Chapter 7

Delta-4 Application Support Environment (Deltase)

The Delta-4 system architecture is intended for open, dependable, distributed systems. The purpose of Deltase, the *Delta-4 Application Support Environment*, is to provide a *homogeneous computing environment* for the support of distributed applications in Delta-4 systems.

This chapter covers the concepts of Deltase, and its generic functions and features. The actual functions and features in any particular implementation will be described in the literature related to that implementation. The prototype implementations within the Delta-4 project are described in the *Delta-4 Implementation Guide* [Delta-4 1990].

7.1. Purpose and Background

The decomposition of an application into a number of *language-level* program modules, which interact with one another by means of procedure calls, is a widely-used methodology for applications that are implemented as a single program and executed on a single machine. The aim of Deltase is to support this application structure in *distributed* systems; that is, to extend, to distributed fault-tolerant systems, an application structure that is widely-used within applications for execution on a single computer.

Deltase provides a uniform *virtual execution environment* for the execution of, and interactions between, *language-level* program modules (see figure 1). This virtual execution environment can be mapped onto a distributed system (or a part thereof), but conceals both the physical configuration of the distributed system and the *mapping* of the program modules onto the hosts.

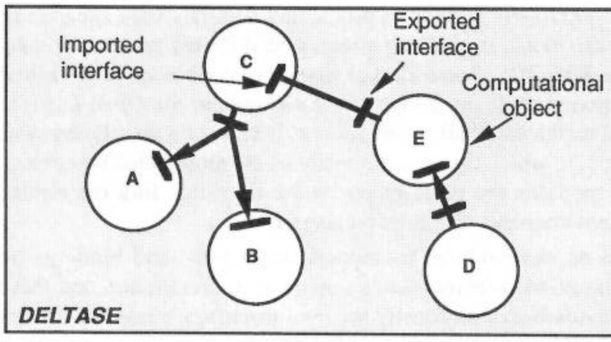


Fig. 1 - Programmer's View of Object-Based System

Deltase is *generic*, and many different realizations are possible. Each host machine that supports this virtual execution environment requires an implementation of Deltase, which would normally interface to the *Local Executive* (LEX), and support one of the languages already available for that host. The Delta-4 *Open System Architecture* (OSA) allows heterogeneous hosts within a single distributed system, with inter-working between the implementations of Deltase for the different hosts, LEXes and language systems.

Deltase is intended to allow migration to future technology, by permitting implementations for new hosts, LEXes, language systems, etc., as these become available.

While the Delta-4 system architecture is aimed primarily at large (distributed) systems, Deltase is intended to cover a wide range of systems, from a single machine (without fault-tolerance) up to large-scale Delta-4 systems. While this chapter is concerned mainly with the use of Deltase in distributed fault-tolerant systems, applications written for use with Deltase may equally be executed within a single machine with a suitable implementation of Deltase. (Since fault-tolerance and distribution are made *transparent* to the application programmer — (see §7.2), such applications can equally-well be executed on systems that do not provide such features.)

Both the Delta-4 *Open System Architecture* (OSA) and the Delta-4 *Extra Performance Architecture* (XPA) include Deltase; Deltase is mandatory for XPA, but is optional for OSA. The requirements for the support of fault-tolerance (in both OSA and XPA) and real-time behaviour (in XPA) impose particular constraints on Deltase.

7.1.1. Open Distributed Processing (ODP)

Deltase is based upon the concepts of the emerging work on *Open Distributed Processing* (ODP), and its related standardization; see [ECMA TR/49]. ODP is concerned with defining both a *generic architecture* for distributed processing systems, and the standards to support this generic architecture.

The ODP standardization work in the *International Standards Organization* (ISO) is concerned with the definition of a *Reference Model for ODP* (RM-ODP); this work is undertaken by ISO/IEC JTC1/SC21 WG7. This reference model is to ODP, what the *Open Systems Interconnection (OSI) Reference Model* was to OSI. The RM-ODP is concerned with architecture that is intended to be applicable to most kinds of (distributed) application.

The ODP standardization work is supported by the *European Computer Manufacturers Association* (ECMA); the technical group ECMA/TC32-TG2 is working on standards for a *Support Environment for Open Distributed Processing* (SE-ODP).

Distributed processing is defined as processing that may span separate computer address spaces, and *open distributed processing* concerns distributed processing that conforms to the requirements of the RM-ODP. Some kind of support environment, or infrastructure, is required to support distributed processing. A *Support Environment for Open Distributed Processing* (SE-ODP) is based on the application programmer's view of a distributed system. This view is illustrated in figure 1, in which the program modules are represented by circles, and interactions between program modules are represented by the lines that link the circles, within an all-pervading support environment (the outer rectangle).

An SE-ODP is an *environment* to support interactions and bindings between program modules. Each program module represents a separate address space, and these address spaces may be located in a number of physically separate machines; a number of address spaces may be co-located within a single machine. An SE-ODP facilitates the construction, operation and maintenance of distributed applications. Standardization of the SE-ODP is necessary where there is a requirement for heterogeneity and multi-vendor procurement.

An SE-ODP may be viewed as a means to provide interaction between distributed program modules, and nothing more. If provided in a sufficiently general way, this *one* function (interaction between distributed program modules) is a sufficient basis for organising the provision of arbitrary function. Therefore, the SE-ODP is an *"enabler"* for distributed processing in general, and need itself have no other function.

The role of an SE-ODP is thus different from that of a conventional operating system, which manages resources and offers rich functionality. Under an operating system, the application interface is that between an application and the operating system. For an SE-ODP, the application interfaces of concern are those *between program modules*.

The current version of Deltase is a *prototype SE-ODP*, providing Deltase with a firm conceptual basis. The long-term objective is the convergence of Deltase with the relevant SE-ODP standards. Conformance to ODP standards will provide an assurance to potential users that they will not become locked into a proprietary system architecture.

7.1.2. Viewpoints

The concept of viewpoints is one that has already emerged from the ISO RM-ODP work. Viewpoints provide a framework for the discussion of distributed processing systems, based on the recognition that a given distributed system can be considered from a number of different points of view. Each viewpoint corresponds approximately to the view that some group of people within an enterprise (or user organization) will have of the distributed processing system:

- The **enterprise viewpoint** is essentially that taken by those responsible for the enterprise (or user organization) as a whole, and is therefore concerned with models of the overall objectives of the enterprise, and the part played by the distributed processing system in achieving those objectives. This viewpoint is concerned with what the system does for the organization, and how the system interacts with the rest of the organization.
- The **information viewpoint** is typically the view of a system designer or systems analyst, and is concerned with models of information structures and information flow. From the information viewpoint, the system specification is of a functional, rather than algorithmic, nature.
- The **computational viewpoint** is that of the application programmer, and models the application software as a set of objects that interact with one another in a defined manner. These objects would typically represent language-level program modules. From the computational viewpoint, these program modules are (by default) independent of the particular computing environment on which they will be executed.
- The **engineering viewpoint** is typically that of a system programmer, and is concerned with how to *engineer* support for the distributed application described by the computational model. The engineering viewpoint is concerned with achieving the required quality attributes for the distributed processing system; in particular, dependability (in its various forms), performance, and real-time responses. The engineering viewpoint is also concerned with the provision of abstract mechanisms to enable the application program modules to be independent of the computing environment.
- The **technology viewpoint** is that of the system builder, and models the computing environment in terms of its basic commodities (computing hardware, operating system software, communication systems, etc.).

Each of these viewpoints gives a different view of a given distributed processing system; these views are not layers, but are views that focus on different concerns.

Deltase, and SE-ODPs generally, are primarily concerned with the *computational and engineering viewpoints*. These are described in the next two sections.

7.2. Computational Model

This section concerns the computational viewpoint, and describes the computational model supported by Deltase. This model defines the rules for the behaviour of computational objects that can be supported by Deltase. Only those computational objects (and by extension, distributed applications) that conform to this model can achieve the full benefits offered by Deltase: transparent distribution, transparent fault-tolerance, portability, independence from LEX, etc.

The model is based on the support of *language-level computational objects*. Since Deltase is a prototype SE-ODP, the computational model primarily concerns the *interaction* between computational objects.

In the *black-box* view, a distributed application (figure 1) is viewed from the outside; the computational objects are seen as *black boxes*, which interact via defined interfaces. This is the view of, say, an application designer.

The *white-box* view is of *one* computational object, from inside that object; other parts of the system are seen only by the *interfaces* used for interaction. This is the programmer's view, and concerns the programming of a computational object in such a way as to conform to the Deltase computational model.

Many of the concepts on which this computational model is based are inherited from ODP and the *Advanced Network System Architecture* (ANSA, see [ANSA 1987, ANSA 1989]). However, there are a number of areas where the Delta-4 project has contributed, particularly in recognising the importance of language mappings.

7.2.1. Distribution Transparencies

Deltase offers a virtual environment that conceals, from the computational objects, the *consequences* of distribution and of the support of distributed fault-tolerance. Those consequences of distribution that are hidden from the application program are referred to as the *distribution transparencies*, as follows:

- **Access transparency** allows interactions between computational objects to be programmed independently of the actual low-level mechanisms used to implement the interaction. This means that the programmer is not concerned with the characteristics of the LEX.
- **Location transparency** ensures that bindings between objects are independent of the communication route that connects them. This means that the programming of interactions between objects is independent of the relative locations of the interacting objects.
- **Migration transparency** allows the re-location of an object in a distributed system without making that change of location (migration) apparent to the interacting objects.
- **Replication transparency** allows replicas of an object to be used, without the existence of the replicas being apparent to either the replicated object or to any other object that interacts with the replicated object. Within the Delta-4 system architecture,

replication is essential for the provision of fault-tolerance. Replication may also be used in Delta-4 and other architectures to increase performance.

- **Fault transparency** allows inter-object interactions to be programmed without regard to the possible existence of certain classes of fault within the distributed system.
- **Concurrency transparency** allows the concurrent use of a resource, without permitting inconsistent use of that resource.
- **Language transparency** allows the programming of inter-object interactions to be independent of the language system used, such that the programming of one object is independent of the language system used for the generation of the other object.
- **Machine transparency** allows the programming of inter-object interactions such that the programming of an object is independent of the particular type of host machine on which any other object may be executed.

These distribution transparencies hide differences that might arise from differences in processor hardware, in operating system software, in programming languages or in software development systems used for constructing or executing the application objects. In short, the distribution transparencies hide the consequences of *distribution* from the objects.

From the computational viewpoint, the distributed application is independent of the mapping of the computational objects onto the underlying computing environment; similarly, the replication of objects is transparent, so that this description makes no further reference to replicated objects.

7.2.2. Computational Objects

From the computational viewpoint, a *distributed application* consists of a number of (language-level) program modules, which can interact with one another. This is illustrated in figure 1.

The important feature of this computational model is that the program modules are *computational objects*; each computational object encapsulates some *state* (data) and the *operations* on that state.

From the computational viewpoint, these operations provide the only means whereby other computational objects may read, or cause changes to, the state encapsulated by a computational object. Direct manipulation, by one computational object, of the state of another computational object is not permitted, and, in the general case where two interacting computational objects are executing on distinct host machines, will not be possible.

The following text, adapted from [Golberg and Robson 1981], provides a clear description of the principles of object interactions:

> "The set of *operations* to which an object can respond is called its interface with the rest of the system. The only way to interact with an object is through its interface. A crucial property of an object is that its private memory can be manipulated only by that object's own activity. A crucial property of *operations* is that they are the only way to invoke an object's activity. These properties ensure that the implementation of one object cannot depend on the internal details of other objects, only on the *operations* to which they respond."

(This text has been adapted slightly to bring its terminology into line with that used in this book. In particular, the word "message" in the original has been changed to *operation*; from the computational viewpoint, interactions take place by the invocation of operations and the return of results. From the engineering viewpoint, such operations will be seen as the passing and return of messages.)

The implementation of the object's encapsulated state, and of the operations supported, is not visible to other computational objects.

7.2.3. Service Interfaces

From the computational viewpoint, object interactions are based on the offering and invocation of services, via clearly-defined interfaces.

The set of operations supported by a computational object (and made available for invocation by other computational objects) constitutes the interface to that object. In many cases, these operations fall into a number of distinct sets, and it is convenient to treat such a set of operations as a distinct interface, referred to as a *service interface*.

