CS2056 DISTRIBUTED SYSTEMS

UNIT V

Distributed Shared Memory-Introduction-Design and implementation issues-Sequential consistency and Ivy case study Release consistency and Munin case study-Other consistency models.

CORBA Case Study- Introduction-CORBA RMI-CORBA services.

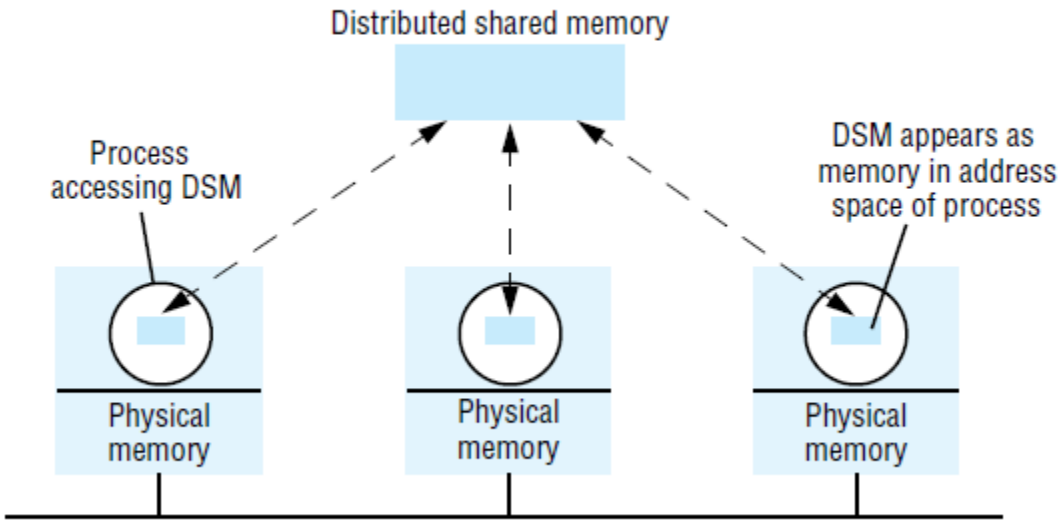
DISTRIBUTED SHARED MEMORY

Distributed shared memory (DSM) is an abstraction used for sharing data between computers that do not share physical memory. Processes access DSM by reads and updates to what appears to be ordinary memory within their address space. However, an underlying runtime system ensures transparently that processes executing at different computers observe the updates made by one another.

The main point of DSM is that it spares the programmer the concerns of message passing when writing applications that might otherwise have to use it. DSM is primarily a tool for parallel applications or for any distributed application or group of applications in which individual shared data items can be accessed directly. DSM is in general less appropriate in client-server systems, where clients normally view server-held resources as abstract data and access them by request (for reasons of modularity and protection).

Message passing cannot be avoided altogether in a distributed system: in the absence of physically shared memory, the DSM runtime support has to send updates in messages between computers. DSM systems manage replicated data: each computer has a local copy of recently accessed data items stored in DSM, for speed of access.

The distributed shared memory abstraction



In distributed memory multiprocessors and clusters of off-the-shelf computing components (see Section 6.3), the processors do not share memory but are connected by a very high-speed network. These systems, like general-purpose distributed systems, can scale to much greater numbers of processors than a shared-memory multiprocessor's 64 or so. A central question that has been pursued by the DSM and multiprocessor research communities is whether the investment in knowledge of shared memory algorithms and the associated software can be directly transferred to a more scalable distributed memory architecture.

Message passing versus DSM

As a communication mechanism, DSM is comparable with message passing rather than with request-reply-based communication, since its application to parallel processing, in particular, entails the use of asynchronous communication. The DSM and message passing approaches to programming can be contrasted as follows:

Programming model:

Under the message passing model, variables have to be marshalled from one process, transmitted and unmarshalled into other variables at the receiving process. By contrast, with shared memory the processes involved share variables directly, so no marshalling is necessary – even of pointers to shared variables – and thus no separate communication operations are necessary.

Efficiency :

Experiments show that certain parallel programs developed for DSM can be made to perform about as well as functionally equivalent programs written for message passing platforms on the same hardware – at least in the case of relatively small numbers of computers (ten or so). However, this result cannot be generalized. The performance of a program based on DSM depends upon many factors, as we shall discuss below – particularly the pattern of data sharing.

Implementation approaches to DSM

Distributed shared memory is implemented using one or a combination of specialized hardware, conventional paged virtual memory or middleware:

Hardware:

Shared-memory multiprocessor architectures based on a NUMA architecture rely on specialized hardware to provide the processors with a consistent view of shared memory. They handle

memory LOAD and STORE instructions by communicating with remote memory and cache modules as necessary to store and retrieve data.

Paged virtual memory:

Many systems, including Ivy and Mether , implement DSM as a region of virtual memory occupying the same address range in the address space of every participating process.

```
#include "world.h"
struct shared { int a, b; };
Program Writer:
main()
{
struct shared *p;
methersetup(); /* Initialize the Mether runtime */
p = (struct shared *)METHERBASE;
/* overlay structure on METHER segment */
p->a = p->b = 0; /* initialize fields to zero */
while(TRUE){ /* continuously update structure fields */
p –>a = p –>a + 1;
p –>b = p –>b - 1;
}
}
```

Program Reader:

```
main()
{
struct shared *p;
methersetup();
p = (struct shared *)METHERBASE;
while(TRUE){ /* read the fields once every second */
printf("a = %d, b = %d\n", p –>a, p –>b);
sleep(1);
}
}
```

Middleware:

Some languages such as Orca, support forms of DSM without any hardware or paging support, in a platform-neutral way. In this type of implementation, sharing is implemented by communication between instances of the user-level support layer in clients and servers. Processes make calls to this layer when they access data items in DSM. The instances of this layer at the different computers access local data items and communicate as necessary to maintain consistency.

Design and implementation issues

The synchronization model used to access DSM consistently at the application level; the DSM consistency model, which governs the consistency of data values accessed from different computers; the update options for communicating written values between computers; the granularity of sharing in a DSM implementation; and the problem of thrashing.

Structure

A DSM system is just such a replication system. Each application process is presented with some abstraction of a collection of objects, but in this case the 'collection' looks more or less like memory. That is, the objects can be addressed in some fashion or other. Different approaches to DSM vary in what they consider to be an 'object' and in how objects are addressed. We consider three approaches, which view DSM as being composed respectively of contiguous bytes, language-level objects or immutable data items.

Byte-oriented

This type of DSM is accessed as ordinary virtual memory – a contiguous array of bytes. It is the view illustrated above by the Mether system. It is also the view of many other DSM systems, including Ivy.It allows applications (and language implementations) to impose whatever data structures they want on the shared memory. The shared objects are directly addressible memory locations (in practice, the shared locations may be multi-byte words rather than individual bytes). The only operations upon those objects are *read* (or LOAD) and *write* (or STORE). If *x* and *y* are two memory locations, then we denote instances of these operations as follows:

$R(x)a$ – a *read* operation that reads the value a from location x.
$W(x)b$ – a *write* operation that stores value b at location x.

Object-oriented

The shared memory is structured as a collection of language-level objects with higher-level semantics than simple *read* / *write* variables, such as stacks and dictionaries. The contents of the shared memory are changed only by invocations upon these objects and never by direct access to their member variables. An advantage of viewing memory in this way is that object semantics can be utilized when enforcing consistency.

Immutable data

When reading or taking a tuple from tuple space, a process provides a tuple specification and the tuple space returns any tuple that matches that specification – this is a type of associative addressing. To enable processes to synchronize their activities, the *read* and *take* operations both block until there is a matching tuple in the tuple space.

