been used to control the tendency of the central server to become a bottleneck.

class HandleT {
  OwnerT* owner;
public:
  HandleT(OwnerT* own) { owner = own; }
  Info read() { return owner->read(); }
  void write(Info value) { owner->write(value); }
};

Fig. 7. Type HandleT in the central server algorithm.
If we have several shared objects, we may have a number of servers and distribute the objects among them. Distribution can be made statically, by partitioning the address space into areas and assigning a server to each area, for instance. In this approach, when an object is created at a given address, a server is permanently assigned to this object, according to the object address. In a different, dynamic approach, no permanent association exists between objects and servers. Following this approach, we allow each node to be a central server. Moreover, the association between an object and its central server may change at run time.

In more detail, object $\xi$ is created in the node of its owner. Consequently, all calls made by the owner to the operations of the object are initially translated into local calls. After execution of operation `changeCentralServer()` of the owner, these calls are executed remotely, and the task of determining the present location of the object is demanded to the remote call mechanism. As pointed out in Section 2.3, an application program using a given local object does not need to be modified if the object is moved to a remote location in a different node, provided that all the operations of this object are executed by using `remote()`. In fact, the owner is unaware of the present location of the shared object in the network. Similar considerations apply to the handles with respect to the owner. The handles are unaware of the network location of the owner. Of course, if object $\xi$ is known to never migrate from the node of its owner, the definition of type `OwnerT` will codify the calls to the object operations as local calls, to improve efficiency.

It should be noted that migration of object $\xi$ does not cause migration of semaphore `mutex`. This is a consequence of the fact that `mutex` is part of the owner. In this way, we avoid the potential problems connected with the movements across the network of the queue of the processes in wait for the semaphore.

The possibility to redistribute the central server responsibilities for different objects among the nodes introduces a form of dynamic workload reconfiguration. However, with respect to a situation in which an object and its owner reside in the same node, a network traffic increase follows after a central server change from the remote calls to the shared object operations. As far as workload balancing among the nodes is concerned, this approach will be effective only if a substantial amount of data processing activity is connected with each object access. Even after a central server change, the duty of performing the access remains concentrated in a single node, the new central server. If a more dynamic workload partitioning mechanism is needed, a different object sharing algorithm must be used, e.g. migration.

### 3.2. Migration

In the migration algorithm, a single copy of shared object $\xi$ exists throughout the network, in the node running the handle that accessed the object most
recently (Fig. 8). Suppose that $\xi$ is presently open in node $N$. If an application program running in a different node $M$ issues an operation on $\xi$, the object migrates from $N$ to $M$, and then the operation is executed in $M$. This means that the descriptor $\Delta_{T}$ of type $T$ must be replicated in $M$ and the descriptor $\Delta_{\xi}$ of object $\xi$ must be opened in $M$.

Fig. 9 shows the definition of type $\text{OwnerT}$ in the migration algorithm. A semaphore called $\text{mutex}$ is used to implement a critical section protecting $\xi$. Constructor $\text{OwnerT}()$ initialises $\text{mutex}$ to 1 and allocates $\xi$. Operations $\text{acquire()}$ and $\text{release()}$ will be used by the handles to acquire and release the critical section. Operation $\text{ppX()}$ returns the address of $\xi$. Destructor $\text{~OwnerT}()$ deletes $\xi$.

Fig. 10 shows the definition of type $\text{HandleT}$. The $\text{HandleT}()$ constructor replicates type descriptor $\Delta_{T}$. The $\text{read()}$ operation acquires the critical section protecting $\xi$, and then opens object descriptor $\Delta_{\xi}$. As a result, $\Delta_{\xi}$ migrates from its present location to the node executing the handle. Then, the read access to $\xi$ is performed locally, the critical section is released and the result of the read is returned. The $\text{write()}$ operation has a similar structure. Finally, the $\text{~Handle()}$ destructor removes type descriptor $\Delta_{T}$. However, the destructor does not close $\Delta_{\xi}$. Instead, after termination of the handle, $\Delta_{\xi}$ will migrate to a different node on the first access to $\xi$ performed by the handle running in that node. It should be clear that as long as a given handle is accessing $\xi$, the critical section protecting $\xi$ prevents the handle running in a different node from causing migration of $\Delta_{\xi}$ to that node.