The distinct service interfaces supported by a computational object give different "views" onto that object. For example, in the case of an object offering a file service, there might be a general interface providing the common file-handling operations, and a specialized interface for management of the file service (available to only a limited group of users). Different service interfaces may be made available to different groups of users.

A *service interface specification* comprises the set of specifications of the operations within that service interface; each operation is defined in terms of:

- what parameters (including the type of each parameter) must be supplied when invoking the operation;
- what results will be returned (again, including the type of each parameter in the result);
- what *visible* effect this operation will have on the rest of the system;
- what effect (if any) this operation will have on the results of subsequent operations on that object;
- any constraints on the ordering of operations (e.g., a file cannot be accessed until it has been opened, etc.).

The specification of a service interface should exist independently of any service user(s) and service provider(s), since there should be many possible implementations of a computational object to support that interface. Within a system each service interface specification is assigned a *name* (the service interface type name) that is unique within the chosen naming context.

A *service provider* is a computational object that *provides* the services defined by a particular service interface, as defined by the service interface definition.

A *service user* is an object that *uses* (some or all of) the services defined by a particular service interface.

A service interface is *asymmetric*; the service user and service provider have complementary relationships to the service interface; the *service user* invokes operations on the *service provider*.

Figure 2 shows the elements involved in the interactions between two computational objects *A* and *B*. Object *A* is the *service user*; object *B* is the *service provider*. This interaction is constrained by a particular service interface; the only interactions supported are the operations and responses defined by that interface. A *D-association* is the logical path via which the computational objects interact; (see §7.2.5).

A given computational object may, of course, be both a user of some services and a provider of other services; the terms "service user" and "service provider" refer only to their relationship to the particular service interface, and not to the computational object as a whole.

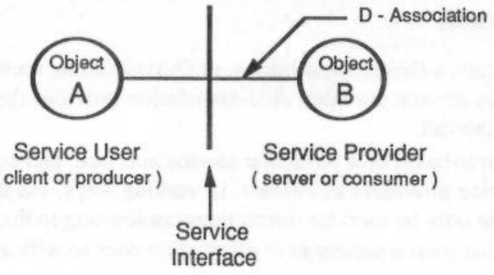


Fig. 2 - Simple Object Interaction

7.2.4. Remote Procedure Call (RPC)

The invocation of an operation in another computational object, with the passing of parameters and the return of results, is a direct extension, to distributed systems, of the widely-used procedure call mechanism provided by languages such as "C", Ada, etc.

The *Remote Procedure Call* (RPC) mechanism provides a means of programming the interactions between objects using existing language constructs, with each operation in a service interface treated as a separate procedure; the operation is invoked by a call on that (remote) procedure. This concept may be applied to both the service user (which invokes the operation by means of the procedure-call mechanism) and the service provider (in which the operation is written as a procedure).

The remote procedure call approach means that the same language construct, the procedure call, can be used both for the invocation of a service that is provided internally by a local procedure, and for the invocation of a service provided externally by another object.

At the programming-language level, the invocation of a remote operation is expressed by means of a procedure call with the parameters as language-level typed entities. The important point to note is that inter-object interactions can be programmed directly in the high-level language, with all the language-based type-checking mechanisms at both the calling and the called objects. The use of RPC provides an assurance that both the service user and the service provider conform to the same interface (the service interface specification); this is an important feature for distributed applications, and one that is not provided where the objects generate their own messages. Remote procedure calls provide a way of abstracting away from messages, and using language-level facilities for programming the interactions between computational objects.

By definition, a *local* procedure call (when the call is part of the same program as the procedure called) causes immediate entry to the called procedure. In order to keep the semantics of a *remote* procedure call as close as possible to the semantics of a local procedure call, a remote procedure call (where the called procedure is not part of the same program as the procedure call) also causes the remote procedure to be executed (and any results returned), before execution of the calling program is permitted to proceed beyond that remote procedure call.

Additional information on remote procedure calls can be found in [Birrell and Nelson 1984] (an important early paper on the implementation of RPCs), and the ECMA RPC standard [ECMA 127].

7.2.5. D-Associations

For two objects to interact, a *Deltase association*, or *D-association*, must be established between the service user and the service provider. A D-association provides the logical path via which computational objects interact.

Each D-association is based on a particular service interface, and enables a logical group of service users and service providers to interact, in various ways, via that service interface. A given D-association can only be used for interactions conforming to that service interface. Each computational object that joins a particular D-association does so with a defined role, either as a service user or as a service provider.

There may be two or more D-associations based on a given service interface type, each of these D-associations having its own set of service user and service providers.

Two computational objects can interact (via a particular service interface) only when both are members of the same D-association and have complementary roles (one as service user, the other as service provider). Finding the required D-association is achieved by the "trading" of service interfaces (see §7.3.3).

One computational object may be a member of several D-associations, with roles according to whether the computational object is a user of those services or a provider of those services. The white-box view of a D-association is a channel through which service requests are sent (in the case of a service user) or received (in the case of a service provider).

In order to support a variety of interaction styles, and, in particular, producer-consumer interactions (see §7.2.6), a D-association may include *multiple service providers* as well as multiple service users. In the *black-box* view, a D-association is modelled as a *multi-point virtual circuit*. This is illustrated in figure 3.

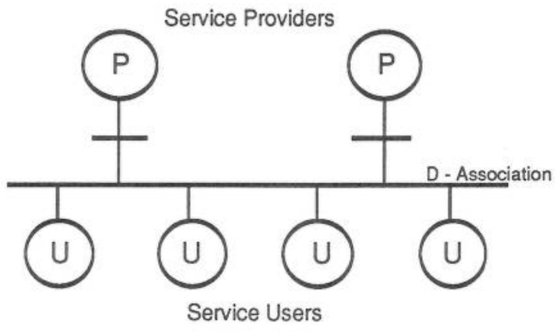


Fig. 3 - D-Associations

For a replicated object, all replicas will be members of the same set of D-associations, although this replication is not visible from the computational viewpoint.

A distinction is made between *creating* a D-association and *providing a service* via that D-association. A D-association is normally created by a service provider, but may, instead, be created by an *agent* (which then makes no further use of that D-association). Once a D-association is established, users and providers of that service may join it.

When a service user issues a service request, that request is sent to all service providers on the selected D-association. Multiple service providers are permitted only on those D-associations where none of the operations results in a reply to the service user, since a service user cannot be expected to handle multiple replies. This type of D-association is intended for the handling of producer-consumer operations, allowing a single service request to be multicast to a number of consumers.

Interface trading is the process of finding a suitable D-association, based on a particular service interface.

For a service user, interface trading is concerned with finding a D-association based on the required service interface, and providing access to a suitable service provider. This is done by searching a catalogue of D-associations (which is effectively a catalogue of services offered); the D-associations are identified by the service interface type name.

This approach to the establishment of D-associations is suitable for use in open systems, and does not rely on prior knowledge of the identity of either the D-association or the service provider.

From the computational viewpoint, interface trading may be invoked explicitly by the application programmer to create, join or leave particular D-associations. By default, interface trading will be carried out transparently. The *trading system* is discussed in section §7.3.3 as part of the engineering model.

7.2.6. Interaction Styles

There are two distinct styles of interaction (illustrated in figure 4), as follows:

1) **Client-server** interaction operates on the principle of a *handshake*; the service user invokes an operation on the service provider; on completion of the specified operation the result is returned to the service user. This is the interaction style of the remote procedure call, as described above. A "null" reply may be used to indicate completion of the invoked operation without passing any results.
2) **Producer-consumer** interaction is a *fire-and-forget* interaction; the producer (service user) invokes an operation on the consumer (service provider), but there is *no feedback* from the consumer to the producer. This style of interaction allows the call from the producer to be multicast to a number of consumers.

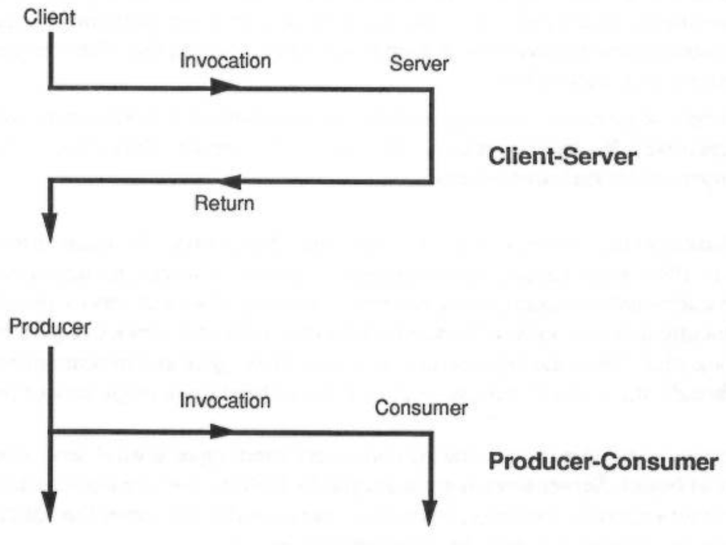


Fig. 4 - Client-Server and Producer-Consumer Interactions

The client-server operation is typically associated with a one-to-one interaction, as typified by a remote procedure call. A single D-association may be used independently by a number of service users (that are members of that D-association) to invoke operations on a service provider; this is the typical *server* situation, as in the case of a file-server.

By exploiting the "null" reply option, client-server operation may be extended to a single invocation of a number of servers; the "reply" condition would normally be that a null reply be received from all servers.

The producer-consumer style allows a number of different modes of operation. In particular, when used over a D-association that permits a number of consumers (service providers), an operation may be multicast to a number of visibly-distinct destinations; that is, the destinations are distinct from the computational viewpoint (and not just a single set of replicas).

One example of the use of the producer-consumer style is the dissemination of "alarm" messages, in an industrial plant, to a number of distinct alarm processing components. Another example is the "commit" operation of a distributed transaction.

These interaction styles are properties of the *operations* concerned. A given service interface may include both client-server operations and producer-consumer operations. Only when the service interface fulfils certain conditions may the D-association support more than one service provider. These conditions are now described.

7.2.7. Concurrency, Threads and Parallelism

A service provider will typically handle service requests from a number of service users, and may thus be faced with the requirement to handle a queue of service requests.

The simple solution is to handle one service request at a time, and for some service providers this may well be the best solution. However, for systems where overall performance is important, such a solution ignores the following:

1) For many service providers, the servicing of a request is not simply a matter of computation, but involves either invocation of other services, or access to a local device — such as a disc. In these cases there will be time during which the service provider is suspended; such time could be utilized in servicing other requests. This is particularly important in a distributed system, where the other services may be executed by another host.
2) Simple sequential processing provides no opportunities for one service request to be preempted by another request of higher precedence. This case is particularly important for real-time systems.

7.2.7.1. Concurrent Processing of Service Requests. Deltase provides the mechanisms to allow a service provider to process a number of service requests *concurrently*. From the computational viewpoint, the concurrent processing of several service requests is seen as several concurrent invocations of the service provider, with each service request handled by a separate invocation. From the engineering viewpoint, these separate invocations are handled by separate threads, the "server" threads, with each thread handling a single service request (see §7.3.5).

Server (or *spawned*) threads provide the concurrent handling of several service requests by a computational object. Server threads are managed by Deltase, and are not directly visible to the application programmer. However, interference between distinct server threads could occur through the use of shared data within the computational object.

The application programmer must be aware of the possibility of multiple concurrent invocations when programming a computational object. Persistent shared data may be accessed on behalf of any service request, and may, therefore, be subject to conflicting requests from concurrent invocations of the computational object. It is essential that such shared data be protected from the inconsistencies that could arise. However, this protection is the responsibility of the application programmer, using existing techniques (semaphores, monitors, etc.).

7.2.7.2. Concurrent Processing within a Service Request. *Forked* threads provide a facility for distinct threads within a computational object. These threads are directly visible to the application programmer, since they are explicitly created and deleted by program commands.

A remote procedure call causes the calling thread to be blocked; other forked threads within the same computational object may execute if there is processing to be done.

Each forked thread is associated with a particular server thread, and thus with a particular invocation of the computational object. At the start of the execution of a service request, there are no forked threads; any that are required must be created. A forked thread may be deleted when its activity is complete. When execution of that service request is complete (that is, the server thread becomes free), all remaining forked threads (associated with that server thread) are destroyed.