Synchronization model

Many applications apply constraints concerning the values stored in shared memory. This is as true of applications based on DSM as it is of applications written for sharedmemory multiprocessors (or indeed for any concurrent programs that share data, such as operating system kernels and multi-threaded servers). For example, if *a* and *b* are two variables stored in DSM, then a constraint might be that *a*=*b* always. If two or moreprocesses execute the following code:

a:= *a* + 1;

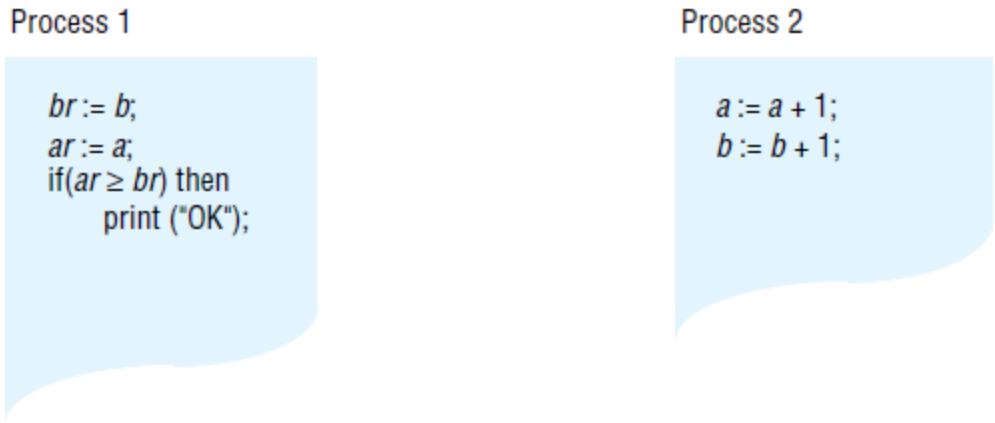
b := *b* + 1;

then an inconsistency may arise. Suppose *a* and *b* are initially zero and that process 1gets as far as setting *a* to 1. Before it can increment *b*, process 2 sets *a* to 2 and *b* to 1.

Consistency model

The local replica manager is implemented by a combination of middleware (the DSM runtime layer in each process) and the kernel. It is usual for middleware to perform the majority of DSM processing. Even in a page-based DSM implementation, the kernel usually provides only basic page mapping, page-fault handling and communication mechanisms and middleware is responsible for implementing the page-sharing policies. If DSM segments are persistent, then one or more storage servers (for example, file servers) will also act as replica managers.

Two processes accessing shared variables



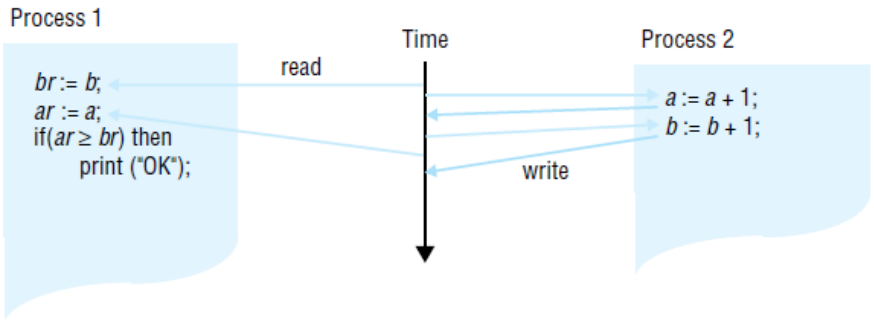
Sequential consistency

A DSM system is said to be sequentially consistent if *for any execution* there is some interleaving of the series of operations issued by all the processes that satisfies the following two criteria:

SC1: The interleaved sequence of operations is such that if R(x) a occurs in the sequence, then either the last write operation that occurs before it in the interleaved sequence is W(x) a, or no write operation occurs before it and *a* is the initial value of *x*.

SC2: The order of operations in the interleaving is consistent with the program order in which each individual client executed them.

Interleaving under sequential consistency



Coherence

Coherence is an example of a weaker form of consistency. Under coherence, every process agrees on the order of write operations to the same location, but they do not necessarily agree on the ordering of write operations to different locations. We can think of coherence as sequential consistency on a locationby- location basis. Coherent DSM can be implemented by taking a protocol for implementing sequential consistency and applying it separately to each unit of replicated data – for example, each page.

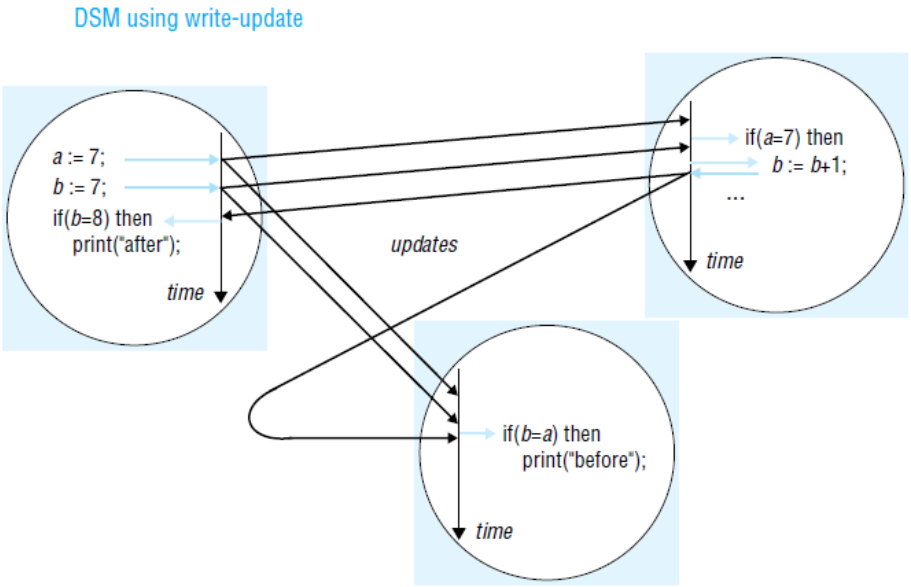
Weak consistency

This model exploits knowledge of synchronization operations in order to relax memory consistency, while appearing to the programmer to implement sequential consistency (at least, under certain conditions that are beyond the scope of this book). For example, if the programmer uses a lock to implement a critical section, then a DSM system can assume that no other process may access the data items accessed under mutual exclusion within it. It is therefore redundant for the DSM system to propagate updates to these items until the process leaves the critical section. While items are left with 'inconsistent' values some of the time, they are not accessed at those points; the execution appears to be sequentially consistent.

Update options

Two main implementation choices have been devised for propagating updates made by one process to the others: write-update and write-invalidate. These are applicable to a variety of DSM consistency models, including sequential consistency. In outline, the options are as follows:

Write-update: The updates made by a process are made locally and multicast to all other replica managers possessing a copy of the data item, which immediately modify the data read by local processes. Processes read the local copies of data items, without the need for communication. In addition to allowing multiple readers, several processes may write the same data item at the same time; this is known as multiple-reader/multiple-writer sharing.



Write-invalidate: This is commonly implemented in the form of multiple-reader/ single-writer sharing. At any time, a data item may either be accessed in read-only mode by one or more processes, or it may be read and written by a single process. An item that is currently accessed in read-only mode can be copied indefinitely to other processes. When a process attempts to write to it, a multicast message is first sent to all other copies to invalidate them and this is acknowledged before the write can take place; the other processes are thereby prevented from reading stale data (that is, data that are not up to date). Any processes attempting to access the data item are blocked if a writer exists.

Granularity

An issue that is related to the structure of DSM is the granularity of sharing. Conceptually, all processes share the entire contents of a DSM. As programs sharing DSM execute, however, only certain parts of the data are actually shared and then only for certain times during the execution. It would clearly be very wasteful for the DSM implementation always to transmit the entire contents of DSM as processes access and update it.

Thrashing

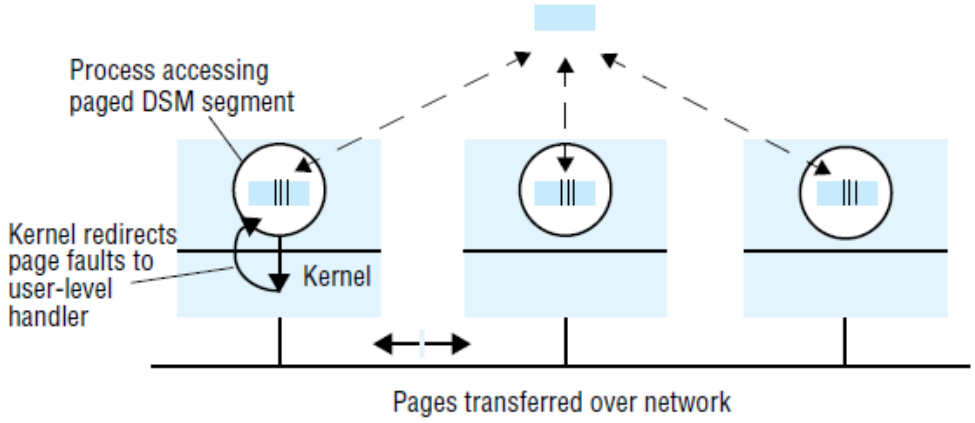
A potential problem with write-invalidate protocols is thrashing. Thrashing is said to occur where the DSM runtime spends an inordinate amount of time invalidating and transferring shared data compared with the time spent by application processes doing useful work. It occurs when several processes compete for the same data item, or for falsely shared data items.