3.2.1. Observations

In our implementation, handles do not have to keep trace of the object movements across the network. We have obtained this important result by relying on the ability of the $\text{open()}$ primitive to identify the node presently storing object descriptor $\Delta_{\xi}$. If a handle issues a sequence of two or more calls to $\text{read()}$ or $\text{write()}$, only the first of these calls causes migration of $\Delta_{\xi}$. The subsequent calls execute $\text{open()}$, however, as seen in Section 2.1, no page copy is actually produced. Of course, a careful implementation of $\text{open()}$ will guarantee low processing time costs in situations of this type.

3.3. Read replication

In the read replication algorithm, multiple copies of the shared object $\xi$ exist, and only one of them can be accessed for write. This copy is stored in the node of a handle that we call the writer (Fig. 11).
the other object copies can only be accessed for read. These copies are stored in the nodes of the handles that we call the readers. Thus, the descriptor $\Delta_\xi$ of $\xi$ is open in the node of the writer and is replicated in the nodes of the readers. The owner of $\xi$ is responsible for maintaining a list of pointers to those handles that hold a valid descriptor replica. This list is called the copy set. A descriptor replica is valid if it reflects the present contents of the descriptor as stored in the writer. In order to assume the role of the writer, a handle must qualify to the owner. This action causes the owner to invalidate the descriptor replicas of every handle in the copy set. The handle in a given node controls whether the replica stored in this node is valid before accessing this replica. If the replica is not valid, the read access is preceded by the update of this replica.

Fig. 12 shows the definition of type $\text{OwnerT}$. Semaphore $\text{mutex}$ is used to implement a critical section protecting the copy set. The definition of type $\text{CopySetT}$ can be easily imagined and is not shown. The internal representation of the copy set takes the form of a collection of pointers to handles. Initially, the copy set is empty. Operations $\text{acquireForRead()}$ and $\text{release()}$ are used by the readers to acquire and release the critical section protecting the copy set. $\text{release()}$ also causes insertion of the reader into the copy set. Operation $\text{acquireForWrite()}$ allows a handle wishing to assume the role of the writer to qualify to the owner. Besides acquiring the critical section protecting the copy set, this operation executes a loop that uses operations $\text{first()}, \text{next()}$ and $\text{last()}$ of type $\text{CopySetT}$ to obtain the pointers to the handles in the copy set, one handle for each iteration. An invalidation command is sent to each of these handles by using the $\text{invalidate()}$ operation of type $\text{Handle()}$, and then the handle is extracted from the copy set.

Fig. 13 shows the definition of type $\text{HandleT}$. Boolean data member $\text{valid}$, if set, specifies that the local replica of shared object $\xi$ is valid. $\text{valid}$ is set by the handle and is cleared by the owner by using operation $\text{invalidate()}$. It follows that a critical section must protect the accesses to $\text{valid}$. Operation $\text{read()}$ inspects $\text{valid}$ to ascertain whether the local object replica is valid. If this is not the case, it replicates the object descriptor. This action is preceded by the acquisition of the critical section protecting the copy set in the owner, and is followed by the insertion of the new reader into the copy set and the release of the critical section. To this aim, operations $\text{acquireForRead()}$ and $\text{release()}$ of the owner are used. Operation $\text{write()}$ issues a remote call to the $\text{acquireForWrite()}$ operation of the owner, thereby invalidating all the other object replicas. Then, object descriptor $\Delta_\xi$ is opened, $\text{valid}$ is set, the write is carried out and the critical section is released. Finally, the $\text{~Handle()}$ destructor invokes the $\text{removeHandle()}$ operation of the owner.
to remove the handle from the copy set. Then, the
local replicas of both type descriptor \( \Delta_T \) and object de-
scriptor \( \Delta_x \) are removed.

