CROSSNET
SYSTEMS LIMITED

Unit 3, The Paddock
Hambridge Road
Newbury
Berkshire
United Kingdom

Project: DBV-OSI II

Project: ONE – OPAC Network in Europe

Z39.50 Application programmer's Interface

SYSTEM
ARCHITECTURAL INFORMATION

Document No.: D-020
Document Version: 2.0
Z39.50 API Software Version 2.1

July 1997

Prepared by:	J. Hough	Crossnet
	R. Bull	Crossnet

This document is derived from the DBV OSI II project SES (System External Specification) document D-020 produced by Crossnet Systems Ltd. This document is produced under the scope of the ONE project,
LIB-ONE/2-3099

DOCUMENT STATUS SHEET

DOCUMENT STATUS SHEET			
1. DOCUMENT TITLE: **DBV-OSI II Z39.50 API System Architectural Information**			
2. ISSUE	3. REVISION	4. DATE	5. REASON FOR CHANGE
Issue 1	1	20 Nov. 96	Based on DBV OSI II D-020 Document
Issue 2	1	31 Jul. 97	Incorporates comments for additional supported Services.

Table of Contents

Table of Figures

1. Introduction

This document describes the modules developed by Crossnet Systems Ltd of the DBV OSI II Z39.50-1995 API. This document describes the operation and interfacing between the API and user application and recommends an approach for developing Origin and Target systems.

This document was originally produced for the DBV OSI II project for use by the Partners in the DBV-OSI consortium.

This version of the document reflects Version 2 of the API, which has been extended by Crossnet Systems Limited in their participation in the ONE project, (OPAC network in EUROPE). ONE is co-funded under the European Commission Telematics (Libraries) programme. project number `LIB-ONE/2-3099`.

1.1 Abbreviations and Nomenclature

The following abbreviations and nomenclature are used in this document.

ACSE	Association Control Service Element
AIX	IBM Workstation UNIX environment.
ANSI	American National Standards Institute.
APDU	Application Protocol Data Unit. Synonymous with PDU.
API	Application Programmers Interface.
ASCII	American Standard Code for Information Interchange. Typically used as a synonym for readable text.
ASN.1	Abstract Syntax Notation 1. A design and specification language.
A-Association	Application related Z39.50 state
BER	Basic Encoding Rules. Standard rules used for encoding and decoding Z39.50 PDU data units.
BSD 4.x	Berkeley Software Distribution 4.3. A 'flavour' of UNIX provided as the default operating environment on certain UNIX workstations including SUN.
DBV-OSI II	Project for which this document applies
FIFO	First-In First-Out. A UNIX named pipe which allows inter-process communication
GCC	GNU 'gcc' ANSI C compiler
GDB	GNU 'gdb' source code debugger
GNU	Gnu's Not Unix. The prefix name given to a range of products produced by the Free Software Foundation.
OSI	Open Systems Interconnection. A communication architecture.

IR-Target	Information Retrieval Target. A Z39.50 Target program.
ISODE	'ISO' Development Environment. A collection of library routines and programs that implements an extensive set of OSI upper-level services. The 'ISO' part of the system name has no meaning.
O-Machine	Z39.50 protocol Operation state machine
PDU	Protocol Data Unit. A logical data unit used by Z39.50.
RPN	Reverse Polish Notation.
SCCS	Source Code Control System. A UNIX version control program.
SELECT	A BSD 4.3 socket programming function.
SES	Software External Specification.
SR	Search and Retrieve.
snacc	Sample Neufeld ASN.1 to C/C++ Compiler. An ASN.1 to C/C++ Compiler.
TCP/IP	Transmission Control Protocol / Internet Protocol.
tbl	Table Format. This is a file format which has been developed at Crossnet.
X.25	Recommendation describing the access interface to packet-switched data networks.
Z39.50	A standard produced to facilitate the interconnection of computer systems.
Z-Association	See Z-Associativity
Z-Associativity	Protocol related Z39.50 state.
Z-Machine	Z39.50 protocol Z-Associativity state machine

Where the document refers to implementation within the service-user component of a program, this is to be considered as a recommendation, and is clearly marked as such.