SEQUENTIAL CONSISTENCY AND IVY CASE STUDY

The system model

The basic model to be considered is one in which a collection of processes shares a segment of DSM. The segment is mapped to the same range of addresses in each process, so that meaningful pointer values can be stored in the segment. The processes execute at computers equipped with a paged memory management unit. We shall assume that there is only one process per computer that accesses the DSM segment. There may in reality be several such processes at a computer. However, these could then share DSM pages directly (the same page frame can be used in the page tables used by the different processes). The only complication would be to coordinate fetching and propagating updates to a page when two or more local processes access it. This description ignores such details.

System model for page-based DSM



Paging is transparent to the application components within processes; they can logically both read and write any data in DSM. However, the DSM runtime restricts page access permissions in order to maintain sequential consistency when processing reads and writes. Paged memory management units allow the access permissions to a data page to be set to none, read-only or read-write.

The problem of write-update

The previous section outlined the general implementation alternatives of write-update and write-invalidation. In practice, if the DSM is page-based, then write-update is used only if writes can

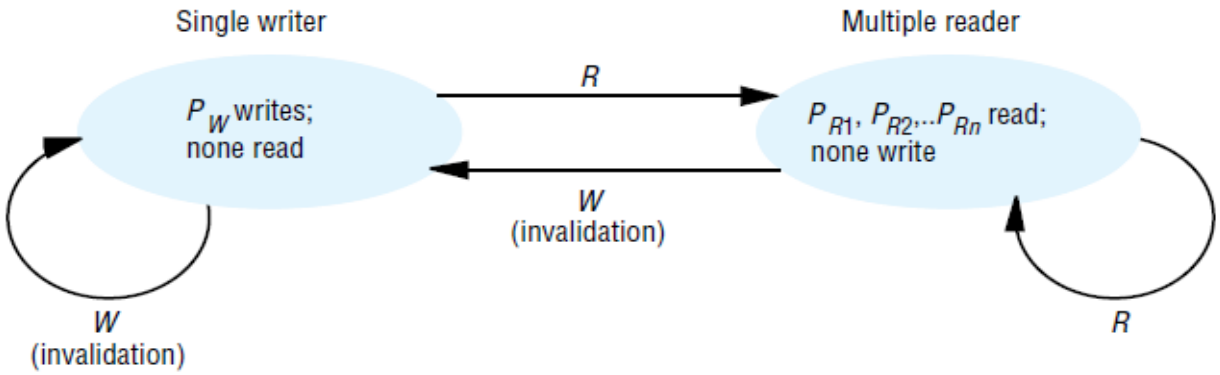
be buffered. This is because standard page-fault handling is unsuited to the task of processing every single write update to a page.

Write invalidation

Invalidation-based algorithms use page protection to enforce consistent data sharing. When a process is updating a page, it has read and write permissions locally; all other processes have no access permissions to the page. When one or more processes are reading the page, they have read-only permission; all other processes have no access permissions (although they may acquire read permissions). No other combinations are possible.

- The page is transferred to P_w, if it does not already have an up-to-date read-only copy.
- All other copies are invalidated: the page permissions are set to no access at all members of *copyset*(*p*).
- *copyset*(*p*) := { P_w }.
- *owner*(*p*) := P_w.
- The DSM runtime layer in P_w places the page with read-write permissions at the appropriate location in its address space and restarts the faulting instruction.

8 State transitions under write-invalidation



Note: *R* = read fault occurs; *W* = write fault occurs.

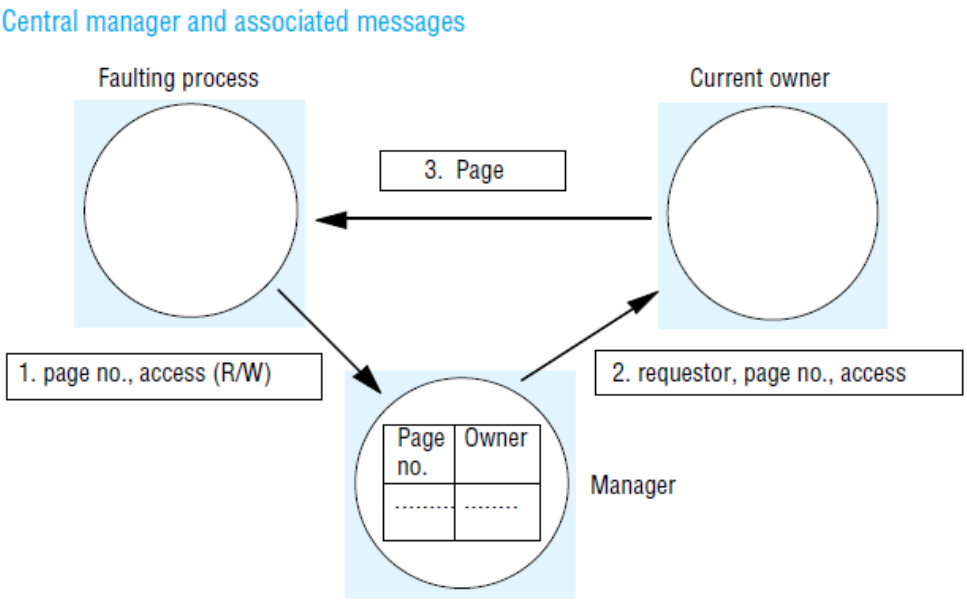
Invalidation protocols

Two important problems remain to be addressed in a protocol to implement the invalidation scheme:

1. How to locate *owner*(*p*) for a given page *p*.

2. Where to store *copyset*(*p*).

For Ivy, Li and Hudak [1989] describe several architectures and protocols that take varying approaches to these problems. The simplest we shall describe is their improved centralized manager algorithm. In it, a single server called a manager is used to store the location (transport address) of *owner*(*p*) for every page *p*. The manager could be one of the processes running the application, or it could be any other process. In this algorithm, the set *copyset*(*p*) is stored at *owner*(*p*). That is, the identifiers and transport addresses of the members of *copyset*(*p*) are stored.



Using multicast to locate the owner

Multicast can be used to eliminate the manager completely. When a process faults, it multicasts its page request to all the other processes. Only the process that owns the page replies. Care must be taken to ensure correct behaviour if two clients request the same page at more or less the same time: each client must obtain the page eventually, even if its request is multicast during transfer of ownership.

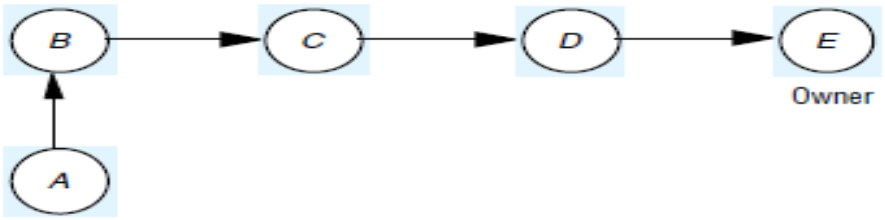
A dynamic distributed manager algorithm

The owner of a page is located by following chains of hints that are set up as ownership of the page is transferred from computer to computer. The length of the chain – that is, the number of forwarding messages necessary to locate the owner – threatens to increase indefinitely. The algorithm overcomes this by updating the hints as more upto- date values become available. Hints are updated and requests are forwarded as follows:

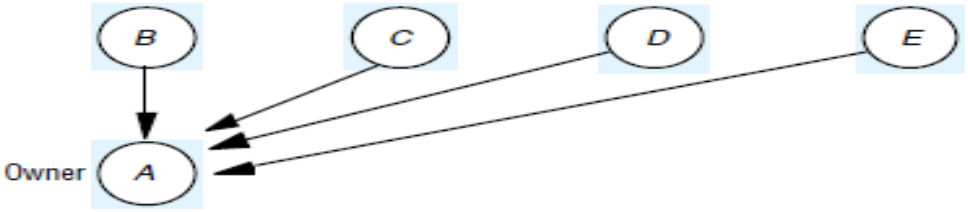
- When a process transfers ownership of page *p* to another process, it updates *probOwner*(*p*) to be the recipient.
- When a process handles an invalidation request for a page *p*, it updates *probOwner*(*p*) to be the requester.
- When a process that has requested read access to a page *p* receives it, it updates *probOwner*(*p*) to be the provider.
- When a process receives a request for a page *p* that it does not own, it forwards the request to *probOwner*(*p*) and resets *probOwner*(*p*) to be the requester.