3.3.1. Observation

We maintain the information concerning the valid-
ity of the object replica stored in each given handle
not only in the handle, but also in the owner. This
result is obtained by eliminating a handle from the copy
set as soon as the replica in this handle is invalidated,
and inserting the handle again into the copy set when
the replica is updated. In this way, when a new handle
assumes the role of the writer, we save the transmis-
sion of an invalidation request from the owner to each
handle that holds an invalid replica.
3.4. Full replication

In the full replication algorithm, multiple copies of the shared object $\xi$ exist, one copy for each handle, and every copy can be accessed for both read and write. This effect is obtained as follows. The descriptor $\Delta_\xi$ of object $\xi$ is opened in the node of the owner, and is replicated in the node of each handle (Fig. 14). The owner maintains a copy set taking the form of a list of pointers to handles. A handle wishing to write a new value into $\xi$ sends a write request to the owner. The owner propagates this request to every handle in the copy set.

Fig. 15 shows the definition of type OwnerT. Operation newHandle() adds a new handle to the copy set, and operation removeHandle() removes a handle from the copy set. Operation propagate() is called by a handle wishing to assign a new value to $\xi$. Execution of this operation writes the new value locally, and then sends an update request to each handle in the copy set.

Fig. 16 shows the definition of type HandleT. Constructor HandleT() replicates both type descriptor $\Delta_T$ and object descriptor $\Delta_\xi$. Moreover, the new handle is inserted into the copy set by using the newHandle() operation of the owner. Operation read() executes the read access to the local copy of $\xi$. Operation write() sends a propagation request to the owner. Operation update() updates the value of the local replica of $\xi$ by using the new object value received from the owner. Finally, destructor ~HandleT() causes the owner to delete the handle from the copy set, and then removes the local replicas of both $\Delta_T$ and $\Delta_\xi$.

4. Discussion

4.1. Network traffic

Table 1 summarises the network costs associated with the execution of the two object operations, read and write, in each of the four object sharing algorithms. The costs are expressed in terms of the number of remote calls and the number of page copies. Of course, from the point of view of application program performance, different time requirements are associated with a page copy produced by the replicate() primitive with respect to a copy produced by the open() primitive. In fact, replicate() copies the page contents from primary memory, whereas open() copies these con-
Contents from secondary memory (as seen in Section 2.1, an exception is a situation in which the page is already open in a different node). On the other hand, as far as network traffic is concerned, these two primitives are equivalent, and therefore, the table does not distinguish the two cases.

For all the algorithms, the number of messages needed to accomplish a given shared object operation is independent of the number of nodes that form the network. Thus, any network size increase leads to no program performance degradation. In two cases, the central server and the migration algorithms, the cost is even independent of the number of handles participating in the object sharing activities. In the presence of a high number of participants, this may be a reason for selecting one of these algorithms.

Of course, from the point of view of network performance, the central server algorithm is a good performer, as it requires no page copy at all. In this algorithm, for both read and write operations, the cost associated with an access to a shared object is equal to two remote calls. The first call is generated by the handle to transmit the access request to the owner, the other call is generated by the owner to the object. Of course, this is a worst case situation that only applies if workload redistribution has occurred and the object has been moved away from the node of the owner. In the common situation in which the owner and the
object reside in the same node, the cost of a read or write is that of a single remote call.