1.2 References

1. ANSI Z39.50-1995 Search and Retrieval Standard.

2. Colbert E, (1989), The Object Oriented Software Development Method: A Practical Approach to Object-Oriented Development, TRI-Ada 1989 ACM.

3. Wilkie, G, Object-Oriented Software Engineering: The Professional Developer's Guide, (1993), Addison-Wesley Publishing Company, ISBN 0-201-62767-1

1.3 Supported Features of the Z39.50 Protocol

Version 2.1 of the software supports:

Z39.50 Services Supported (by both Origin and Target):

- Initialise
- Search
- Present
- Segment
- Result-Set Delete
- Scan
- Resource Report
- Sort
- Access Control
- Resource Control
- Trigger Resource Control
- Extended Service
- Close

Suported Extended Services:

- Item Order
- Update

Supported Record Types:

- All MARC Records listed in the Z39.50 Standard
- Explain
- Unstructured Text (SUTRS) Record
- Opac Record
- Summary Record
- Extended Service Task Package Record

Other Suported EXTERNALS:

Resource Report 1
Resource Report 2
Access Control Prompt-1
Inter-Library Loan (ILL ISO-10160, 10161) (request, status)
Character and Language Negotiation Record

Known Supported UNIX Platforms:

SUN-Sparc Sun-Os 4.1.2, Solaris 2.5
Hewlett Packard PA1.1, (HP-UX 9.05)
Siemens Nixdorf (SINIX-N 5,42)
DEC Alpha (OSF/1)
IBM RS6000 (AIX 3.2.5)
Linux 1.1.47 (Slackware)

The following are known to be ported to by users of the software:

DEC Ultrix
Data General

2. Developing an Application

In an Origin or Target application using the DBV OSI II Z39.50-1995 API, the application is to be considered as having two parts,

- the API itself, which is the service-provider part;

- the application which is the service-user part.

2.1 Responsibilities Provided by the API

The DBV OSI II Z39.50-a995 API comprises:

- **Origin Service Provider** - These are the API functions which provide association control and Z39.50 encoding/decoding/data transfer routines for an Origin application. Their functionality is defined in the document - DBV-OSI II API-Definition Document.
- **Target Service Provider** - These are the API functions which provide association control and Z39.50 encoding/decoding/data transfer routines for a Target application. Their functionality is defined in the document - DBV-OSI II API-Definition Document
- **Utilitiy functions** - These are the API functions which provide various functionality useful to the service user programmer of either Origin or Target applications. Their functionality is defined in the documents - DBV-OSI II API-Definition Document.
- **Test Tool** - The test tool is a suite of programs which will be used to verify the functionality of the API (Origin service provider, Target service provider and utility functions). Reference document DBV-OSI II Test Tool Manual for complete details.
- **Target Daemon** - This is a TCP/IP daemon utility which can be used as a front-end daemon for receiving Origin associations.
- **Origin and Target Test Programs** - These are useful application programs which provide example use of the API incuding source code.

2.2 Responsibilities of the Application Developer

With the DBV OSI II Z39.50-1995 API, an application developer may develop the following:

- **Origin Application** - An Origin application may be a standalone application or it may be part of a larger system.

  In the case of the standalone application, the application will typically be a bespoke user interface designed to offer the user search and retrieval operations.

  In the case of the Origin being part of a larger application, the API may be integrated within the larger system or may be instigated as a result of a user requesting to access a remote host. In this latter case, the Origin application may be spawned from a daemon that is connected from the local host.

In either case, the application has to effectively manage user operations and map these to the appropriate API data structures and call the appropriate API functions. The return values of the API functions have to be interpreted by the Origin application and appropriate information given back to the user.

In any application, the developer should be conversant with the operation descriptions described in the ANSI Z39.50-1995 Specification document.