The commands to control forked threads are defined by a service interface, and invoked by procedure calls; to the computational object, the handling of forked threads appears the same as other services. The commands provided include those for creating, deleting and activating forked threads within that server thread. The scheduling of threads within a computational object is performed by Deltase.

As described above, the control of access to shared data within the computational object is the responsibility of the application programmer.

7.2.7.3. Parallelism. Within a distributed system, true parallelism may occur when services requested by one computational object are executed on different hosts. The achievement of this is outside the control of the application programmer, and will depend on how the system is configured; the location of the service provider is deliberately hidden through location transparency. However, the application programmer is able to create opportunities for parallelism; if service requests are invoked sequentially (with one completing before the next one is invoked), then there are no opportunities for parallelism.

The programmer has two alternative strategies that may be used to offer "true" parallelism, where services invoked are executed on other host machines, so that such services may be executed in parallel with each other and with the capsule invoking these requests.

Forked threads may be used in association with remote procedure calls, to achieve several concurrent service requests from a given service user. When an RPC is used within one thread, that thread will be blocked (suspended) until the result is returned, but it is only that thread that is blocked. Other threads that are not blocked may execute, and, in the course of their execution, make an RPC, causing that thread to be blocked. The result is that, at a given time, there may be a number of outstanding service requests originating from a single server thread.

Delta-4 has explored an alternative mechanism for use with languages that do not support multiple threads; this is the *Remote Service Request* (RSR) mechanism. RSR is a procedure-call-like method of invoking a remote service, without blocking the calling thread; this allows execution of the calling thread to continue whilst the remote service is fulfilled. A remote service request carries an implicit pledge to synchronize with the service completion and accept any results, by means of a *Remote Service Wait* (RSW) call. The means of guaranteeing that this pledge is kept, and that its context is appropriate, are language-dependent issues. RSR does not require changes to the language compiler, and carries with it the benefits of expressing the parameters for the remote service as language-level entities, in the same way as for RPC.

The RSR construct provides a mechanism for the parallel invocation of a number of remote procedure calls, without the need for explicit creation of separate (forked) threads and without the need for language extensions. The construct is not elegant to use, relying on the application programmer to invoke RSW within a valid context; as defined, the RSR mechanism is unstructured.

To provide the facility for multiple remote procedure calls in an elegant manner (and therefore less prone to error), a "COBEGIN" construct has been defined as an extension to "C", to provide a structure for programming multiple remote procedure calls without the need for creation and deletion of forked threads.

The COBEGIN construct allows a set of remote procedures calls to be specified; these calls are to be invoked without waiting for the replies, so that the calls may be executed in parallel. The calling thread continues until a specified point is reached, and then waits until the results have been returned from all the set of remote procedure calls.

A pre-processor is used to convert each COBEGIN construct into a number of as RSR and RSW operations, so that no changes to the "C" compiler are required.

RPC, RSR/RSW and COBEGIN are alternative methods of programming the invocation of interactions at the language level within a service user; there is no means whereby a service provider can distinguish between a service request invoked via RPC and one invoked via RSR or COBEGIN.

7.2.7.4. Use of D-associations. For a given service interface, a single D-association is sufficient to meet the requirements of a multi-threaded service user; a D-association can support several concurrent service requests, serviced by concurrent operations in the service provider. These service requests arise not only from concurrent operations in one service user, but also from several different service users.

7.2.8. Interface Definition Language

Where the service user and service provider both use the same programming language, the service interface may conveniently be defined as a set of procedure definitions in that language.

Deltase currently uses a *"C" Interface Definition Language* (CIDL) [Delta-4 1990] for defining the interfaces between objects written in "C". Similarly, Ada package definitions provide an interface definition for objects written as Ada packages. However, to support interactions between an object written in "C" and one written in Ada, it is necessary to produce two equivalent interface definitions, one in "C" and one in Ada.

To support interactions between objects written in different languages, a service interface specification should be in a form that can be readily converted into equivalent language-dependent forms, so that the correct use of that interface can be assured for the different languages concerned. The aim is that, eventually, the specifications of service interfaces should be written in a standardized *Interface Definition Language* (IDL). An IDL is a language used for defining service interfaces in such a way that the definitions are not dependent on any particular programming language.

Software tools will be required to mechanize the transformation of a service interface specification (in IDL) into its *representation* in the actual programming language of implementation. This representation is a description of a set of operations, together with their parameters, expressed as the appropriate construct in each individual programming language.

Only by such software tools can the necessary levels of assurance be given that the interface definitions in different languages are equivalent.

Once such tools are available, the benefits of any interface checking (such as type-checking) carried out by the relevant compilers will be inherited into the distributed environment, even where service user and service provider are implemented in different programming languages. At present, this highly desirable check on the correctness of distributed interactions can be applied in full only where service user and service provider are written in the same language; this is a considerable advance on traditional (message-based) methods.

Configuration control also becomes necessary where the use of service interface definitions becomes widespread, with the aim of ensuring that both the service user and the service provider conform to the same *version* of the interface definition.

7.2.9. Remote Procedure Specifications

Because these interfaces are for use in distributed systems, there are certain restrictions on these interface definitions resulting from the separation of the interacting objects. In particular:

1) The interface must be defined in such a way that all the information required by the service provider is included in the parameters; with a remote procedure call, the called procedure (in the service provider) cannot access the context of the calling procedure (in the service user).

2) The information passed as parameters must be meaningful in the context of the called procedure; pointers, and other local information may not be meaningful in another context. This applies equally to parameters returned as results.

3) The specification of each operation must be compatible with the syntax of procedure calls in a wide range of programming languages. Thus, few languages permit variable numbers of parameters, or different types for a given parameter. The full benefit of the remote procedure call paradigm can only be derived where the procedure can be invoked within the scope of the available language facilities.

This model assumes that the response from the remote procedure will always reach the computational object from which the remote procedure was invoked; that is, there is an assumption that, in a Delta-4 system, a remote procedure call can be treated (by the applications programmer) exactly like a local (i.e., internal) procedure call. This simple model makes no provision for handling the situation where the return from the remote procedure is lost, due to failure of the machine on which the remote procedure is executed, or for any other reason. In the event of loss of the response, the thread from which the remote procedure was invoked would be suspended indefinitely; this would have a knock-on effect, where the thread is providing a service for another object.

With a local procedure call, both the calling program and the called procedure are executed by the same machine, without requiring the use of a communication system. If the called procedure is lost through machine failure, then so is the calling program. Thus, for a local procedure call, there is no point in making provision for loss of the return from a called procedure.

A non-fault-tolerant distributed system represents the other end of the spectrum of possible ways of handling loss of response from a (remote) procedure call. To prevent the system from locking up, on failure of a machine or partitioning of the network, the applications must be able to recover from the failure of a remote procedure call.

For applications to be used on Delta-4-based systems, there is a range of possibilites between these two extremes; for instance, if the remote service is replicated, the designer may assume that it will exhibit sufficient availability without the need for special precautions. There is a compromise here between providing transparency to distribution, and yet being able to recover from failures that arise because of distribution.

Where the degree of transparent availability achieved through Delta-4 fault-tolerance mechanisms is inadequate, then the recovery mechanisms provided by Deltase may be used to handle the consequences in an application-specific manner; fault-tolerance is no longer fully transparent for applications that do so.

The recovery mechanisms are based on the use of *event handlers*, as described in §9.7.4. Thus an event is raised if no response from the remote procedure has arrived at the calling program within a specified time.

Event handlers provide a powerful mechanism for handling exceptional conditions; in the Delta-4 Extra Performance Architecture, where timeliness is of concern, the event handling mechanism is used to invoke any necessary recovery action when the required time constraints are not met.

The event handling mechanism provides the following options which are somewhat analogous to the facilities provided by the Ada programming language:

1) application-specific event handlers (visible to the applications programmer) may be provided (as part of the application) for handling specific events;
2) default event handlers may be included as part an application for handling specific events;
3) if there is no event handler for a specific event, then the event is passed to the client invoking the service on which the event was raised.

For further information on event handlers, the reader is referred to §9.7.4.

7.2.10. Application Programmer's View

The computational viewpoint is the application programmer's view; this is the view of the system as it appears from the source code of a program module written in a high-level language.

In order to achieve the properties offered to computational objects, *all* external interactions must follow the rules laid down for interactions between computational objects. For a given computational object, all the services used must be accessed via service interfaces, as described. This precludes the use of "standard" libraries that make direct calls on the local operating system.

The requirements for portability of a computational object between implementations of Deltase for different environments are:

1) the use of a standard programming language, and the program development systems to support such a language;
2) availability of the services used by that object in any system to which the object is ported;
3) all external interactions to take place via service interfaces, as described above; such interactions may be programmed using only language-level entities with RPC or RSR.

Departure from these rules (for example, for performance reasons) may reduce the portability of that computational object and its transparency to distribution.

7.2.11. Interactions with the non-Deltase World

To provide commercially-viable solutions for distributed computing systems in the real world, then applications based on Deltase have to be able to interact with existing software and with input-output device drivers. While the details are the concern of the engineering viewpoint, some of these issues will be directly visible to the application programmer from the computational viewpoint.

7.2.11.1. Transformers. The term *transformer* is used to refer to an entity that interacts both with the Deltase world of computational objects and explicitly with some part of the non-Deltase world. In particular, a *transformer object* is a *"half-object"*, that has the properties of a computational object from one view (and can therefore be modelled as such in the computational viewpoint), but also interacts with the non-Deltase world. To the Deltase world a transformer object appears as a computational object, and interacts with other computational objects via the Deltase mechanisms. From the other side, the transformer interacts explicitly with some other computational world.

An important application of transformer objects is to provide an interface from the Deltase world to an existing (non-Deltase) software package. To make services provided by a non-Deltase application (such as a database, see §7.6) available in the Deltase world, a transformer object appears to the Deltase world as a *service provider*, and to the non-Deltase world as a *service user*. This provides a method by which computational objects can gain access to the services of existing software packages. Typically, the software accessed in this way will be proprietary and commercially important; two examples are:

- software offering standard OSI services such as FTAM;
- database software supporting standard interfacing methodologies such as SQL.

Equally, a transformer object may be used to make the services provided in the Deltase world available to the non-Deltase world. In this case, the transformer appears to the Deltase world as a *service user*, and to the non-Deltase world as a *service provider*. There is the possibility of providing a service to an existing service specification; the server would have the benefits of Deltase expressive power and representational transparency, together with Delta-4 dependability.

Another example of the use of transformer objects is in the interfacing to input-output devices or device driver software; that is, in interfacing to the real world. This is discussed in chapter 12.

The transformer concept also provides a method of inter-working between the Deltase world of computational objects and "other" computational worlds, such as those of X-windows or MMS (Manufacturing Messaging Service).

An application programmer developing a transformer object will necessarily see interactions both with the Deltase world (via service interfaces), and with the non-Deltase world. However, to emphasise the point made earlier, Deltase can guarantee the properties and transparencies, described for the computational model, only where the computational objects are "pure", i.e., *all* their external interactions conform to Deltase. The implications of transformer objects on fault-tolerance are discussed in chapter 7.

7.2.11.2. Interactions with the Local Environment. The term "local environment" here refers to the local operating system.

To give objects a consistent view of externally-provided services, those services provided by the local environment (and invoked explicitly by the object) should be invoked in the same way as services provided by other objects; that is, the interactions between an object and the local environment should be modelled as interactions between objects, via a defined service interface. An object should not be able to distinguish between a service provided by another object, and a service provided by the local environment.

Thus, the (visible) parts of the local environment are accessed via defined service interfaces, through which the computational object accesses the services provided locally.

Wherever possible, these interfaces should be generic, i.e., not dependent on any particular local environment. One of the aims of Deltase is to make possible the creation of computational objects that are independent of local environment. An important contribution to this objective is by ensuring that an object's explicit interactions with its local environment may be programmed in the same way as interactions with other objects.

7.3. Engineering Model

7.3.1. Introduction to the Engineering Model

The engineering viewpoint is the *system programmer's view* of the distributed system, and is concerned with the *engineering* necessary to support the *virtual machine* (computational viewpoint) on the available computing environment (technology viewpoint). This support must be provided in such a way that the required *distribution transparencies* and *quality attributes* (performance, dependability, etc.) are achieved.