Migration adds the cost of a page copy produced by execution of the open() primitive. However, if a handle generates a sequence of S object accesses and this sequence is not interrupted by accesses from the other handles, then only the first access in the sequence requires actual object migration. Thus, a single page copy is needed for the entire sequence. In a situation of this type, the average cost of a read or a write, expressed in terms of the number of page copies, is given by ratio 1/S. The number of remote calls to the operations of the owner is the same as in the central server algorithm. However, in the central server algorithm the node of the owner bears the entire workload associated with every shared object access, whereas in the migration algorithm these calls are only aimed at entering and exiting a critical section, and consequently, they absorb a small amount of processing power. It should be recalled that read() and write() are representative of the two classes of operations, those accessing the shared object for read only and those accessing the object for both read and write. A substantial amount of data processing activities may be connected with the execution of the operations in the two classes. In a situation of this type, real advantages follow from the form of dynamic workload balance embodied by the migration algorithm.

In the read replication algorithm, multiple read accesses can proceed in parallel on behalf of as many handles. If the local replica of the object descriptor is valid, a read access takes place without even informing the owner. In the worst case situation of an invalid local replica, the network cost is two remote calls and one page copy. In the hypothesis of an uninterrupted sequence of S read accesses, this cost is paid for the first access only. Thus, the average cost of each access is 2/S remote calls and 1/S page copies. As far as the writes are concerned, we have to invalidate all the valid object copies stored in the nodes of the readers. In the case of a sequence of S writes, invalidation will be carried out by the first write. This means that

---

Table 1

Network traffic

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Remote</td>
<td>Page copies</td>
</tr>
<tr>
<td>Central server</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Migration</td>
<td>2</td>
<td>1/Sa</td>
</tr>
<tr>
<td>Read replication</td>
<td>2/S</td>
<td>1/S</td>
</tr>
<tr>
<td>Full replication</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

---

*a* Number of accesses in an uninterrupted sequence.

*b* Number of handles presently holding a valid object copy.

*c* Number of handles.
the average invalidation cost for each write is given by ratio $2 + V/S$, $V$ being the number of valid object copies.

Finally, in the full replication algorithm both the read and write accesses are performed on the local object copy at no network cost. On the other hand, each write must be propagated to all the handles. High processing time costs may follow in each node to accomplish the actions involved in the write.

4.2. Implementation issues

A simple implementation scheme for our virtual memory primitives uses a single master node as a repository for the information for virtual page management. This information takes the form of a table, the master table, featuring one entry for each active page (Fig. 17). The entry for a given page contains the address $P$ of this page, the name $B$ of the node reserving a secondary memory block for this page and, if the page is open, the name $F$ of the node reserving a primary memory frame for this page. Furthermore, a block table and a frame table are stored in each given node $N$. The block table features one entry for each page active in $N$. The entry for a given page specifies the address $P$ of this page and the block $B$ reserved for page storage in the secondary memory of $N$. Thus, the maximum number of entries of the block table is equal to the number of the blocks that form the secondary memory of $N$. The frame table has one entry for each page open in $N$. The entry for a given page specifies the address $P$ of this page and the frame $\varphi$ reserved for page storage in the primary memory of $N$. Thus, the maximum number of entries of the frame table is equal to the number of the frames that form the primary memory of $N$.

Let $v$ be a virtual address generated by the processor of node $N$, and let $P$ be the page address corresponding to this virtual address. Conceptually, translation of $v$ into a physical address proceeds as follows (Fig. 18). Quantity $P$ is used to access the frame table and find the number $\varphi$ of the corresponding primary memory frame. This frame number is paired with the offset to obtain the address of the referenced information item in the primary memory of $N$.

In a system configuration of this type, let us summarise the actions involved in the execution of the $P \rightarrow \text{replicate}(M)$ virtual memory primitive, for instance. Let $\Pi$ be the page referenced by $P$, and let $N$ be the node where $\text{replicate}()$ has been issued. A message containing quantity $P$ is sent from $N$ to the master node. On receiving this message, the master node performs a search in the master table for the entry corresponding to $P$. The name $F$ of the node where page $\Pi$ is open is read from this entry and is returned to $N$. Then, node $N$ sends a message containing quantities $P$ and $M$ to node $F$. On receiving this message, node $F$ performs a search in the frame table for the entry corresponding to $P$. The name $\varphi$ of the frame storing page $\Pi$ is read from this entry. The contents of this frame are sent to node $M$, where they are copied into a free frame $\varphi'$. Finally, an entry is reserved for
\( \Pi \) in the frame table of \( M \). Quantities \( P \) and \( \psi' \) are inserted into this entry.