The first three updates follow simply from the protocol for transferring page ownership and providing read-only copies. The rationale for the update when forwarding requests is that, for write requests, the requester will soon be the owner, even though it is not currently. In fact, in Li and Hudak's algorithm, assumed here, the *probOwner* update is made whether the request is for read access or write access.

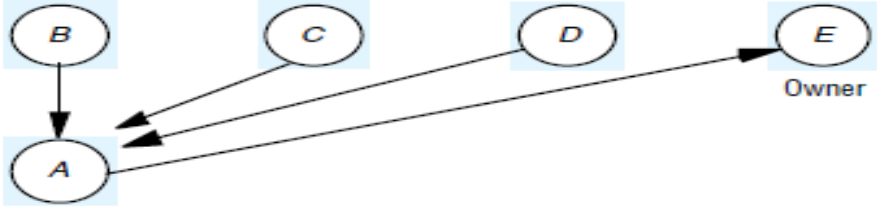
Updating *probOwner* pointers



(a) *probOwner* pointers just before process *A* takes a page fault for a page owned by *E*



(b) Write fault: *probOwner* pointers after *A*'s write request is forwarded



(c) Read fault: *probOwner* pointers after *A*'s read request is forwarded

Thrashing

It can be argued that it is the programmer's responsibility to avoid thrashing. The programmer could annotate data items in order to assist the DSM runtime in minimizing page copying and ownership transfers. The latter approach is discussed in the next section in the context of the Munin DSM system.

RELEASE CONSISTENCY AND MUNIN CASE STUDY

Release consistency was introduced with the Dash multiprocessor, which implements DSM in hardware, primarily using a write-invalidation protocol [Lenoski *etal*. 1992]. Munin and Treadmarks [Keleher *et al.* 1992] have adopted a software implementation of it. Release consistency is weaker than sequential consistency and cheaper to implement, but it has reasonable semantics that are tractable to programmers.

The idea of release consistency is to reduce DSM overheads by exploiting the fact that programmers use synchronization objects such as semaphores, locks and barriers. A DSM implementation can use knowledge of accesses to these objects to allow memory to become inconsistent at certain points, while the use of synchronization objects nonetheless preserves application-level consistency.

Memory accesses

In order to understand release consistency – or any other memory model that takes synchronization into account – we begin by categorizing memory accesses according to their role, if any, in synchronization. Furthermore, we shall discuss how memory accesses may be performed asynchronously to gain performance and give a simple operational model of how memory accesses take effect.

As we said above, DSM implementations on general-purpose distributed systems may use message passing rather than shared variables to implement synchronization, for reasons of efficiency.

acquireLock(var int lock)*: // *lock is passed by-reference

***while* (*testAndSet*(*lock*) = 1)**

***skip*;**

releaseLock(var int lock)*: // *lock is passed by-reference

***lock* := 0;**

Types of memory access

The main distinction is between *competing* accesses and *noncompeting* (*ordinary*) accesses. Two accesses are competing if:

- they may occur concurrently (there is no enforced ordering between them) and
- at least one is a *write*.

So two *read* operations can never be competing; a *read* and a *write* to the same location made by two processes that synchronize between the operations (and so order them) are non-competing.

We further divide competing accesses into *synchronization* and *nonsynchronization* accesses:

- synchronization accesses are *read* or *write* operations that contribute to synchronization;
- non-synchronization accesses are *read* or *write* operations that are concurrent but that do not contribute to synchronization.

Performing asynchronous operations

In view of the asynchronous operation that we have outlined, we distinguish between the point at which a *read* or *write* operation is issued – when the process first commences execution of the operation – and the point when the instruction is *performed* or completed.

We shall assume that our DSM is at least coherent. It means that every process agrees on the order of *write* operations to the same location. Given this assumption, we may speak unambiguously of the ordering of *write* operations to a given location.

Release consistency

The requirements that we wish to meet are:

- to preserve the synchronization semantics of objects such as locks and barriers;
- to gain performance, we allow a degree of asynchronicity for memory operations;
- to constrain the overlap between memory accesses in order to guarantee executions that provide the equivalent of sequential consistency.

Munin

The Munin DSM design [Carter *et al*. 1991] attempts to improve the efficiency of DSM by implementing the release consistency model. Furthermore, Munin allows programmers to annotate their data items according to the way in which they are shared, so that optimizations can be made in the update options selected for maintaining consistency. It is implemented upon the V

kernel [Cheriton and Zwaenepoel 1985], which was one of the first kernels to allow user-level threads to handle page faults and manipulate page tables.

The following points apply to Munin's implementation of release consistency:

- Munin sends update or invalidation information as soon as a lock is released.
- The programmer can make annotations that associate a lock with particular data items. In this case, the DSM runtime can propagate relevant updates in the same message that transfers the lock to a waiting process – ensuring that the lock's recipient has copies of the data it needs before it accesses them.

Sharing annotations

Munin implements a variety of consistency protocols, which are applied at the granularity of individual data items. The protocols are parameterized according to the following options:

- whether to use a write-update or write-invalidate protocol;
- whether several replicas of a modifiable data item may exist simultaneously;
- whether or not to delay updates or invalidations (for example, under release consistency);
- whether the item has a fixed owner, to which all updates must be sent;
- whether the same data item may be modified concurrently by several writers;
- whether the data item is shared by a fixed set of processes;
- whether the data item may be modified.

Read-only: No updates may be made after initialization and the item may be freely copied.

Migratory: Processes typically take turns in making several accesses to the item, at least one of which is an update. For example, the item might be accessed within a critical section. Munin always gives both read and write access together to such an object, even when a process takes a read fault. This saves subsequent write-fault processing.

Write-shared: Several processes update the same data item (for example, an array) concurrently, but this annotation is a declaration from the programmer that the processes do not update the same parts of it. This means that Munin can avoid false sharing but must propagate only those words in the data item that are actually updated at each process. To do this, Munin makes a copy of a page (inside a write-fault handler) just before it is updated locally. Only the differences between the two versions are sent in an update.

Producer-consumer: The data object is shared by a fixed set of processes, only one of which updates it. As we explained when discussing thrashing above, a writeupdate protocol is most suitable here. Moreover, updates may be delayed under the model of release consistency, assuming that the processes use locks to synchronize their accesses.

Reduction: The data item is always modified by being locked, read, updated and unlocked. An example of this is a global minimum in a parallel computation, which must be fetched and modified atomically if it is greater than the local minimum. These items are stored at a fixed owner. Updates are sent to the owner, which propagates them.

Result: Several processes update different words within the data item; a single process reads the whole item. For example, different 'worker' processes might fill in different elements of an array, which is then processed by a 'master' process. The point here is that the updates need only be propagated to the master and not to the workers (as would occur under the 'write-shared' annotation just described).

Conventional: The data item is managed under an invalidation protocol similar to that described in the previous section. No process may therefore read a stale version of the data item.

OTHER CONSISTENCY MODELS

Models of memory consistency can be divided into *uniform models*, which do not distinguish between types of memory access, and *hybrid models*, which do distinguish between ordinary and synchronization accesses (as well as other types of access).