While many of the concepts described in this section are inherited from ODP and ANSA, Delta-4 has a distinctive contribution to make in the application of these concepts for the support of dependability and real-time in distributed systems.

From the engineering viewpoint, Deltase is seen to support the computational model by a combination of:

1) **generation support**: in the transformation of the language-level computational objects into *capsules*; capsules are executable Deltase-based software components — the capsule generation software is referred to collectively as *Deltase/GEN*;

2) **run-time support**: the Deltase run-time support subsystem, which includes additional capsules that are not visible to the application programmer — the Deltase run-time support software is referred to collectively as *Deltase/XEQ*.

Within Delta-4, the term software component is used generally to refer to the executable units, or modules, of a distributed application. The term *capsule* is used to refer to a software component that is the representation, in the engineering viewpoint, of one (or more) computational object(s), all of which interact *only* via Deltase. The term "capsule" is taken from the ECMA ODP work [ECMA TR/49]. The term *transformer capsule* is used for a capsule that contains one (or more) transformer objects, so that there are some potential interactions that are not under the control of Deltase.

A capsule is seen as an engineering object. In Delta-4, the ODP concept (representing an *address space*) is mapped onto a software component. From the engineering viewpoint, a system is modelled as a number of interacting capsules.

In the technology viewpoint, a capsule is mapped onto an executable unit supported by the local operating system, such as a UNIX process.

From the engineering viewpoint, a capsule is seen to contain one or more computational objects that are surrounded by additional software to perform the transformations and mappings that take place at run-time. This additional software is referred to as the *envelope*. The process of generating the envelope can be automated by suitable software tools, thus preserving the view that Deltase supports language-level computational objects. Each capsule contains one envelope, which represents all code added to the computational objects to produce the capsule; such code includes libraries and automatically-generated code, for all computational objects (and transformer objects, if any), within that capsule.

There is an analogy between these mechanisms and the compile-time and run-time support subsystems of a high-level programming language. Taken together, these subsystems provide the programmer with the illusion of preparing code to run on a machine which "understands" the high-level language program, whereas much of the "understanding" is resolved prior to any execution being possible, during compilation to executable code.

Figure 5 shows two phases, or "Epochs", in the generation and installation of a capsule.

1) The **capsule generation phase**, in which computational objects are transformed into capsules by Deltase/GEN, and the envelope is generated from a combination of selected libraries and automatically-generated code. A capsule is generated for use in a particular environment.
2) The **capsule installation phase**, in which the capsule is loaded onto the host (or hosts) on which the capsule is to be executed. The initialization code is executed, during which time all the default trading takes place, to establish D-associations for services provided and to join D-associations for services used.

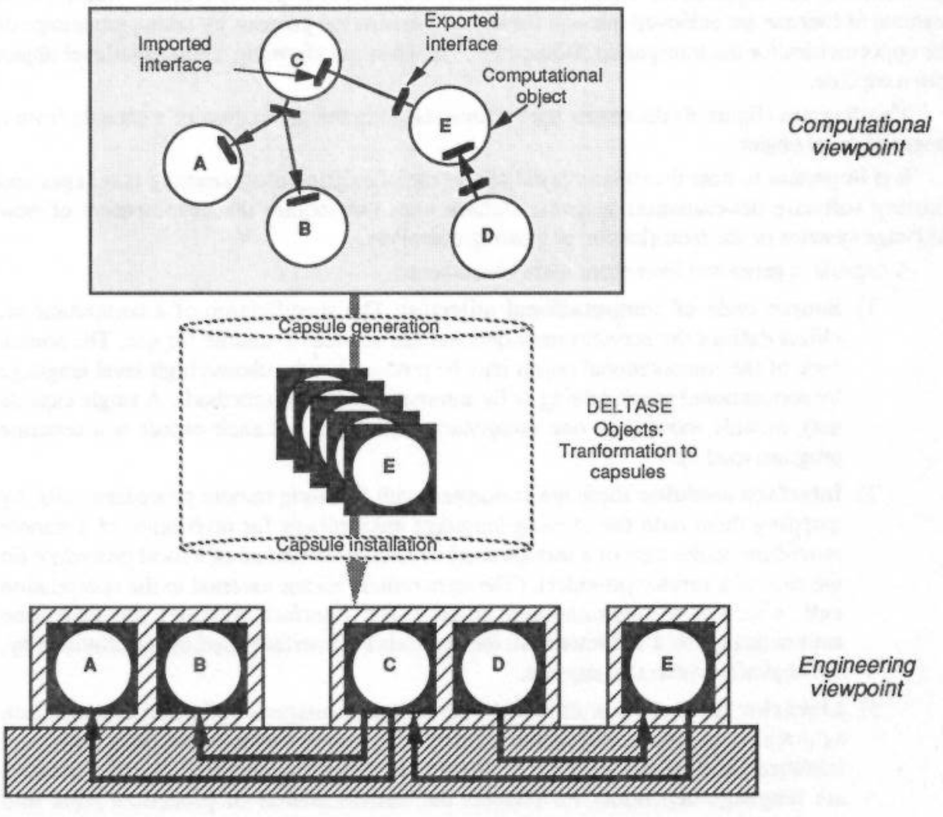


Fig. 5 - Relationship of Computational and Engineering Viewpoints

Once this installation phase is complete, the capsule is available for execution as part of the run-time system.

The *Deltase run-time support subsystem, Deltase/XEQ,* provides a number of services that are *always present*, and is itself a distributed application. In accordance with the best engineering practice, it is therefore based on defined Deltase principles, namely those supported by Deltase; Deltase makes recursive use of itself.

From the computational viewpoint, Deltase/XEQ consists of a number of computational objects, which may then be transformed into capsules, with all the benefits (distribution transparencies, portability, etc.) that Deltase seeks to confer on distributed applications.

Most of the capsules in Deltase/XEQ are not normally visible to the application programmer; in particular, the capsules that provide the trading activities. Interactions with Deltase/XEQ are visible in the engineering viewpoint, and are invoked from the code that constitutes the envelope. (From the computational viewpoint, these are two separate applications that co-exist but are not visible to one another.)

7.3.2. Generation of Capsules

7.3.2.1. Introduction. Deltase provides support for *language-level objects*, and the generation of the capsules for a particular system is an essential part of Deltase. Many of the features of Deltase are achieved through the capsule generation process, by taking advantage of the opportunities for the transparent addition of code when transforming a language-level object into a capsule.

The diagram (figure 6) illustrates the various stages in the generation of a capsule from a language-level object.

It is important to note that this is based on the use of existing programming languages and existing software development systems; Deltase does not require the development of new language systems or the modification of existing compilers.

A capsule is generated from three main constituents:

1) **Source code of computational object(s)**: The specification of a computational object defines the services provided and the services available for use. The source code of the computational object may be produced in the chosen high-level language by conventional programming or by automatic generation methods. A single capsule may include more than one computational object, but each object is a separate program module.
2) **Interface modules**: these are concerned with handling remote procedure calls, by mapping them onto the existing language mechanisms for invocation of a remote procedure (in the case of a service user) or remote invocation of a local procedure (in the case of a service provider). (The term remote means external to the compilation unit, which is a computational object.) The interface modules are generated automatically, by a software tool, for each service interface used by, or provided by, the object(s) within that capsule.
3) **Libraries**: these provide all non-unique support management for that capsule, such as thread scheduling, dispatching, interfacing to the communication system, and initialization code. Some of these libraries are dependent on the LEX; other libraries are language-dependent (to support the transformation of procedure calls into messages, and vice-versa). Selection of a capsule's fault-tolerance model will result in the appropriate library being used for those parts of the envelope that are dependent on the fault-tolerance model. These libraries are not visible to the

computational object itself, but form an essential part of the infrastructure to support the computational object(s) within a capsule.

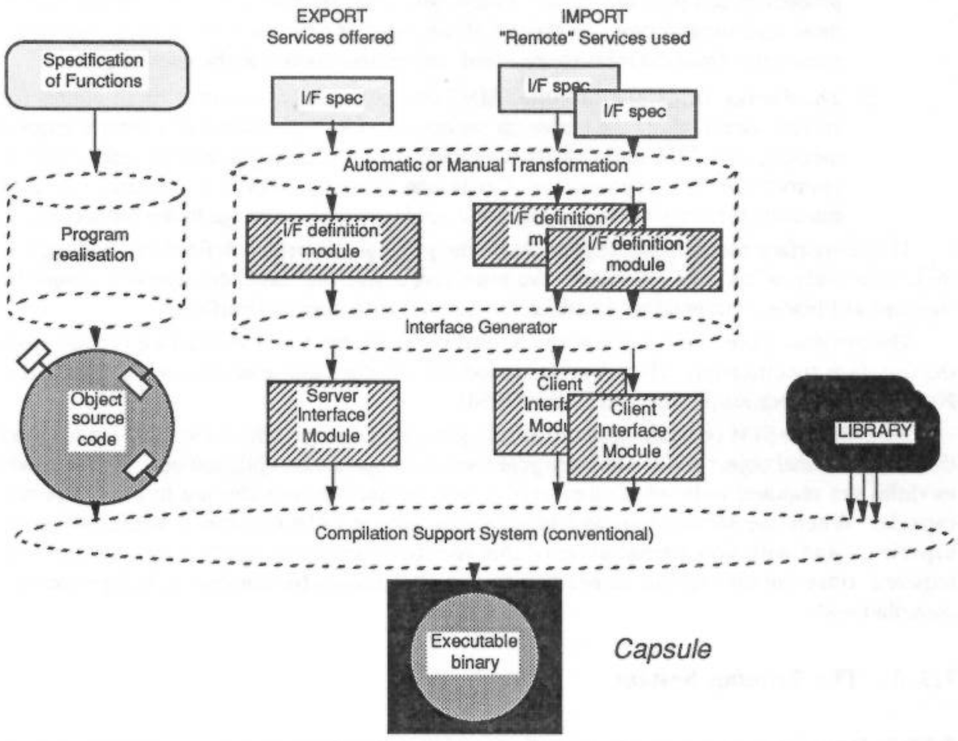


Fig. 6 - Capsule Generation by Deltase/GEN

The interface modules and the libraries, together, constitute the *envelope* and implement the Deltase abstractions that the programmer is encouraged to use.

The output produced by Deltase/GEN is a file containing a binary representation of the capsule for execution in a particular local execution environment. This file provides a template, that may be used to instantiate instances of that capsule wherever that run-time environment exists.

Because Deltase supports language-level objects, which provide opportunities for the addition of environment-dependent code during the generation process, portability of capsules (executable binary) is not a requirement for Deltase objects.

However portability could, in the future be based on an "architecture-neutral" intermediate code version of the object; this would allow the addition of the environment-dependent code (the envelope), while *hiding* the original source code and therefore protecting the intellectual property rights of the originator. This approach implies a subdivision of the generation phase into two (or more) separate phases.

7.3.2.2. Interface Modules. The language-specific definitions of the interfaces are used both by the programmer who is constructing the specified object functionality in the chosen language, and by the major Deltase transformation component, the *interface generator*.

There are two types of interface module:

1) The **Client Interface Module** (CIM) is the server's (or service provider's) *representative* within the client (or service user). The CIM marshals (assembles) the procedure call parameters into a message, sends the message to the server via the local operating system; on receipt of the reply, if any, the CIM unpacks the result parameters from the message received, and returns control to the client object.

2) The **Server Interface Module** (SIM) acts as the *representative* of all clients (or service users) within the server (or service provider). On receipt of a service request message, the SIM unmarshals (unpacks) the procedure parameters, and calls the appropriate server procedure. On return from the server procedure, the SIM marshals the reply into a message, and sends the reply message to the service user.

These interface modules are dependent on the particular interface definitions, and require their own body of code to implement the interface; hence the interface modules cannot be supplied as libraries, but must be generated for that particular service interface.

The interface generator is a software tool that generates the required interface module from the interface specification. These interface modules are the Deltase equivalent of Birrell and Nelson's *stub* modules [Birrell and Nelson 1984].