As a further example, let us consider the remote call primitive \( \text{remote}(pf, pX, arg1, arg2, \ldots) \). Execution of this primitive in node \( N \) converts the address \( pX \) of the object \( \xi \) involved in the remote operation into the address \( dX \) of object descriptor \( \Delta_\xi \). Then, a message containing quantity \( dX \) is sent from \( N \) to the master node. On receiving this message, the master node performs a search in the master table for the entry corresponding to \( dX \). The name \( F \) of the node where \( dX \) is open is read from this entry and is returned to \( N \). Now, a marshalling activity takes place in \( N \), aimed at giving message form to address \( pf \) and the arguments \( arg1, arg2, \ldots \) of the remote operation. The resulting message is sent to node \( F \), where it is unmarshalled and is used to originate the local call \( (pX \mapsto pf)(arg1, arg2, \ldots) \). The results of this call are sent back to \( N \).

4.2.1. Observations

If one of the arguments \( arg1, arg2, \ldots \) of a remote call is transmitted by using a pointer, the marshalling process does not involve the data structure referenced by this pointer. Instead, only the pointer is sent to the recipient node \( F \), where it is converted into a page address as part of the actions carried out by the called operation. This page address is used to open the corresponding page, thereby gaining access to the data structure. Similar considerations apply to the operation results as well. So doing, we take advantage of a salient property of a distributed, single address space environment, i.e. a pointer preserves its meaning even when it is transmitted to a different node [14]. Significant performance improvements ensue, especially for large data structures, from the points of view of both the time required to marshal and unmarshal the arguments and the network traffic generated by the transmission of the arguments across the network (by contrast, in a traditional, multiple address space environment the scope of a pointer is confined within the boundaries of the process allocating this pointer and the node running this process).

It should be clear that the simple arrangement of the information for memory management, sketched out previously, is only a conceptual framework, rather than an effective implementation scheme. This is especially the case for the partitioning of this information into the three system tables, the master table, the block table and the frame table. Actual implementation will carefully enhance performance in terms of both table access times and space requirements for table storage, for instance. We rely on a centralised view of memory management in which the master node is accessed in the execution of every virtual memory primitive to find the location in memory of the page involved in the execution of this primitive. Of course, the master table tends to become a system bottleneck. Serious scalability problems arise as a consequence of the increase of the number of accesses to this table that follows if the network size increases. Different implementations may well be devised, distributing the memory management information among the nodes [2]. Similarly, the address translation scheme, outlined above, should be considered only as an abstract specification of the intended behaviour. Actual implementation will give efficient support to virtual to physical address translation, by taking advantage of the recent results of research made in this direction, for instance [8,9,12,16,24].

Incidentally, it is worth to note that a processor generating large addresses is an important requisite in every single address space architecture [5,11,21]. In the presence of small addresses, the size of the address space available for each process results unacceptably small. In fact, the recent advent of 64-bit processors is an important motivation for the increasing interest being paid to the single address space paradigm.

4.3. Prototypes

The memory management system, described in this paper, has been implemented in prototype form [6]. Our implementation supports a full distribution of the activities connected with memory management. The main objectives have been to keep the storage requirements of the information for memory management low and to minimise the number of messages transmitted across the network to determine the location of any given page in the distributed memory. Our solution leads to network costs that are independent of the number of nodes. Caching mechanisms, supported at the software level, reduce the network traffic by maintaining local copies of recently used address translation information.
The virtual memory primitives have been implemented within the framework of the Unix\textsuperscript{1} operating system, by taking advantage of the Unix facilities for process management and network communication. The resulting saving in coding efforts allowed us to concentrate our attention on the design of the application program interface. In fact, this prototype has been essentially aimed at determining the system requirements for the support of our intended classes of applications. In this respect, we paid special attention to object and process migration\textsuperscript{17} and the support of forms of application-controlled memory management\textsuperscript{3}. This prototype demonstrated that the programming effort connected with the coding of the virtual memory primitives is moderate. In fact, the average size of each page operation is 133 lines of C\textsuperscript{CC} source code.