Other uniform consistency models include:

Causal consistency: Reads and writes may be related by the happened-before relationship . This is defined to hold between memory operations when either (a) they are made by the same process; (b) a process reads a value written by another process; or (c) there exists a sequence of such operations linking the two operations. The model's constraint is that the value returned by a read must be consistent with the happened-before relationship.

Processor consistency: The memory is both coherent and adheres to the pipelined RAM model (see below). The simplest way to think of processor consistency is that the memory is coherent and that all processes agree on the ordering of any two write accesses made by the same process – that is, they agree with its program order.

Pipelined RAM: All processors agree on the order of writes issued by any given processor

In addition to release consistency, hybrid models include:

Entry consistency: Entry consistency was proposed for the Midway DSM system. In this model, every shared variable is bound to a synchronization object such as a lock, which governs access to that variable. Any process that first acquires the lock is guaranteed to read the latest value of the variable. A process wishing to write the variable must first obtain the corresponding lock in 'exclusive' mode – making it the only process able to access the variable.

Several processes may read the variable concurrently by holding the lock in nonexclusive mode. Midway avoids the tendency to false sharing in release consistency, but at the expense of increased programming complexity.

Scope consistency: This memory model [Iftode *et al.* 1996] attempts to simplify the programming model of entry consistency. In scope consistency, variables are associated with synchronization objects largely automatically instead of relying on the programmer to associate locks with variables explicitly. For example, the system can monitor which variables are updated in a critical section.

Weak consistency: Weak consistency [Dubois *et al.* 1988] does not distinguish between *acquire* and *release* synchronization accesses. One of its guarantees is that all previous ordinary accesses complete before *either* type of synchronization access completes.

Common Object Request Broker Architecture (CORBA)

CORBA is a middeware design that allows application programs to communicate with one another irrespective of their programming languages, their hardware and software platforms, the networks they communicate over and their implementors.

Applications are built from CORBA objects, which implement interfaces defined in CORBA's interface definition language, IDL. Clients access the methods in the IDL interfaces of CORBA objects by means of RMI. The middleware component that supports RMI is called the Object Request Broker or ORB.

Introduction

The OMG (Object Management Group) was formed in 1989 with a view to encouraging the adoption of distributed object systems in order to gain the benefits of object-oriented

programming for software development and to make use of distributed systems, which were becoming widespread. To achieve its aims, the OMG advocated the use of open systems based on standard object-oriented interfaces. These systems would be built from heterogeneous hardware, computer networks, operating systems and programming languages.

An important motivation was to allow distributed objects to be implemented in any programming language and to be able to communicate with one another. They therefore designed an interface language that was independent of any specific implementation language.

They introduced a metaphor, the *object request broker*(or ORB), whose role is to help a client to invoke a method on an object. This role involves locating the object, activating the object if necessary and then communicating the client's request to the object, which carries it out and replies.

In 1991, a specification for an object request broker architecture known as CORBA (Common Object Request Broker Architecture) was agreed by a group of companies. This was followed in 1996 by the CORBA 2.0 specification, which defined standards enabling implementations made by different developers to communicate with one another. These standards are called the General Inter-ORB protocol or GIOP. It is intended that GIOP can be implemented over any transport layer with connections. The implementation of GIOP for the Internet uses the TCP protocol and is called the Internet Inter-ORB Protocol or IIOP [OMG 2004a]. CORBA 3 first appeared in late 1999 and a component model has been added recently.

The main components of CORBA's language-independent RMI framework are the following:

- An interface definition language known as IDL,
- The GIOP defines an external data representation, called CDR. It also defines specific formats for the messages in a request-reply protocol. In addition to request and reply messages, it specifies messages for enquiring about the location of an object, for cancelling requests and for reporting errors.
- The IIOP, an implementation of GIOP defines a standard form for remote object references,

CORBA RMI

Programming in a multi-language RMI system such as CORBA RMI requires more of the programmer than programming in a single-language RMI system such as Java RMI.

The following new concepts need to be learned:

- the object model offered by CORBA;
- the interface definition language and its mapping onto the implementation language.

CORBA's object model

The CORBA object model is similar to the one described in , but clients are not necessarily objects – a client can be any program that sends request messages to remote objects and receives replies. The term *CORBA object* is used to refer to remote objects. Thus, a CORBA object implements an IDL interface, has a remote object reference and is able to respond to invocations of methods in its IDL interface. A CORBA object can be implemented by a language that is not objectoriented, for example without the concept of class. Since implementation languages will have different notions of class or even none at all, the class concept does not exist in CORBA. Therefore classes cannot be defined in CORBA IDL, which means that instances of classes cannot be passed as arguments.

CORBA IDL

These are preceded by definitions of two *structs*, which are used as parameter types in defining the methods. Note in particular that *GraphicalObject* is defined as a *struct* , whereas it was a class in the Java RMI example. A component whose type is a *struct* has a set of fields containing values of various types like the instance variables of an object, but it has no methods.

Parameters and results in CORBA IDL:

Each parameter is marked as being for input or output or both, using the keywords *in* , *out* or *inout* illustrates a simple example of the use of those keywords

IDL interfaces *Shape* and *ShapeList*

```
struct Rectangle{                                                    1
    long width;
    long height;
    long x;
    long y;
} ;
struct GraphicalObject {                                             2
    string type;
    Rectangle enclosing;
    boolean isFilled;
};
interface Shape {                                                    3
    long getVersion() ;
    GraphicalObject getAllState() ;    // returns state of the GraphicalObject
};
typedef sequence <Shape, 100> All;                                   4
interface ShapeList {                                                5
    exception FullException{ };                                      6
    Shape newShape(in GraphicalObject g) raises (FullException);     7
    All allShapes();              // returns sequence of remote object references  8
    long getVersion() ;
};
```

The semantics of parameter passing are as follows:

Passing CORBA objects:

Any parameter whose type is specified by the name of an IDL interface, such as the return value *Shape* in line 7, is a reference to a CORBA object and the value of a remote object reference is passed.

Passing CORBA primitive and constructed types:

Arguments of primitive and constructed types are copied and passed by value. On arrival, a new value is created in the recipient's process. For example, the *struct GraphicalObject* passed as argument (in line 7) produces a new copy of this *struct* at the server.

Type *Object* :

Object is the name of a type whose values are remote object references. It is effectively a common supertype of all of IDL interface types such as *Shape* and *ShapeList*.

Exceptions in CORBA IDL:

CORBA IDL allows exceptions to be defined in interfaces and thrown by their methods. To illustrate this point, we have defined our list of shapes in the server as a sequence of a fixed length (line 4) and have defined *FullException* (line 6), which is thrown by the method *newShape* (line 7) if the client attempts to add a shape when the sequence is full.

Invocation semantics:

Remote invocation in CORBA has *at-most-once* call semantics as the default. However, IDL may specify that the invocation of a particular method has *maybe* semantics by using the *oneway* keyword. The client does not block on *oneway* requests, which can be used only for methods without results.

The CORBA Naming service

It is a binder that provides operations including *rebind* for servers to register the remote object references of CORBA objects by name and *resolve* for clients to look them up by name. The names are structured in a hierarchic fashion, and each name in a path is inside a structure called a *NameComponent* . This makes access in a simple example seem rather complex.

CORBA pseudo objects

Implementations of CORBA provide interfaces to the functionality of the ORB that programmers need to use. In particular, they include interfaces to two of the components in the *ORB cor*e and the *Object Adaptor*

CORBA client and server example

This is followed by a discussion of callbacks in CORBA. We use Java as the client and server languages, but the approach is similar for other languages. The interface compiler *idlj* can be applied to the CORBA interfaces to generate the following items:

Java interfaces generated by *idlj* from CORBA interface *ShapeList*.

```
public interface ShapeListOperations {
   Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException;
   Shape[] allShapes();
   int getVersion();
}
```

```
public interface ShapeList extends ShapeListOperations, org.omg.CORBA.Object,
      org.omg.CORBA.portable.IDLEntity { }
```

.