Each CIM or SIM consists of automatically generated source code in the same language as the computational object and is normally generated as a separate compilation unit. The interface modules are required only where the service user and service provider are to be in separate capsules. Where the service user and service provider are to be co-located within the same capsule — and with no external users of this service — then the interface modules are not required, since the linking can be handled by the mechanisms for combining independently-compiled units.

7.3.3. The Trading System

7.3.3.1. Introduction. As described earlier, a service user must establish a binding to a D-association before those services can be invoked. The purpose of the trading system is to enable such bindings to be established (and deleted) *dynamically*, based on matching the service interface type required with the service interface type of the available D-associations. The trading system enables a service user to establish a logical path to a service provider for the purpose of invoking services and receiving replies, without either the service user or the service provider having to know the location or the identity of each other.

The trading system is based on:

1) a *trading catalogue* which holds information on the available D-associations;

2) a number of *traders* that offer the trading service and are responsible for execution of the operations on the trading system.

Logically, the trading catalogue is a distributed catalogue, but it may be partitioned into a number of separate named catalogues, or *trading scopes*. There is a separate record in the trading catalogue for each available D-association; this record holds the name of the particular service interface type on which that D-association is based. When a new D-association is created, its record is created in the specified partition of trading catalogue. The use of distinct trading scopes allows the service interface type name to be context-relative. The trading catalogue is analogous to a "Yellow Pages" telephone directory, which gives a list of service *offers*.

The two principal operations on the trading system are the export and import operations as follows (figure 7):

1) An *export* operation is used by a *service provider* to create a D-association and to act as a provider of that service. The export operation specifies both the name of the service interface type (on which the D-association is to be based) and the name of the catalogue (or trading scope) where a reference to the D-association is to be lodged (for use by subsequent import commands).

2) An *import* operation is used by a *service user* to join a D-association for access to the services provided — as defined by the service interface type. An import operation causes the specified named catalogue(s) to be searched for a D-association supporting the specified service interface type. Having selected a particular D-association providing the required service, a *binding* is established between the service user and the selected D-association.

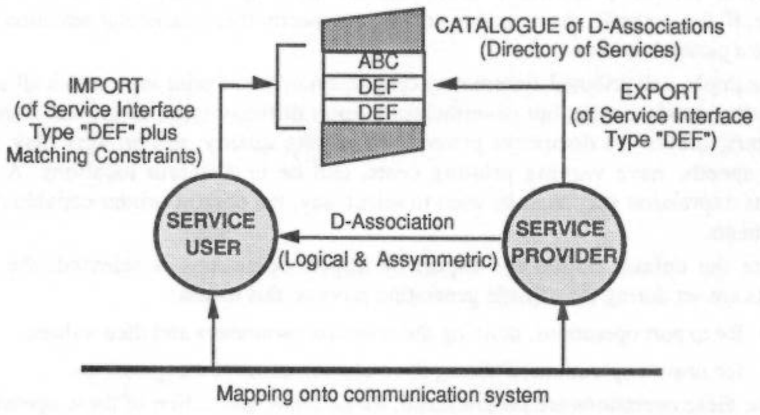


Fig. 7 - Principles of Trading

Most import and export operations are carried out, transparently to the programmer, as part of the capsule initialization code (which is created by Deltase/GEN and is executed as soon as an instance of a capsule becomes "live"). This initialization software creates (by means of an export operation) a D-association for each service *provided* by the capsule, and establishes a binding to (i.e., joins — by means of an import operation — a D-association for) each service *used* by the capsule. The capsule remains a member of all of these D-associations for the whole of its life. These default operations would not be visible to the application programmer.

Each capsule is given a preset binding to its local trader, so that a capsule can always perform export and import operations, as essential prerequisites to interactions with other capsules.

However, for certain specialized computational objects, it may be necessary for the joining and leaving of D-associations to be under direct program control, at run-time. In these cases, *import* and *withdraw import* operations will be written into the code of the computational object.

When a capsule joins a D-association, the communication paths (and associated resources) are set up using information from the catalogue; the D-association is mapped onto the communication system or the facilities of the local execution environment.

7.3.3.2. Constraints Matching. The aim of interface trading is to select a suitable D-association for a particular import request. Where there is (within the specified trading scope) more than one D-association matching, the selection can be refined by means of *matching constraints*. In the absence of any matching constraints, the first D-association supporting the specified service interface type will be selected. Matching constraints may also be used where specific constraints apply to any D-association selected, so that even if there is only one D-association matching the service interface type, it will not be selected unless the matching constraints also allow selection.

Associated with each service interface type are a number of selection parameters; these parameters are associated with the service interface, but do not form part of the service interface as visible to the application programmer. When a D-association is created, values are assigned to these selection parameters.

An import operation includes a *matching constraint expression*, expressing relationships between the selection parameters. These expressions are evaluated for each compatible D-association to find the "preferred" D-association. It may happen that *none* of the D-associations is suitable, if, for example, the matching constraints specify that a particular selection parameter must have a particular value.

For example, a distributed system may contain a number of print servers that all support the same service interface type, but nevertheless support different types of printer. Some may be line printers, others A4 document printers of varying quality; the printers may operate at different speeds, have varying printing costs, and be in different locations. A matching constraints expression may then be used to select, say, the nearest printer capable of printing A4 documents.

Where the default option for export or import operations is selected, the matching constraints are set during the capsule generation process; this means:

- for export operations, defining the selection parameters and their values;
- for import operations, defining the matching constraint expressions.

Where these operations are programmed, for dynamic invocation of these operations, then the corresponding information must be included in the computational object concerned.

7.3.3.3. Access Control. The trading system handles the creation and joining of all D-associations, and is thus able to control both the service offered and the access to those services. A particular service user can access a given service only if that service user is permitted to join the D-association through which the service is accessed. Similarly, a service provider can offer a service only if the export operation is accepted.

A trading system may control not only the creation of D-associations, but also which capsules may join which associations.

7.3.3.4. Negotiation of Transfer Syntax. Interactions between heterogeneous environments require the use of an architecture-neutral transfer syntax, such as ASN.1, for the encoding of the RPC parameters and results.

The code to perform the encoding and decoding can be generated automatically, as part of the envelope, and forms an essential part of the support for the transparency to heterogeneous machines and heterogeneous language systems. However, such encoding and decoding impose an overhead on interactions, and an objective is to avoid such conversions where the interacting capsules use an identical native transfer syntax. The native transfer syntax depends not only on the local execution environment, but also on the language systems, since different compilers may represent a given data-type in different ways even where the processors are identical.

The trading system has an opportunity to provide information for determining whether use of an intermediate transfer syntax is necessary or not. On an import operation, the service user could be informed whether the service provider uses an identical transfer syntax or not. In the latter case, conversion to an intermediate transfer syntax will be necessary, for both the invocation request and its reply. This approach would also require that the client and server interface modules (CIM and SIM) could handle both their native transfer syntax and the architecture-neutral transfer syntax.

7.3.3.5. Epochs of Trading. Trading can occur at several different times, or epochs in the life of a capsule. The only requirement is that the binding to the D-association be established before the first invocation of an operation of the interface. This can be:

1) during *construction* — when capsules are generated;
2) during *configuration* — when an instance of a capsule is installed on a host for execution;
3) during *initialization* — when capsules are first entered, D-associations are established by execution of the initialization code that was automatically generated;
4) during *execution* — when capsules are active; the program itself invokes operations on the trading system.

7.3.3.6. Joining a D-Association. There are a number of circumstances in which a capsule might require to join an existing D-association other than by an import operation:

1) where there are to be a number of distinct *service providers* on that D-association, in which each receiver is to act as a service provider; for example, in the handling of system alarms;
2) where a capsule has been created by cloning, and therefore needs to join all of the D-associations of which its other replicas are members;
3) where a capsule has been given a reference to a D-association, as a parameter, to allow that capsule to join the specified D-association for access to the service. (This is like passing a file reference, to enable a print server to print the file without the invoker having to pass the whole file.)

In each of these cases the particular D-association to be joined would be identified by reference, rather than found by the normal operation of the import command.

Additional operations on the trading system are required for the following:

1) to enable an agent to *create* a D-association (without joining it as either service user or service provider);
2) to enable a service provider to *join* an existing D-association (for use when there is more than one service provider on that D-association);
3) to enable a service user to *leave* a D-association;
4) to enable a service provider to *cancel* its offer of service, closing the D-association if there is no other service provider on that D-association.

7.3.3.7. Trading System as a Distributed Application. The Deltase trading system is a distributed application, and may be implemented as a Deltase-based application, which is not normally visible to the application programmer. Where the trading system is accessed directly from an applications program module, the operations on the trading system are invoked via service interfaces, in the same way as other interactions.

Each capsule must have a preset local binding to the D-association for operations on the trading system, so that a capsule can always interact with the trading system.

There are many possible implementations of the trading system to provide the required functionality. As a distributed application, a trading system can take advantage of the implementation choices offered by Deltase; such choices include the grouping of computational objects into capsules, the fault-tolerance model used by each capsule, and the distribution of the capsules among the available hosts. One constraint is that there is normally a component of the trading system on each host, both to provide a purely *local* trading service, and to provide the interface to the rest of the trading system — the *global* trading system.

From the computational viewpoint, access to a service is the same, whether the service be local or global; this amounts to a selection of the trading catalogue. However, the choice can affect both the *performance* (access to a globally-provided service will normally take longer than a locally-provided service), and *fault-tolerance*. The choice can be made by the system designer, and hidden from the application programmer.

7.3.3.8. Local and Global Trading. The term "local" here means "within the same host machine". When the local portion of the trading catalogue is selected, the service will be provided locally, *and will not be replicated.*

The diagram (figure 8) shows a service user (client) and service provider (server) performing *local* trading, by reference to the local portion of the trading catalogue. This is of course only possible when the two capsules reside on the same host. In this case, a D-association is mapped onto the Inter-Process Communication (IPC) facilities of the local operating system.

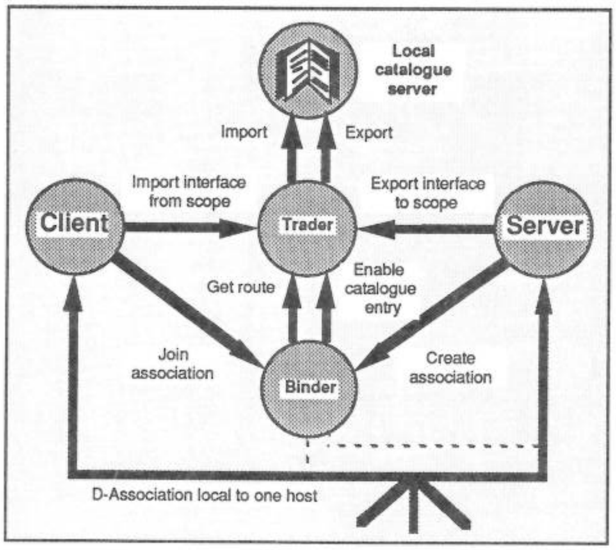


Fig. 8 - Local Trading

Figure 9 shows a service user and service provider performing *global* trading and binding, by reference to a global portion of the trading catalogue. The service user and service provider may be replicated, as required, and may reside on either the same host or on different hosts. Global trading is the normal case, and the D-association is mapped onto the underlying communication system, and inherits its properties.