At present, we are developing a second prototype from scratch. This new implementation is mainly aimed at performance measurements, and consequently, it is being built on a bare machine. So doing, we avoid interferences from an underlying operating system, that negatively affect the results of the measurements made. Limited forms of fault tolerance have been added, aimed at maintaining consistency of the memory management information in spite of specific classes of node and network failures\textsuperscript{7}.

5. Concluding remarks

In a distributed system framework, we have considered a memory management system implementing the single address space paradigm in a persistent, uniform environment. The application program interface of the system consists of a small, powerful set of primitives that implement the concept of a virtual memory page. These primitives make it possible to explicitly control allocation and deletion of the virtual pages in the physical storage, as well as the page movements in the primary and secondary memory resources distributed across the network nodes.

With reference to an object type defining the two basic operations, read and write, we have considered four algorithms for object sharing, corresponding to different paradigms of object migration and/or replication. We have presented the C++ programs implementing the four algorithms. These programs demonstrate that our virtual memory primitives provide effective support for distributed shared memory implementation. The network costs of each algorithm have been analysed by using the programs rather than modelling interprocess interactions at a higher abstraction level (e.g. by using graphs at the node level). The costs have been evaluated in terms of the number of remote calls and the number of page copies transmitted across the network.

In our approach, the programmer selects the algorithm that is more appropriate for each given object, according to the object behaviour in memory and the access pattern of the intended class of applications to the object\textsuperscript{10,23}. In the implementation illustrated in this paper, the link between the object and the algorithm is static. This is only a consequence of the need to keep presentation short. The modifications to be made to the programs presented in Section 3 to permit dynamic changes of the object sharing algorithm can be easily imagined. These modifications will take advantage of the fact that the solutions given use a unified object interface, the handle, that is independent of the algorithm. In fact, the handle is aimed at hiding the detail of the algorithm within the object implementation.

By moving the object sharing algorithm to the object implementation level, we give the programmer an opportunity to improve program performance and/or reduce network traffic. Our work demonstrates that these advantages can be gained at little programming effort, provided that adequate support is given by the underlying memory management system.

References


\textsuperscript{1} Unix is a registered trademark of AT&T.


Gianluca Dini was born in Leghorn, Italy, in 1965. He received his Dott. Ing. degree in electronics engineering from the University of Pisa, Pisa, Italy, in 1990, and his Perfezionamento (equivalent to a PhD degree) in computer engineering from the Scuola Superiore di Studi Universitari e di Perfezionamento “S. Anna”, Pisa, Italy, in 1995. From 1993 to 1999 he was a researcher at the Dipartimento di Ingegneria della Informazione, Elettronica, Informatica, Telecomunicazioni of the University of Pisa, where he is now an Associate Professor. His research interests include distributed systems and distributed programming.

Lanfranco Lopriore was born in Pisa, Italy, in 1954. In 1978, he received his Dott. Ing. degree in electronics engineering from the University of Pisa, Pisa, Italy. From 1982 to 1990, he was a researcher at the Istituto di Elaborazione della Informazione of the Italian National Research Council, Pisa. In 1990, he was made a Full Professor of computer engineering at the Dipartimento di Sistemi, University of Calabria, Cosenza, Italy. At present, he is a Full Professor of computer engineering at the Dipartimento di Ingegneria della Informazione, Elettronica, Informatica, Telecomunicazioni of the University of Pisa. His research interests are in the field of architectural supports for distributed systems and high level languages. He made research in the areas of program debugging environments, cache memories and capability-based architectures.