- The equivalent Java interfaces – two per IDL interface. The name of the first Java interface ends in *Operations* – this interface just defines the operations in the IDL interface. The Java second interface has the same name as the IDL interface and implements the operations in the first interface as well as those in an interface suitable for a CORBA object.
- The server skeletons for each *idl* interface. The names of skeleton classes end in *POA* , for example *ShapeListPOA.*
- The proxy classes or client stubs, one for each IDL interface. The names of these classes end in *Stub* , for example *_ShapeListStub*\
- A Java class to correspond to each of the *structs* defined with the IDL interfaces. In our example, classes *Rectangle* and *GraphicalObject* are generated. Each of these classes contains a declaration of one instance variable for each field in the corresponding *struct* and a pair of constructors, but no other methods.
- Classes called helpers and holders, one for each of the types defined in the IDL interface. A helper class contains the *narrow* method, which is used to cast down from a given object reference to the class to which it belongs, which is lower down the class hierarchy. For example, the *narrow* method in *ShapeHelper* casts down to class *Shape* . The holder classes deal with *out* and *inout* arguments, which cannot be mapped directly onto Java.

Server program

The server program should contain implementations of one or more IDL interfaces. For a server written in an object-oriented language such as Java or C++, these implementations are implemented as servant classes. CORBA objects are instances of servant classes.

When a server creates an instance of a servant class, it must register it with the POA, which makes the instance into a CORBA object and gives it a remote object reference. Unless this is done, the CORBA object will not be able to receive remote invocations. Readers who studied Chapter 5 carefully may realize that registering the object with the POA causes it to be recorded in the CORBA equivalent of the remote object table.

ShapeListServant class of the Java server program for CORBA interface *ShapeList*

```
import org.omg.CORBA.*;
import org.omg.PortableServer.POA;
class ShapeListServant extends ShapeListPOA {
    private POA theRootpoa;
    private Shape theList[];
    private int version;
     private static int n=0;
    public ShapeListServant(POA rootpoa){
        theRootpoa = rootpoa;
        // initialize the other instance variables
    }
    public Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException {1
        version++;
        Shape s = null;
        ShapeServant shapeRef = new ShapeServant( g, version);
        try {
            org.omg.CORBA.Object ref = theRoopoa.servant_to_reference(shapeRef);  2
            s = ShapeHelper.narrow(ref);
        } catch (Exception e) {}
         if(n >=100) throw new ShapeListPackage.FullException();
        theList[n++] = s;
        return s;
    }
    public  Shape[] allShapes(){ ... }
    public int getVersion() { ... }
}
```

Java class *ShapeListServer*

```
import org.omg.CosNaming.*;
 import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
public class ShapeListServer {
      public static void main(String args[]) {
         try{
            ORB orb = ORB.init(args, null);                                    1
            POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));   2
            rootpoa.the_POAManager().activate();                               3
            ShapeListServant SLSRef = new ShapeListServant(rootpoa);           4
            org.omg.CORBA.Object ref = rootpoa.servant_to_reference(SLSRef);   5
            ShapeList SLRef = ShapeListHelper.narrow(ref);
            org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);          6
            NameComponent nc = new NameComponent("ShapeList", "");             7
            NameComponent path[] = {nc};                                       8
            ncRef.rebind(path, SLRef);                                         9
            orb.run();                                                         10
         } catch (Exception e) { ... }
      }
}
```

The client program

It creates and initializes an ORB (line 1), then contacts the Naming Service to get a reference to the remote *ShapeList* object by using its *resolve* method (line 2). After that it invokes its method *allShapes* (line 3) to obtain a sequence of remote object references to all the *Shapes* currently held at the server. It then invokes the *getAllState* method (line 4), giving as argument the first remote object reference in the sequence returned; the result is supplied as an instance of the *GraphicalObject* class.

Java client program for CORBA interfaces *Shape* and *ShapeList*

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
public class ShapeListClient{
    public static void main(String args[]) {
        try{
            ORB orb = ORB.init(args, null);                          1
            org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent("ShapeList", "");
            NameComponent path [] = { nc };
            ShapeList shapeListRef =
            ShapeListHelper.narrow(ncRef.resolve(path));             2
            Shape[] sList = shapeListRef.allShapes();                3
            GraphicalObject g = sList[0].getAllState();              4
        } catch(org.omg.CORBA.SystemException e) {...}
    }
}
```

Callbacks

Callbacks can be implemented in CORBA in a manner similar to the one described for Java RMI For example, the *WhiteboardCallback* interface may be defined as follows:

```
interface WhiteboardCallback {
oneway void callback(in int version);
};
```

This interface is implemented as a CORBA object by the client, enabling the server to send the client a version number whenever new objects are added. But before the server can do this, the client needs to inform the server of the remote object reference of its object. To make this possible, the *ShapeList* interface requires additional methods such as *register* and *deregister*, as follows:

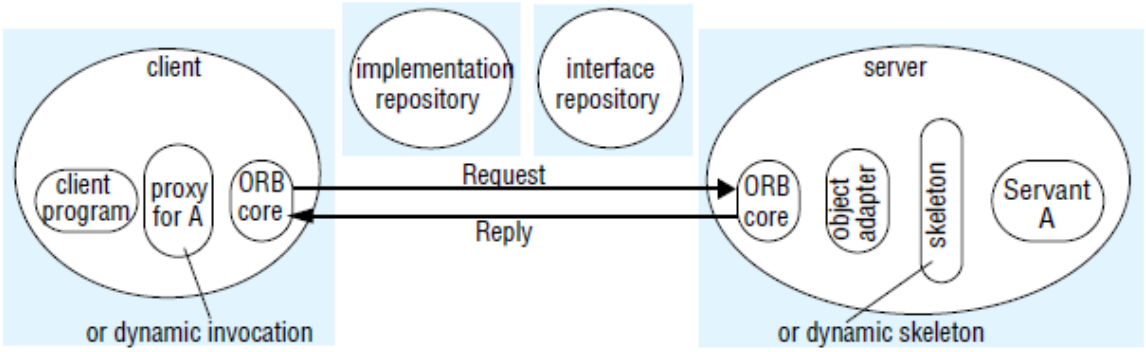
```
int register(in WhiteboardCallback callback);
void deregister(in int callbackId);
```

After a client has obtained a reference to the *ShapeList* object and created an instance of *WhiteboardCallback*, it uses the *register* method of *ShapeList* to inform the server that it is interested in receiving callbacks. The *ShapeList* object in the server is responsible for keeping a list of interested clients and notifying all of them each time its version number increases when a new object is added.

The architecture of CORBA

The architecture is designed to support the role of an object request broker that enables clients to invoke methods in remote objects, where both clients and servers can be implemented in a variety of programming languages. The main components of the CORBA architecture are illustrated in Figure

The main components of the CORBA architecture



CORBA provides for both static and dynamic invocations. Static invocations are used when the remote interface of the CORBA object is known at compile time, enabling client stubs and server skeletons to be used. If the remote interface is not known at compile time, dynamic invocation must be used. Most programmers prefer to use static invocation because it provides a more natural programming model.

ORB core ◊ The role of the ORB core is similar to that of the communication module . In addition, an ORB core provides an interface that includes the following:

- operations enabling it to be started and stopped;
- operations to convert between remote object references and strings;
- operations to provide argument lists for requests using dynamic invocation.

Object adapter

The role of an *object adapter* is to bridge the gap between CORBA objects with IDL interfaces and the programming language interfaces of the corresponding servant classes. This role also includes that of the remote reference and dispatcher modules. An object adapter has the following tasks:

- it creates remote object references for CORBA objects;

- it dispatches each RMI via a skeleton to the appropriate servant;
- it activates and deactivates servants.

An object adapter gives each CORBA object a unique *object name*, which forms part of its remote object reference. The same name is used each time an object is activated. The object name may be specified by the application program or generated by the object adapter. Each CORBA object is registered with its object adapter, which may keep a remote object table that maps the names of CORBA objects to their servants.

Portable object adapter

The CORBA 2.2 standard for object adapters is called the Portable Object Adapter. It is called portable because it allows applications and servants to be run on ORBs produced by different developers [Vinoski 1998]. This is achieved by means of the standardization of the skeleton classes and of the interactions between the POA and the servants. The POA supports CORBA objects with two different sorts of lifetimes:

- those whose lifetimes are restricted to that of the process their servants are instantiated in;
- those whose lifetimes can span the instantiations of servants in multiple processes.

Skeletons

Skeleton classes are generated in the language of the server by an IDL compiler. As before, remote method invocations are dispatched via the appropriate skeleton to a particular servant, and the skeleton unmarshals the arguments in request messages and marshals exceptions and results in reply messages.

Client stubs/proxies

These are in the client language. The class of a proxy (for object oriented languages) or a set of stub procedures (for procedural languages) is generated from an IDL interface by an IDL compiler for the client language. As before, the client stubs/proxies marshal the arguments in invocation requests and unmarshal exceptions and results in replies.

Implementation repository

- An implementation repository is responsible for activating registered servers on demand and for locating servers that are currently running. The object adapter name is used to refer to servers when registering and activating them.
- An implementation repository stores a mapping from the names of object adapters to the pathnames of files containing object implementations.

- Object implementations and object adapter names are generally registered with the implementation repository when server programs are installed.
- When object implementations are activated in servers, the hostname and port number of the server are added to the mapping.

Interface repository

The role of the interface repository is to provide information about registered IDL interfaces to clients and servers that require it. For an interface of a given type it can supply the names of the methods and for each method, the names and types of the arguments and exceptions. Thus, the interface repository adds a facility for reflection to CORBA

Dynamic invocation interface

The dynamic invocation interface allows clients to make dynamic invocations on remote CORBA objects. It is used when it is not practical to employ proxies. The client can obtain from the interface repository the necessary information about the methods available for a given CORBA object. The client may use this information to construct an invocation with suitable arguments and send it to the server.

Dynamic skeletons

If a server uses dynamic skeletons, then it can accept invocations on the interface of a CORBA object for which it has no skeleton. When a dynamic skeleton receives an invocation, it inspects the contents of the request to discover its target object, the method to be invoked and the arguments. It then invokes the target.

Legacy code

The term *legacy code* refers to existing code that was not designed with distributed objects in mind. A piece of legacy code may be made into a CORBA object by defining an IDL interface for it and providing an implementation of an appropriate object adapter and the necessary skeletons.

CORBA Interface Definition Language

The CORBA Interface Definition Language, IDL, provides facilities for defining modules, interfaces, types, attributes and method signatures. IDL has the same lexical rules as C++ but has additional keywords to support distribution, for example *interface*, *any*, *attribute*, *in*, *out*, *inout*, *readonly*, *raises*. It also allows standard C++ preprocessing facilities.

IDL Modules

The module construct allows interfaces and other IDL type definitions to be grouped in logical units. A *module* defines a naming scope, which prevents names defined within a module clashing with names defined outside it.

IDL module *Whiteboard*.