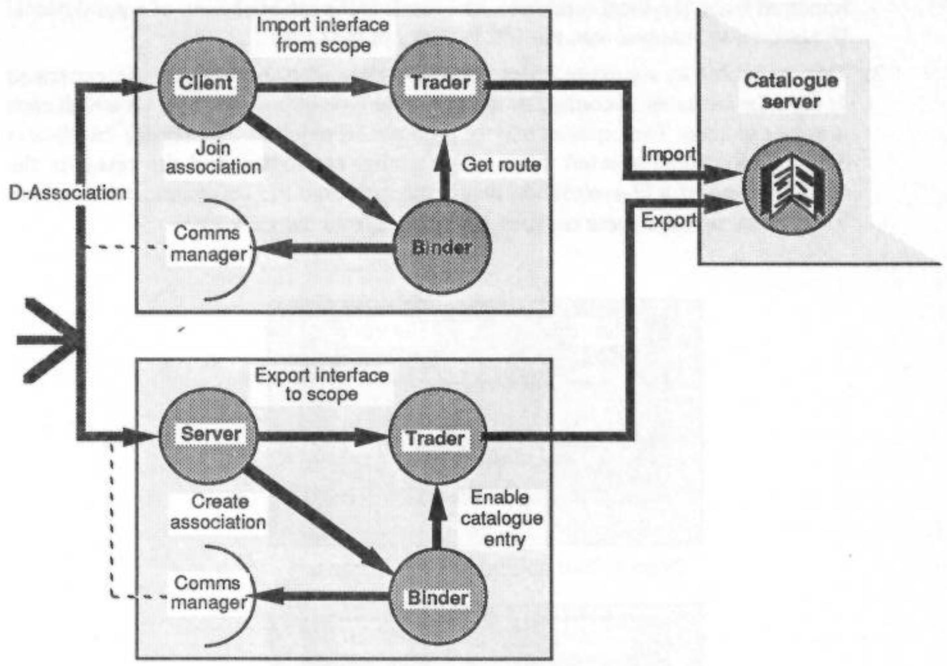


Fig. 9 - Global Trading

7.3.4. Implementation of Remote Procedure Calls

7.3.4.1. Configurations. This section follows the course of a remote procedure call, to show how all the various actions, which have been introduced separately, fit together. The word "remote" refers to a procedure outside the language-level computational object that made the procedure call.

Figure 10 shows the same two computational objects transformed into capsules in three different ways, with different implementations of the remote procedure call; from the computational viewpoint, these three cases are all indistinguishable.

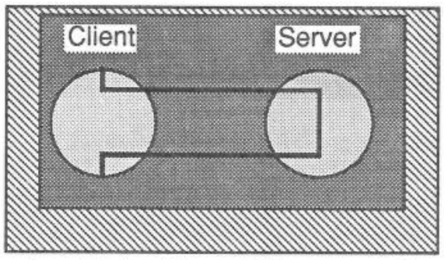
The three cases are as follows:

1) **Two objects in one capsule**: In this case, the service user and the service provider objects have been combined into a single capsule, and generated on the assumption that the use of this particular service will not be offered to other capsules. This case therefore simplifies to that of combining two source modules into a single compiled entity. The D-association is replaced by a conventional inter-module binding, established by the compiler and linker. This diagram illustrates why it is not always appropriate to map computational objects to capsules on a one-to-one basis.

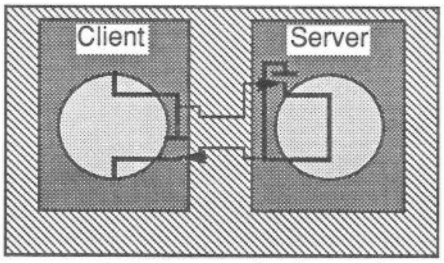
2) **Two capsules in the same local execution environment**: In this case, each computational object has been generated into a separate capsule, on the assumption that both capsules will be installed on the same machine. The default operations on the trading system ensure that their service interfaces will be exported to, and

imported from, the local catalogue. The result is the establishment of a purely-local D-association, mapped onto the IPC facilities of the LEX.

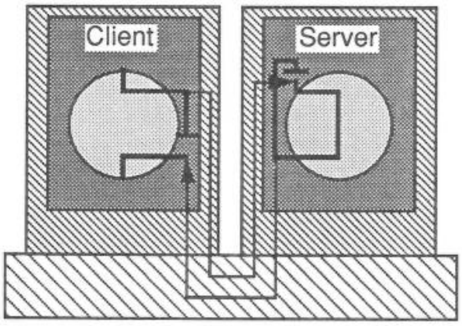
3) **Two capsules in separate hosts**: The two computational objects are generated as separate capsules, according to the particular type of host machine on which each is to be executed. The capsules may be replicated if required. The service interface is exported to, and imported from, some global catalogue, and the result is the establishment of a D-association that is mapped onto the communication system. Interactions between these capsules take place across the network.



Case 1: Two objects within one capsule



Case 2: Two capsules in the same local execution environment



Case 3: Two capsules in separate hosts

Fig. 10 - Three Versions of Inter-Object Procedure Call

7.3.4.2. Execution of an RPC from the Engineering Viewpoint. This section follows through the execution of an RPC, as seen from the engineering viewpoint, for the case where the service user and service provider are in separate capsules (figure 11).

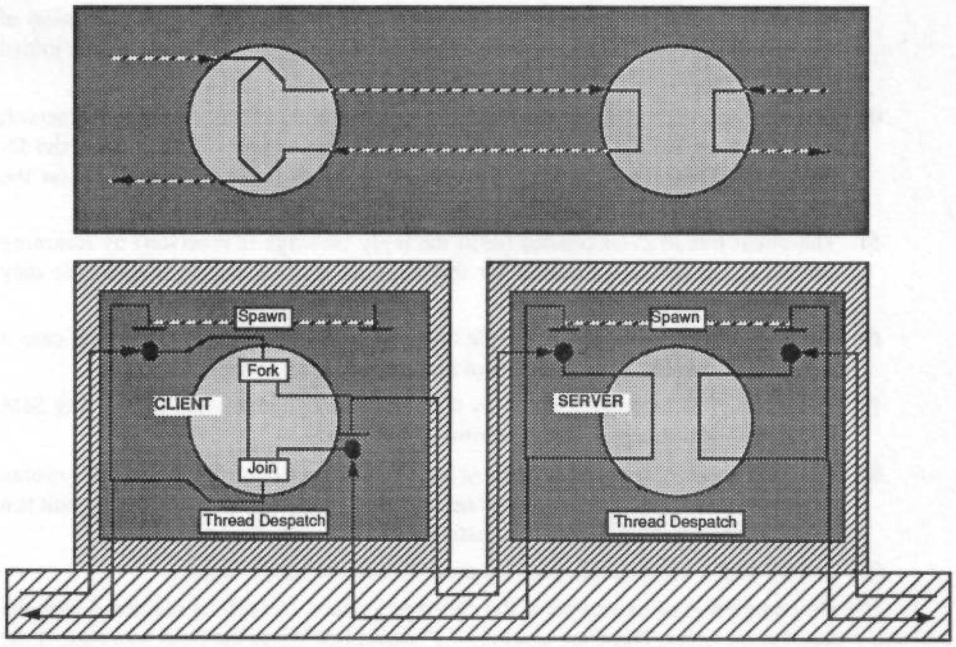


Fig. 11 - RPC in Engineering Viewpoint

At the language level, invocation of a remote procedure is the same as invocation of an external procedure. The procedure interface must have been defined in such a way that all the necessary information is passed in the parameters (since the context of the caller will not be accessible to the remote procedure) and that only parameters that are meaningful in the remote environment are specified. The remote procedure is defined as an external procedure, and therefore in a separate compilation unit. A *Client Interface Module* (CIM) is the local representative of the remote procedure, and forms the separate compilation unit required by the language system. A CIM is generated automatically from the service interface specification, compiled separately, and linked with the other modules of the capsule.

The capsule containing the service provider (server) is generated in a similar way, with the *Server Interface Module* (SIM) being the local representative of all remote service users.

The establishment of a logical interaction path (via a D-association) is described in section §7.3.3; typically, this would be done by the capsule initialization software of the service provider and service user.

At run-time, the sequence of events in the execution of a remote procedure call is as follows:

1) Invocation of the remote procedure results in a call on the corresponding procedure in the CIM.

2) On procedure invocation, the CIM assembles (marshals) the procedure call parameters (including the identity of the particular procedure) into a message.

3) In the general case, the client and server may have been generated using different language systems, and may be executed on hosts of different type. The procedure call parameters must therefore be converted by the CIM into a *transfer syntax*, such as ASN.1, which will be understood by the SIM. Where both the client and server use the *same* parameter formats, then the conversion is not necessary, with

consequent saving of time. The trading system should provide an indication of whether the use of a transfer syntax is necessary, at the time that the client is joined to the D-association.

4) The message is passed, via the LEX and the communication system, to the server, depending on the destination given by the trading system when joining the D-association. The CIM is dependent on the local operating system and on the interface to the communication system.

5) The client thread is suspended (until the reply message is received) by returning control to the thread scheduler for this capsule; activity within the capsule may proceed in other threads.

6) The server may not be able to handle the new request immediately, in which case it will be added to the server's queue of requests awaiting service.

7) The message is received initially by the *Server Interface Module* (SIM). The SIM acts as the local-to-server representative of the client.

8) The SIM unpacks the service-request message, converting from the transfer syntax to native syntax where an intermediate transfer syntax has been used; the result is a set of procedure call parameters in native form.

9) The SIM now invokes the specified procedure in the server object.

10) On completion of execution of the called procedure, the SIM is reentered with the result parameters. These are assembled ("marshalled") into the reply message; if the request message used the native transfer syntax, then so does the reply, otherwise the message is converted into the intermediate transfer syntax.

11) The reply message is then returned to its originator, using the communication system if the original was sent via the communication system and the local operating system.

12) The reply message may have to wait before it can be received by the service user. When the reply is received, the thread that was suspended (when the call was made) may resume execution from the point of suspension in the CIM.

13) The CIM unpacks the result parameters from the reply message, converting them to native syntax if an intermediate transfer syntax was used.

14) The results are returned to the computational object, and execution of the application code resumes at the point in the calling thread immediately after the procedure call, as if returning from a local procedure.

7.3.4.3. Implementation of Transparencies. The interface modules (CIM and SIM), along with the Deltase libraries that they use, play a major part in achieving the distribution transparencies introduced above:

- the underlying operating system primitives are hidden, giving *access transparency*;
- the bindings established via the trading system are provided in such a way as to give *location transparency*;
- by producing the client and server interface modules in the language of implementation of the respective client and server, and by use of an intermediate transfer syntax, *transparency of heterogeneous languages and machines* can be achieved;

- by making use of the services of the *Dependable Communication System* (DCS), and enclosing the local activity necessary for fault-tolerance, *replication transparency and fault transparency* may be provided.

7.3.5. The Handling of Threads

Concurrency of execution within a capsule is provided by *multiple threads*; each thread is a distinct "locus of control" within a given computational object. As described above, Deltase supports two types of thread (*server* threads and *forked* threads). The management of both types of threads is provided by Deltase as part of the envelope, created by Deltase/GEN.

Server threads are long-term threads, treated by Deltase as separate resources. Each incoming service request is assigned to a server thread that is currently free; each server thread can handle only one service request at a time. When execution of that service request is complete, the thread is suspended until it is required for execution of another service request. Server threads enable a capsule to handle several service requests concurrently.

Deltase treats all server threads as equivalent, so that an incoming service request may be assigned to any free server thread within that capsule. Such threads must not preserve any knowledge of previous assignments; any history that must be preserved is represented in the persistent data that is shared by all threads within the capsule.

When a capsule is initialized, several server threads are "spawned" to become dormant Deltase resources, for animation by a thread dispatcher in fulfilment of service requests. Should all the server threads become simultaneously animated, then any further service request must be retained (on a queue) until a server thread become available.

Other strategies could be adopted; a slower alternative would be to create new server threads on demand, deleting each on completion of its service.

Server threads provide concurrent invocation of independent services within one object; there is no true parallelism between these threads since, within a single processor, only one of these threads can be executing at a given time. However, the use of distinct server threads allows the management of concurrency to be provided by Deltase, freeing the application programmer of this responsibility, and allowing competition for access to a resource to be controlled locally.

7.4. Deltase Support for Distributed Fault-Tolerance

7.4.1. Introduction

Deltase combines ODP concepts on distributed systems with the Delta-4 approach to fault-tolerance. The support of fault-tolerance concerns the computational, engineering, and technology viewpoints.

- The **computational viewpoint**: The Deltase *computational model* is based on the requirement that any computational object conforming to this model can be made fault-tolerant, transparently, under Deltase. The Deltase computational model imposes constraints on the behaviour of a computational object, in terms of its external interactions and internal scheduling.
- The **engineering viewpoint**: It is from the *engineering viewpoint* that the mechanisms for the support of fault-tolerance are modelled. These mechanisms are provided by a combination of *generation support (Deltase/GEN)* and *run-time support (Deltase/XEQ)*.

- The **technology viewpoint**: It is from this viewpoint that the special subsystems for the support of fault-tolerance are modelled, such as the *Dependable Communication System* (DCS) which provides the essential distributed consistency (see §7.5.3), and the *fail-silent processors* that are required for the support of some of the models of fault-tolerance.