- **Target Application** - The application will be spawned by the Target daemon that is supplied with the API software, or by some other appropriate mechanism. The Target will include routines to control an association to the remote Origin application (provided by the service provider Target API), routines to control an association to the local host database (provided by the service user), routines to decode the incoming binary data stream and complete the correct DBV Z39.50 structures (service provider Target API), routines to transform the DBV Z39.50 structures into the local host query language and send the data to the local host (both service user). For responses from the local host database that are to be sent to the to the Origin, routines must be provided to receive local host database data, translate from the local host format into the API structures, encode the structure data into a binary stream and transfer them to the Origin (provided by the Target service provider)

NOTE: Whilst the API interprets the Z39.50 state behaviour during the operation of an application, the full responsibility of adhering to the Z39.50 state machine for both the Origin and Target falls to the service user application.

3. System Architecture

This Section describes possible architectures of the Z39.50-1995 Applications when using the DBV OSI II Z39.50-1995 API. A basic architecturan design is provided, from context analysis to application architecture. All architectural design is decomposed to a level whereby implementation independence is maintained.

Further decomposition is deemed to be application specific and is the responsibility of the developer. Since all implementations are to be on UNIX systems, then the architectural design for each platform should be very similar.

In the following subsections, the Colbert 91 methodology is used. Whilst there are many methodologies suitable for describing system architectures, this methodology is chosen since it can be used in the analysis, design and state definition of the software. In addition, it is suitable to an Object-Oriented design to which the Origin and Target programs can be suited. Information on the Colbert methodology can be found in Ref. 1 and Ref. 2. In addition, Appendix A includes diagrams pertaining to the Colbert methodology syntax.

3.1 Context Architecture

The context architecture of a Z39.50 session is shown in Figure 3.1 below. It shows the system with respect to its external entities. In this case, the external entities are the user and the Target database system.

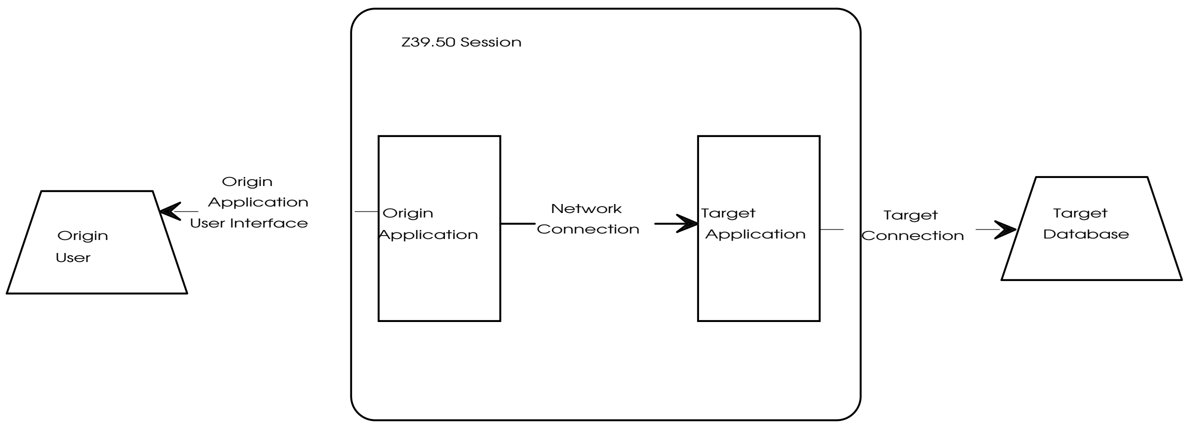


Figure 3.1 Context Diagram of a Z39.50 Session

In the context analysis, the User and the Target Database are external entities to the Z39.50 session and are therefore not the subject of design in this context.

The connections are defined as:

Origin Application User Interface:

The user interface designed for the user. Note that this may be a connection to an existing local host system that is offering an external gateway to a remote Z39.50 Target.

Target Connection:

The interface between the Target application and the Target host database system.

It should be noted that the Network connection between the Origin and Target applications does not consider which network is used, but simply to describe that the two are connected by a network.