```
module Whiteboard {
    struct Rectangle{
    ...} ;
    struct GraphicalObject {
    ...};
    interface Shape {
    ...};
    typedef sequence <Shape, 100> All;
    interface ShapeList {
    ...};
};
```

IDL interface

An IDL interface describes the methods that are available in CORBA objects that implement that interface. Clients of a CORBA object may be developed just from the knowledge of its IDL interface.

IDL methods

The general form of a method signature is:

[oneway] <return_type> <method_name> (parameter1,..., parameterL)

[raises (except1,..., exceptN)] [context (name1,..., nameM)]

where the expressions in square brackets are optional. For an example of a method signature that contains only the required parts, consider:

void getPerson(in string name, out Person p);

IDL types

IDL supports fifteen primitive types, which include short (16-bit), long (32- bit), unsigned short, unsigned long, float (32-bit), double (64-bit), char, Boolean (TRUE, FALSE), octet (8-bit), and any (which can represent any primitive or constructed type).

Attributes

IDL interfaces can have attributes as well as methods. Attributes are like public class fields in Java. Attributes may be defined as *readonly* where appropriate. The attributes are private to CORBA objects, but for each attribute declared, a pair of accessor methods is generated automatically by the IDL compiler, one to retrieve the value of the attribute and the other to set it. For *readonly* attributes, only the getter method is provided. For example, the *PersonList* interface defined in Figure 5.2 includes the following definition of an attribute: *readonly attribute string listname;*

Inheritance

IDL interfaces may be extended. For example, if interface *B* extends interface *A*, this means that it may add new types, constants, exceptions, methods and attributes to those of *A*. An extended interface can redefine types, constants and exceptions, but is not allowed to redefine methods. A value of an extended type is valid as the value of a parameter or result of the parent type. For example, the type *B* is valid as the value of a parameter or result of the type *A*.

interface A { };

interface B: A{ };

interface C {};

interface Z : B, C {};

IDL constructed types.

Type	*Examples*	*Use*
sequence	*typedef sequence <Shape, 100> All;* *typedef sequence <Shape> All* bounded and unbounded sequences of *Shapes*	Defines a type for a variable-length sequence of elements of a specified IDL type. An upper bound on the length may be specified.
string	*string name;* *typedef string<8> SmallString;* unbounded and bounded sequences of characters	Defines a sequences of characters, terminated by the null character. An upper bound on the length may be specified.
array	*typedef octet uniqueId[12];* *typedef GraphicalObject GO[10][8]*	Defines a type for a multi-dimensional fixed-length sequence of elements of a specified IDL type.
record	*struct GraphicalObject {* *string type;* *Rectangle enclosing;* *boolean isFilled;* *};*	Defines a type for a record containing a group of related entities. *Structs* are passed by value in arguments and results.
enumerated	*enum Rand* *(Exp, Number, Name);*	The enumerated type in IDL maps a type name onto a small set of integer values.
union	*union Exp switch (Rand) {* *case Exp: string vote;* *case Number: long n;* *case Name: string s;* *};*	The IDL discriminated union allows one of a given set of types to be passed as an argument. The header is parameterized by an *enum*, which specifies which member is in use.

CORBA SERVICES

CORBA includes specifications for services that may be required by distributed objects. In particular, the Naming Service is an essential addition to any ORB. The CORBA services include the following:

- *Naming Service*:
- *Event Service and Notification Service*:
- *Security service*:
- *Trading service*:

In contrast to the Naming Service which allows CORBA objects to be located by name, the Trading Service [OMG 2000a] allows them to be located by attribute – that is, it is a directory service. Its database contains a mapping from service types and their associated attributes onto remote object references of CORBA objects. The service type is a name, and each attribute is a name-value pair. Clients make queries by specifying the type of service required, together with other arguments specifying constraints on the values of attributes, and preferences for the order in which to receive matching offers. Trading servers can form federations in which they not only use their own databases but also perform queries on behalf of one anothers' clients.

- *Transaction service and concurrency control service*:

The object transaction service [OMG 2003] allows distributed CORBA objects to participate in either flat or nested transactions. The client specifies a transaction as a sequence of RMI calls, which are introduced by *begin* and terminated by *commit* or *rollback* (*abort*). The ORB attaches a transaction identifier to each remote invocation and deals with *begin*, *commit* and *rollback* requests. Clients can also suspend and resume transactions. The transaction service carries out a two-phase commit protocol. The concurrency control service [OMG 2000b] uses locks to apply concurrency control to the access of CORBA objects. It may be used from within transactions or independently.

- *Persistent state service*:

An persistent objects can be implemented by storing them in a passive form in a persistent object store while they are not in use and activating them when they are needed. Although ORBs activate CORBA objects with persistent object references, getting their implementations from the

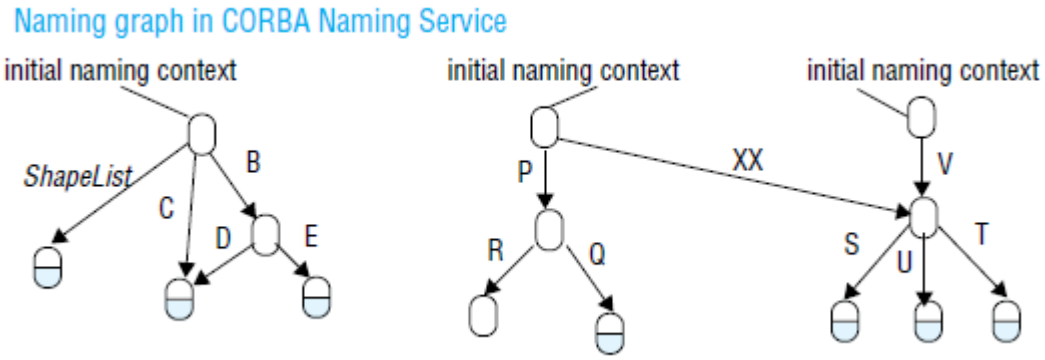
implementation repository, they are not responsible for saving and restoring the state of CORBA objects.

- *Life cycle service*

The life cycle service defines conventions for creating, deleting, copying and moving CORBA objects. It specifies how clients can use factories to create objects in particular locations, allowing persistent storage to be used if required. It defines an interface that allows clients to delete CORBA objects or to move or copy them to a specified location.

CORBA Naming Service

The CORBA Naming Service is a sophisticated example of the binder described in Chapter 5. It allows names to be bound to the remote object references of CORBA objects within naming contexts.



a *naming context* is the scope within which a set of names applies – each of the names within a context must be unique. A name can be associated with either an object reference for a CORBA object in an application or with another context in the naming service.

The names used by the CORBA Naming Service are two-part names, called Name Components, each of which consists of two strings, one for the name and the other for the kind of the object. The kind field provides a single attribute that is intended for use by applications and may contain any useful descriptive information; it is not interpreted by the Naming Service.

Although CORBA objects are given hierarchic names by the Naming Service, these names cannot be expressed as pathnames like those of UNIX files.

Part of the CORBA Naming Service *NamingContext* interface in IDL