7.4.2. Replicated Capsules

The general Delta-4 approach to fault-tolerance is based on the support of *replicated software components*, (see §7.5). The *Deltase* approach to fault-tolerance is based on the support of replicated capsules, such that the support of fault-tolerance is transparent to the *computational object(s)* enclosed. A replicated capsule ensures survival of the state contained therein despite the loss of one of the nodes on which that capsule is represented.

A replicated capsule consists of a group of replicas, whose activity must be coordinated in such a way as to give the appearance of a single capsule. For the support for fault-tolerance to be transparent to a computational object, both the existence of the group of replicas and the activities to coordinate the group must be independent of the computational objects.

Delta-4 supports a number of different models of fault-tolerance (see chapter 6) for use under different circumstances, and the envelope is generated according to the particular fault-tolerance model to be used for that capsule. The activities of a group of replicas are coordinated in different ways for the different models of fault-tolerance.

Fault-tolerance requires that the group of replicas be able to survive the failure of a host (on which one of these replicas is executed) with consequent loss of that replica. For a given replicated capsule to survive over a "long" period, it is essential that replicas lost by host failure be replaced (see chapter 6). The process of replacing a lost replica is referred to as *cloning* (i.e., creating a clone of the remaining replica(s)), and involves the transparent creation of a new replica that is identical to the surviving replicas. The new replica is created on a host where there was no replica of that capsule prior to the start of the cloning process.

The set of possible hosts on which a given capsule can be installed is referred to as the *capsule replication domain*, (see section 6.4). All hosts within a given capsule replication domain are of the same type (and therefore support identical instruction sets), and with equivalent local execution environments.

D-associations are defined in such a way that, if a replicated capsule is a member of a D-association, then all replicas of that capsule are members of that D-association.

7.4.3. Determinism

The execution of a section of code is said to be *deterministic* if, for a given initial state and input message, execution of that section of code always results in the same final state and output message.

Replicas of a capsule do not automatically behave deterministically, even though all replicas of a given capsule:

- have identical code;
- are executed on hosts with identical instruction sets;
- start from the same initial state.

Figures 12 and 13 show two replicas with identical initial states, and an asynchronous event that is allowed to influence the sequence of processing. The asynchronous event might be any number of the following: some sort of time-slicing within the capsule, the end of a time-out, or an indication of message arrival. The replicas are loosely synchronized, and the asynchronous event occurs at different times with respect to the execution of that code:

- *after* message output in the replica on the *left* (in figure 12);
- *before* message output in the replica on the *right* (in figure 12).

In figure 12, the event is allowed to have immediate effect, and affects the two replicas in different ways resulting in different final states. However, in figure 13, the asynchronous event is trapped, and only allowed to take effect *after* the first message output; the result is that both replicas reach the same final state.

To ensure that behaviour of replicas is deterministic, asynchronous events have to be controlled; asynchronous events, which cause a transfer of control within the capsule at an arbitrary point, are not deterministic and cannot be permitted.

A *non-deterministic decision* is one for which re-execution would not necessarily lead to the same decision. With a capsule, all non-deterministic decisions are taken within the envelope, the execution of the code within the computational object being deterministic.

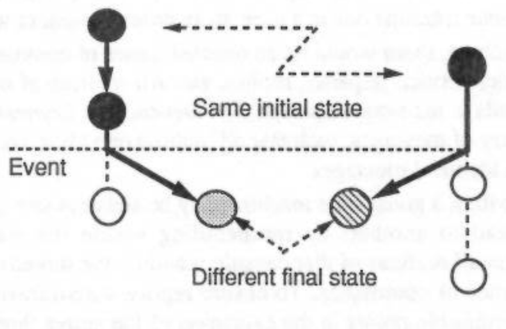


Figure 12: Non-deterministic behaviour

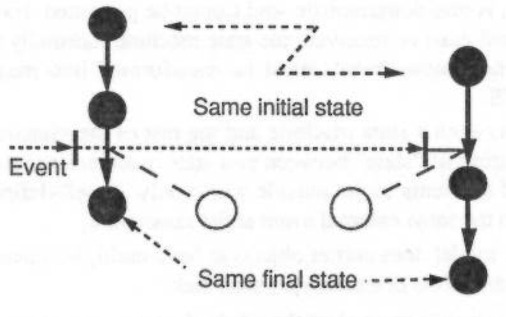


Fig. 13 - Deterministic Behaviour

7.4.4. Replica Execution

The term *replica group determinism* is used to refer to the deterministic behaviour of the replicas of a replicated capsule. The code to provide overall control of the capsule is dependent on the particular fault-tolerance model to be used, and is generated as part of the envelope during capsule-generation.

7.4.4.1. Active Replica Model. The active replica model (whether on fail-uncontrolled or fail-silent hosts) uses the *state machine* model to achieve long-term replica group determinism, without negotiation between the replicas.

Schneider [Schneider 1990] defines a state machine as follows:

> "A *state machine* consists of *state variables*, which encode its state, and *commands* which transform its state. Each command is implemented by a deterministic program; execution of the command is atomic with respect to other commands, and modifies the state variables and/or produces some output. A client of the state machine makes a request to execute a command. The request names a state machine, names the command to be performed, and contains any information needed by the command. Outputs from request processing can be to an actuator (e.g., in a process control system), to some other peripheral device (e.g., a disc or terminal), or to clients awaiting responses from prior requests. (...) Requests are processed by a state machine one at a time, in an order consistent with causality."

For each state machine, there would be an ordered queue of commands to be serviced; the term command includes service requests, replies, etc. All replicas of that state machine must process these commands in the same sequence. The *Dependable Communication System* (DCS) ensures ordered delivery of messages, such that all replicas of a given replicated capsule receive identical sequences of identical messages.

Only one thread within a given state machine may be active at any given instant. Changing from one active thread to another, by rescheduling within the state machine, must be deterministic, such that all replicas of that capsule schedule the threads in the same sequence, given the same sequence of commands. To ensure replica determinism, thread rescheduling may only occur at identifiable points in the execution of the active thread, such as sending or receiving a request. The new thread is determined by the next "command" in the sequence, with all replicas receiving the same sequence of commands (messages).

"Preemption", causing a change from one active thread to another at points other than these clearly-defined points, is non-deterministic, and cannot be permitted. Having arrived at the new state, the next command *must* be received; the state machine (normally the whole capsule) can do nothing else. Asynchronous events must be transformed into messages that are sent as commands via the DCS.

All interactions between a state machine and the rest of the system must take place using commands. Direct sharing of "state" between two state machines cannot be permitted. A state machine may respond to events in the outside world only at well-defined points, at which all replicas can respond to the same external event at the same point.

The state machine model does permit objects to have multiple concurrent threads, *provided* that the scheduling of threads is deterministic, such that:

- all replicas schedule equivalent threads in the same sequence,
- the transfer of control from one thread to another occurs at the same point in all replicas.

Under Deltase, each capsule is treated as a separate state machine, the state variables comprise the total state of that capsule, and the commands include both the service requests (for services offered by that capsule) and the replies (from services used by that capsule).

7.4.4.2. Leader-Follower Model. This model requires fail-silent hosts, since correctness depends upon the behaviour of a single replica (but see section 7.6); this is the only fault-tolerance model used in the XPA variant of the architecture (see chapter 9). One of the replicas operates as the leader, and the other replicas operate as followers. It is the leader that determines the course of execution within the capsule; all non-deterministic decisions are taken by the leader. The leader informs the followers of the course of execution to follow, by the use of short messages (which are not visible to the computational objects).

In its simplest form, the leader-follower model may be considered as a variant on the state machine model, the difference being that under this variant, the leader can *select* which message to process next, rather than be obliged to accept the next message from the ordered queue. Such a decision is not deterministic, but since the host is fail-silent and the decision is passed to the followers, all replicas will process the same message next.

In a more sophisticated form, the use of *preemption points* enables the queue of incoming messages to be scanned more frequently than would otherwise be possible. Without preemption points, the message queue can be scanned only when the envelope is entered during the normal course of execution of a computational object. Preemption points allow the processing of one service request to be *preempted* by a service request of higher precedence; at each preemption point, the envelope is entered to inspect the message queue and to permit rescheduling. Preemption points may be set by a software tool that plants a suitable statement in the source code of the computational object; hence preemption points occur at identical points in all replicas of a given capsule, and preemption can be assured to occur at the same point in all replicas.

The support of preemption points is an important part of the support, by Deltase, for the combined requirements of fault-tolerance and real-time behaviour.

With the leader-follower model, all *non-deterministic* decisions are taken by the code within the envelope of the leader. The code within the computational object must be executed deterministically; once the leader has decided on a particular path, then all replicas must follow that path until the next change of path, as determined by the leader.

7.4.4.3. Passive Replica Model. This model also requires fail-silent hosts. Since only one replica is active at any given time, it is this replica alone that determines the execution path; the state in the standby replicas is updated by means of checkpoints, (see chapter 6). Code within the envelope generates (and sends) checkpoint messages when the replica is in active mode, and receives (and processes) checkpoints when in standby mode.

7.5. Engineering Support for Fault-Tolerance

This section covers those aspects of the engineering model where support for fault-tolerance is visible.

7.5.1. Transformer Capsules

A *transformer capsule* is a software component generated (by Deltase/XEQ) from one or more elementary transformers. (A transformer capsule may also include one or more computational objects.) A transformer capsule will thus have at least some *programmed* interactions with the non-Deltase world.

Because of such interactions with the outside world, the deterministic behaviour of replicas (of a transformer capsule) cannot be guaranteed; the result of an external interaction may, or may not, be identical for all replicas of the transformer capsule.

Where the transformer capsule is merely forming a bridge between two replicated worlds (for example, between Deltase capsules and replicated MMS software components), there is replicated behaviour on both sides of the transformer capsule.

However, where the transformer capsule is providing an interface to local services, then the transformer is also having to reconcile:

- the Deltase world: portable and generic software, with transparent support of replicas, and
- non-replicated local environments: existing applications, local operating systems, device drivers, etc.

A (replicated) capsule, offering a global service, may have to use a locally-provided service based on a local device. One example is that of a global file server that ultimately has to use a number of local file servers for persistent storage (on disc). Another example is that of making available, to users in the Deltase world, the services of proprietary database software (provided as a local server that accesses the disc directly; see §7.6). None of the Delta-4 support mechanisms can assure that these local servers are equivalent, in the replica sense; each of the local servers may have a different set of (local) clients.

A *transformer capsule* may be used to provide a Deltase "front end" to each of the local servers, each transformer capsule being co-located with its associated local server. These transformer capsules provide an interface to the Deltase world, but they cannot be assumed to exhibit identical behaviour.

These transformer capsules are not replicas, but *rivals* (see chapter 12); each presents a *distinct* service interface (although these interfaces will all be of the same type). A (replicated) application module, interfacing to *all* the given set of rival transformer capsules, can derive an agreed behaviour from the observed behaviour of the rivals; this agreed behaviour can then be made available to other computational objects via a service interface (see figure 14).

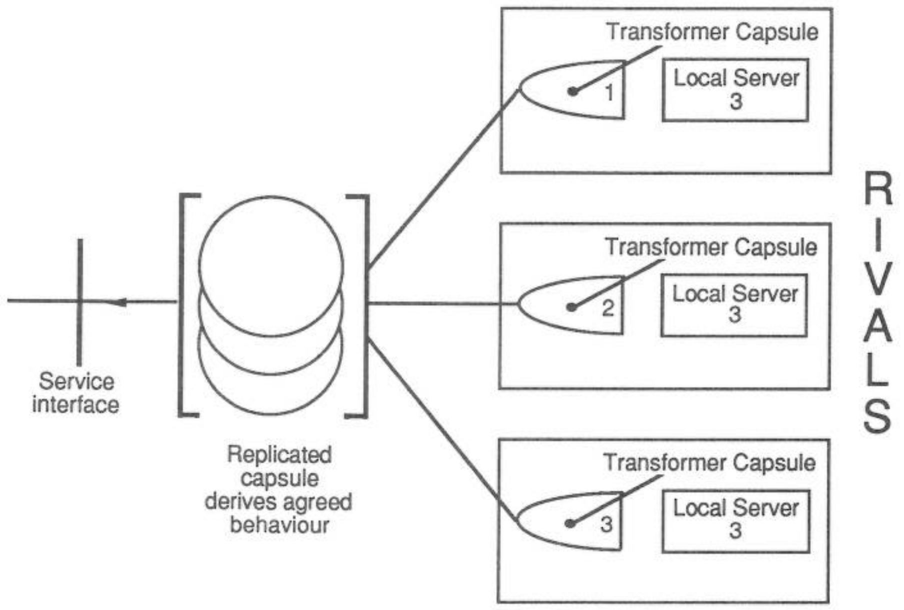


Fig. 14 - Rivals

In practice, various simplifying assumptions may be made; a common assumption is that the transformer capsules can be treated as replicas (which would normally be true for "most" of the time).