For the context of the DBV OSI II project and the ONE project, another contextual view is given below in Figure 3.2. This shows the context of the virtual network architecture, where existing local systems are offering Origin access to remote systems for local users, and offering Target access to remote users via their Origin systems. In the case of the ONE project, not all partners will provide access to remote hosts via their existing system, but will use standalone Origin packages.

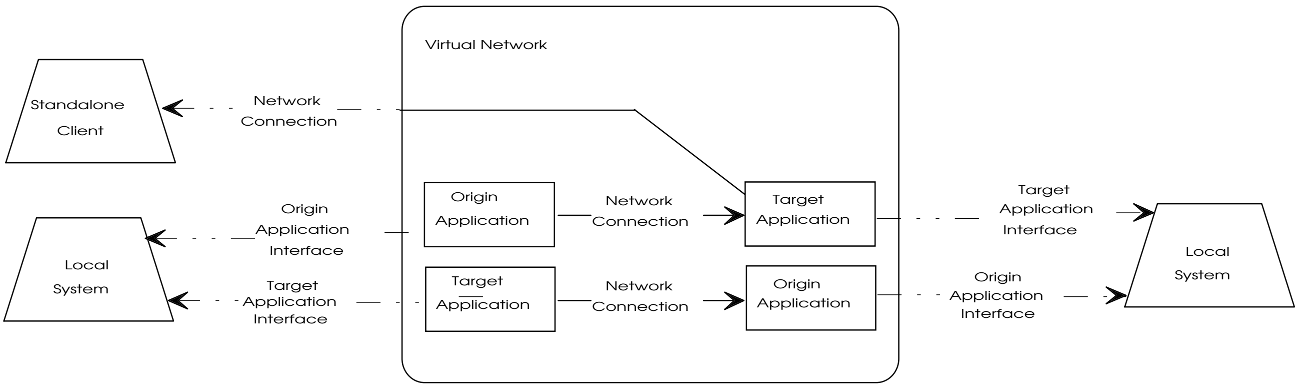


Figure 3.2 Context Diagram of a Z39.50 Virtual Network

In Figure 3.3 below, The next stage of analysis is to decompose the DBV-OSI II Application object itself. This is shown in Figure 3.2 below.

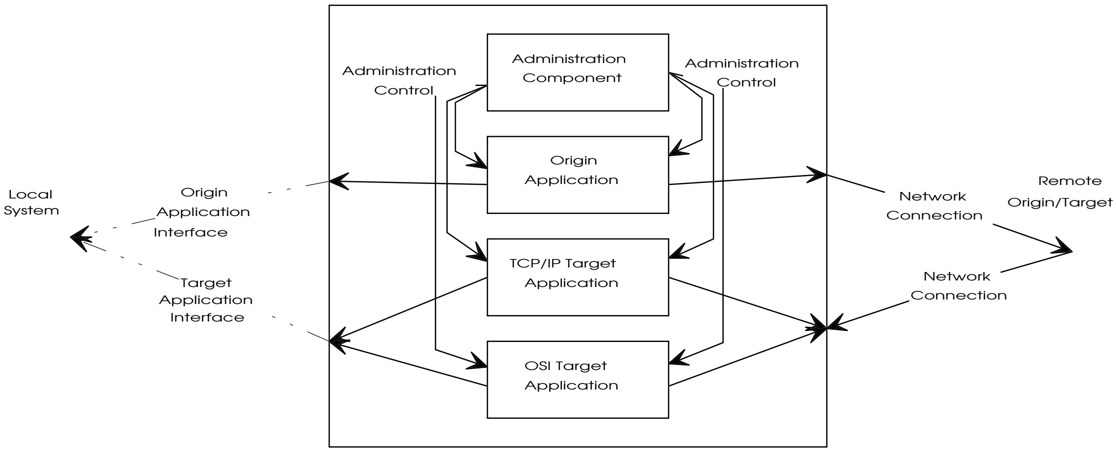


Figure 3.3 Decomposition of Z39.50 Components Associated with a Local System

In this diagram, it can be seen that there are four types of component associated with a local system, namely:

Administration Component: This object is responsible for initiating and destructing the other objects. In practical terms, it can turn-on/off the local daemons for the Target application, or could turn on/off a daemon associated between the local system and the Origin components. This component is an optional facility and is not mandatory in a system.