```
struct NameComponent { string id; string kind; };
typedef sequence <NameComponent> Name;
interface NamingContext {
    void bind (in Name n, in Object obj);
        binds the given name and remote object reference in my context.
    void unbind (in Name n);
        removes an existing binding with the given name.
    void bind_new_context(in Name n);
        creates a new naming context and binds it to a given name in my context.
    Object resolve (in Name n);
        looks up the name in my context and returns its remote object reference.
    void list (in unsigned long how_many, out BindingList bl, out BindingIterator bi);
        returns the names in the bindings in my context.
};
```

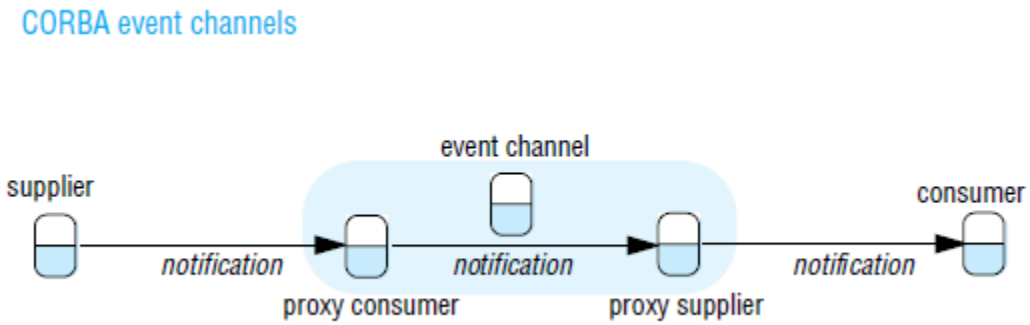
CORBA Event Service

The CORBA Event Service specification defines interfaces allowing objects of interest, called *suppliers*, to communicate notifications to subscribers, called *consumers*. The notifications are communicated as arguments or results of ordinary synchronous CORBA remote method invocations. Notifications may be propagated either by being *pushed* by the supplier to the consumer or *pulled* by the consumer from the supplier. In the first case, the consumers implement the *PushConsumer* interface which includes a method *push* that takes any CORBA data type as argument. Consumers register their remote object references with the suppliers. The supplier invokes the *push* method, passing a notification as argument. In the second case, the supplier implements the *PullSupplier* interface, which includes a method *pull* that receives any CORBA data type as its return value. Suppliers register their remote object references with the consumers. The consumers invoke the *pull* method and receive a notification as result.

The notification itself is transmitted as an argument or result whose type is *any*, which means that the objects exchanging notifications must have an agreement about the contents of notifications. Application programmers, however, may define their own IDL interfaces with notifications of any desired type.

Event channels are CORBA objects that may be used to allow multiple suppliers to communicate with multiple consumers in an asynchronous manner. An event channel acts as a

buffer between suppliers and consumers. It can also multicast the notifications to the consumers. Communication via an event channel may use either the push or pull style. The two styles may be mixed; for example, suppliers may push notifications to the channel and consumers may pull notifications from it.



CORBA Notification Service

The CORBA Notification Service extends the CORBA Event Service, retaining all of its features including event channels, event consumers and event suppliers. The event service provides no support for filtering events or for specifying delivery requirements. Without the use of filters, all the consumers attached to a channel have to receive the same notifications as one another. And without the ability to specify delivery requirements, all of the notifications sent via a channel are given the delivery guarantees built into the implementation.

The notification service adds the following new facilities:

- Notifications may be defined as data structures. This is an enhancement of the limited utility provided by notifications in the event service, whose type could only be either *any* or a type specified by the application programmer.
- Event consumers may use filters that specify exactly which events they are interested in. The filters may be attached to the proxies in a channel. The proxies will forward notifications to event consumers according to constraints specified in filters in terms of the contents of each notification.
- Event suppliers are provided with a means of discovering the events the consumers are interested in. This allows them to generate only those events that are required by the consumers.
- Event consumers can discover the event types offered by the suppliers on a channel, which enables them to subscribe to new events as they become available.

- It is possible to configure the properties of a channel, a proxy or a particular event. These properties include the reliability of event delivery, the priority of events, the ordering required (for example, FIFO or by priority) and the policy for discarding stored events.
- An event type repository is an optional extra. It will provide access to the structure of events, making it convenient to define filtering constraints.

A structured event consists of an event header and an event body. The following example illustrates the contents of the header:

domain type	*event type*	*event name*	*requirements*
"home"	"burglar alarm"	"21 Mar at 2pm"	"priority", 1000

The following example illustrates the information in the body of a structured event:

← *filterable part* →			
name, value	*name, value*	*name, value*	*remainder*
"bell", "ringing"	"door","open"	"cat", "outside"	

Filter objects are used by proxies in making decisions as to whether to forward each notification. A filter is designed as a collection of constraints, each of which is a data structure with two components:

- A list of data structures, each of which indicates an event type in terms of its domain name and event type, for example, *"home"*, *"burglar alarm"*. The list includes all of the event types to which the constraint should apply.
- A string containing a boolean expression involving the values of the event types listed above. For example:

  ("domain type" == "home" && "event type" == "burglar alarm") &&
  ("bell" != "ringing" !! "door" == "open")

CORBA Security Service

The CORBA Security Service [Blakley 1999, Baker 1997, OMG 2002b] includes the following:

- Authentication of principals (users and servers); generating credentials for principals (that is, certificates stating their rights); delegation of credentials is supported

- Access control can be applied to CORBA objects when they receive remote method invocations. Access rights may for example be specified in access control lists (ACLs).
- Security of communication between clients and objects, protecting messages for integrity and confidentiality.
- Auditing by servers of remote method invocations.
- Facilities for non-repudiation. When an object carries out a remote invocation on behalf of a principal, the server creates and stores credentials that prove that the invocation was done by that server on behalf of the requesting principal.

CORBA allows a variety of security policies to be specified according to requirements. A message-protection policy states whether client or server (or both) must be authenticated, and whether messages must be protected against disclosure and/or modification.

Access control takes into account that many applications have large numbers of users and even larger numbers of objects, each with its own set of methods. Users are supplied with a special type of credential called a *privilege* according to their roles.

Objects are grouped into *domains*. Each domain has a single access control policy specifying the access rights for users with particular privileges to objects within that domain. To allow for the unpredictable variety of methods, each method is classified in terms of one of four generic methods (*get*, *set*, *use* and *manage*). *Get* methods just return parts of the object state, *set* methods alter the object state, *use* methods cause the object to do some work, and *manage* methods perform special functions that are not intended to be available for general use. Since CORBA objects have a variety of different interfaces, the access rights must be specified for each new interface in terms of the above generic methods.

In its simplest form, security may be applied in a manner that is transparent to applications. It includes applying the required protection policy to remote method invocations, together with auditing. The security service allows users to acquire their individual credentials and privileges in return for supplying authentication data such as a password.