The use of transformer capsules as rivals, with an application module to derive an agreed behaviour, leaves the two worlds (Deltase and the local environment) unaltered.

7.5.2. Capsule Installation and Cloning

This section concerns capsule installation and capsule cloning. These topics are closely related to the system administration software (see section 8.2 for OSA system administration, and section §9.5 for XPA system administration).

Deltase/GEN produces a "file"; this file acts as a template, from which an instance of that capsule can be established on a host machine where the capsule is to be executed.

Installation is the process of establishing a clean instance of a capsule on a selected host; that instance will start executing from its initial state. If the capsule is replicated, then the replicas start together, from their initial execution point, in loose synchronism.

Cloning is the process of establishing a new *clone* of an existing partially-executed instance of that capsule. Following the loss of a host, typically because of a fault on that host, all the capsules executing on that host are lost. The level of fault-tolerance of the services provided by those capsules is then reduced. In order to restore the level of fault-tolerance, it is essential to create new replicas to replace those that were lost.

A new replica can only be installed on a host of the particular type for which the capsule was generated, and with the particular type of LEX. The new replica must be set up to have the *same state* as the existing replica(s) of that capsule, so that a new replica can play its full part (depending on the particular fault-tolerance model) as soon as possible. This will then minimize the "window of risk" during which the number of replicas is below the level specified. Any break in service (while cloning takes place) must also be kept within the timing constraints acceptable for that service.

The generic components responsible for the installation, cloning and termination of capsules are as follows:

1) **Replication Domain Manager (RDM).** The RDM is responsible for the overall control of the installation, cloning and termination of capsules across all hosts in a Delta-4 system. Events, such as station failure, which may require the cloning of one or more software components, are passed to the RDM; it is the RDM that implements the cloning policy (see section 8.2) and determines on which host a new clone is to be created. The RDM is, itself, a software component, and can be replicated for fault-tolerance.
2) **Local factory.** There is a local factory on each host in the system, to carry out the instructions of the RDM on that particular host. A local factory is an example of a transformer capsule, appearing to the RDM as a capsule, but interacting directly with the LEX to perform the cloning (see figure 15).
3) **Object Manager Entity (OME).** The OME is part of the envelope, to carry out those operations that can only be done from within the capsule. The OME consists of a set of library procedures, which are included in the envelope by Deltase/GEN.

The creation of a new replica of a capsule involves combining information from three sources:

1) **Initial state** (including code and related data) is taken from the file created by Deltase/GEN.

2) **Current state** information is taken from the existing replica(s). The means by which the current state information is supplied varies according to the fault-tolerance model used.
3) **Local state** information is data relating to the allocation, by the LEX, of resources in the local environment, such as buffer addresses or file identifiers. Such information is not visible to the application programmer, but is necessary for the library procedures that interface to the LEX. This information is only valid locally, and must be handled locally, by the OME.

The operational mode of the new replica depends on the particular fault-tolerance model used for that capsule.

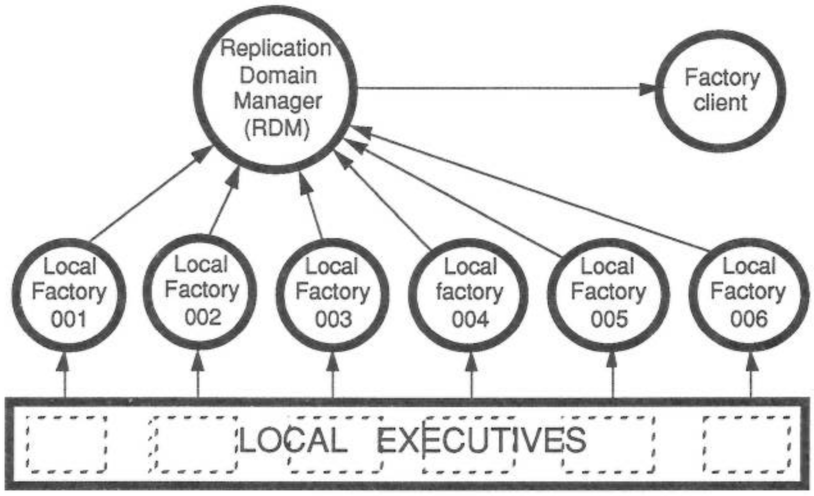


Fig. 15 - RDM and Local Factories

7.5.3. Dependable Communication System

Within a distributed system, a communication system is used to interconnect the distinct hosts that execute the applications software. The Delta-4 approach to fault-tolerance is based on the use of a communication system that provides explicit support for fault-tolerance; such a communication system is called a *Dependable Communication System* (DCS). A general objective of a DCS is to offer *distributed consistency*, to ensure consistent behaviour in distinct nodes.

To support fault-tolerance, Deltase must be used with a suitable DCS. In certain implementations it may be possible for some of the properties necessary for the support of fault-tolerance to be provided by suitable additions to the Deltase libraries, although there would normally be a performance penalty for doing so. Deltase hides the communication system from the application programmer, and neither the communication system nor its use is visible from the computational viewpoint.

Two dependable communication systems are being developed within the Delta-4 project:

1) the *Multipoint Communication System* (MCS, see section §8.1), used in the *Delta-4 Open System Architecture* (OSA), is an OSI-based communication system, for use in open dependable distributed systems;
2) the *Collapsed Layer Communication System* (CLCS, see section §9.6.4), used in *Delta-4 Extra Performance Architecture* (XPA), provides higher performance and support for real-time behaviour in systems having the necessary homogeneity.

7.6. Dependable Databases

Transformers can be used to include *commercially-available database software* within the Delta-4 system architecture, in such a way that the databases can be made *fault-tolerant*, by using replicated copies of the database. The database software is used without modification.

Clients use the standard database call interface for the particular database software, but this interface is treated as a Deltase service interface. Invocation of one of these procedures results in a service request to the database transformer. In principle, different clients could use different interfaces to the database. A number of clients may enjoy concurrent access to the replicated database.

A *transformer* is used to represent the database as a Deltase object, enabling other Deltase objects to use the services provided by the database software. The principle is illustrated in figure 16. Although the principles are generic, the details of this transformer are dependent on the particular database software.

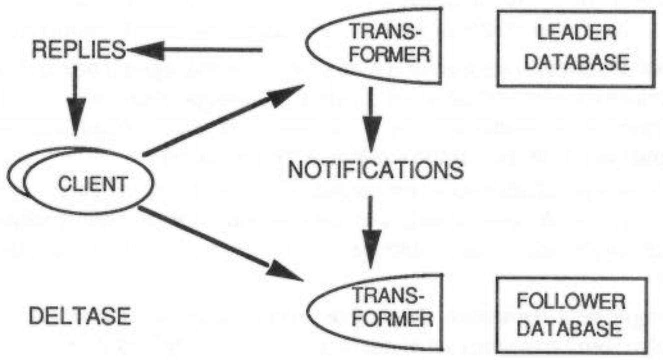


Fig. 16 - The Dependable Database

The database transformer is not a true state machine, and replicas of the transformer differ in internal operation, though not in externally visible effect. In the terminology of [Bond et al. 1987], they do not follow *identical* execution paths, but they do follow *equivalent* paths leading to identical final states.

To ensure replica group consistency, all changes to the database must be executed in the same order in all replicas of the database. This is achieved by using variants of the *leader-follower model* of fault-tolerance (see chapter 6). However, different variants of the model are required to support the database server on fail-silent and fail-uncontrolled hosts.

The variant for *fail-silent* hosts is considered first. The database transformer in one host acts as the leader, and all other instances of the database transformer (in separate machines) act as followers. The leader transformer performs updates in an opportunistic order, and then notifies the followers of the order in which the updates are to be performed. Since the leader host is fail-silent, these notifications are either correct or not produced. If they are not produced, the (first) follower takes over the role of leader.

Locks (including "shared lock" commands preceding read requests to the database) have to be treated in the same way as updates. Pure read requests, which do not change the state of the database, need normally be executed only by the fail-silent leader; only if the leader fails need the new leader execute the read. However, some read requests are implicitly also update requests and must therefore be performed by all replicas; for example, they may increment a pointer to the data being processed.

The leader also replies to all client requests; when it replies (using an atomic or reliable multicast message — see section 9.6.4), the notification to the follower(s) is included in the same multicast. The effect is that the client receives a reply from the leader if, and only if, the follower(s) receive a notification. So, if the leader fails, the (first) follower can avoid duplicating replies; hence the client receives exactly one reply in all circumstances.

Dependable databases can also be supported on *fail-uncontrolled* hosts. The leader-follower model can still be used provided it is combined with the active replication model, as follows:

1) The conventions of notification can be arranged such that an erroneous leader cannot mislead a correct follower. In the leader-follower model, replicas achieve deterministic execution by using the notification as a basis to choose identically from a set of alternative execution paths, any one of which is equally correct.
2) The notification must be expressed so that all correct replicas can easily determine whether it specifies one of these correct paths. If it is correct, there is no problem, but even if it is erroneous, all correct followers may be able to use it to identify a unique correct execution path. If not, they inform System Administration that the leader has failed, and wait until the new leader issues a new notification.
3) Since erroneous behaviour cannot be imposed upon correct followers by an erroneous leader, replicas are sufficiently independent to have their correctness assessed in the same way as is done for fail-uncontrolled active replicas; their interactions with the outside world may be compared.

An example of a notification convention that is too low-level and could propagate error is: "Jump to instruction *I*". A more natural alternative having the properties required is "choose *K*", where *K* identifies one of an enumerated set of alternative execution possibilities. Two cases then arise:

- *K* might be a legitimate alternative, even if proposed by an erroneous leader to resolve some erroneous set of choices. All correct replicas then follow the path *K*.
- If *K* is not one of the legitimate alternatives, each follower independently derives an opinion on the correctness of the leader, and expresses this opinion to permit it to be validated. In this case each follower can continue in the short term by deterministically resolving an acceptable alternative by, for instance, evaluating *K modulo H*, where *H* is, in the opinion of the follower concerned, the number of legitimate alternatives available.

In both these cases, all correct replicas will take the same path and express the same opinion in all interactions with the outside world. These interactions are therefore validated, and any incorrect outputs are detected and masked, assuming the correct replicas are a majority.

This strategy does not automatically bound the latency of errors, since, as with other applications, many normal interactions with a database do not reveal much of the internal state. Thus, for example, a fail-uncontrolled host can return a perfectly correct acknowledgement of a database update request, having miswritten the update to disc. There are two ways to reduce the error latency:

1) The error latency can be bounded by periodically reading the whole database, so that every piece of data is validated.

2) The reply to the update can contain extra replica state information, which is validated but not returned to the client. This reduces the probability of latent errors.

Another problem with databases and some other applications is that users may be permitted to read data without first locking it. This gives the user some performance benefit and assured freedom from deadlock, but the data read is not dependable, since it may be inconsistent or incorrect at the time of use. No model can confer full dependability on such operations, unless it effectively locks the data before the first read, which interferes with the user's semantics.

Database users may also be permitted to read multiple rows of data without specifying the order in which they are to be presented. The database replicas can return differently ordered replies to such a request. The validation routine that compares such replies must recognize them as effectively identical so long as they contain the same rows of data, regardless of order.

The validation of notifications and replies has a performance cost that only arises for databases on fail-uncontrolled hosts. However, the performance cost can be reduced if the leader sends the notification before it starts processing the update. The validation and processing of the notification in the followers then normally overlaps with the processing of the update in the leader, so that replicated replies to the client are available for comparison as early as possible.