Origin Component: This object is responsible for managing Z39.,50 Origin association and service requests to a selected Z39.50 Target.

TCP/IP Target Component: This object is responsible for processing incoming association and Z39.50 service requests from a remote Z39.50 Origin over the TCP/IP communication stack.

OSI Target Component: This object is responsible for processing incoming association and Z39.50 service requests from a remote Z39.50 Origin over the OSI communication stack.

It is the responsibility for an application developer to develop the service-user part of the Origin/Target applications and any respective connection to the Administration Component. The service-provider part of these programs is in the API.

In the context of Figure 3.2 above, it is assumed that suitable connectivity to the local system will be provided. For example over a local area network, or even on the same machine that the local system resides on. In addition, a local system that provides connectivity to remote targets must cater for the switching to the Z39.50 Origin component when a user selects a remote Target.

If this connection is over a LAN, a TCP/IP connection is assumed whereby the local system will act as a TCP/IP client, and the Origin Component will contain a front-end TCP/IP daemon.

This mechanism will enable multiple connections from the local system to the Origin Component to be utilised. The design of the Origin Component will determine how multiple connections are handled. Some advisory architecture is given below.

The TCP/IP- and OSI- Target Components will wait for incoming connections from remote DBV-OSI II Origin components through use of suitable receiving server processes.

For each connection received by the TCP/IP- and the OSI- Target Components from a remote DBV-OSI II Origin Component, a unique Target process should be instantiated by the appropriate daemon. Each instance of a Target component will need to connect to the Target host database. A suitable proprietary mechanism will be used for this, such as a RPC interface, telnet-like interface or proprietary API.

3.2 Multiple Connections

For the Z39.50 Target, it is recommended that the conventional master-slave architecture is adopted to handle multiple concurrent connections. The philosophy behind this approach adopts the following procedure:

a) a Target daemon program waits for incoming association requests from origins;

b) for each Origin process that the association is accepted, an Application process is spawned and is passed the association file descriptors into its main() function. The spawned process will inherit the daemon environment (including the file descriptors) and will assume communication to the Origin directly. This then relinquishes the file descriptors from the daemon which is free to accept further connections from origins (up to the physical or system-defined limit of the machine);

c) the Target negotiates with the Origin specific ports/addresses to use between the Target process and the associated Origin.

This is known as a master-slave approach, in which the slave is the Application program which is instantiated for each Origin connection - there is no need to retain any association Id or duplicate information structures.

It is recommended that for the DBV-OSI II Application Component programs, that this architecture is adopted. The advantages of this architecture are:

1) the Origin and Target Component programs are simpler in architecture,

2) There is a safety feature whereby if a Target program inadvertently terminates only that process is affected - all other processes (and hence users) are not affected.

The disadvantage of this architecture is:

1) More memory and resources are required during system operation.

3.2.1 Origin Component Architecture - Standalone

The Z39.50 Origin Component of a standalone Origin package assumes that only one user will exist at any time. In this case, the Origin application provides a user interface for the user to perform Target selection, association and search and retrieval. All details of these features are entirely for the application designer and not the subject of this document.

3.2.2 Origin Component Architecture - Local System

The Origin Component that is part of a local system should utilise a multi-process architecture to support multiple users. In this sense, the API will need to incorporate both OSI and TCP/IP communications, and the API functions will be used to determine which is required for connection to a particular Target.

The Origin Component itself could contain a TCP/IP daemon to receive local host Origin sessions and run in the master-slave architecture described above. This architecture is shown in Figure 3.4 below.

Note that the API shown in the Origin Processes within the dotted lines. The remainder is the responsibility of the application developer.

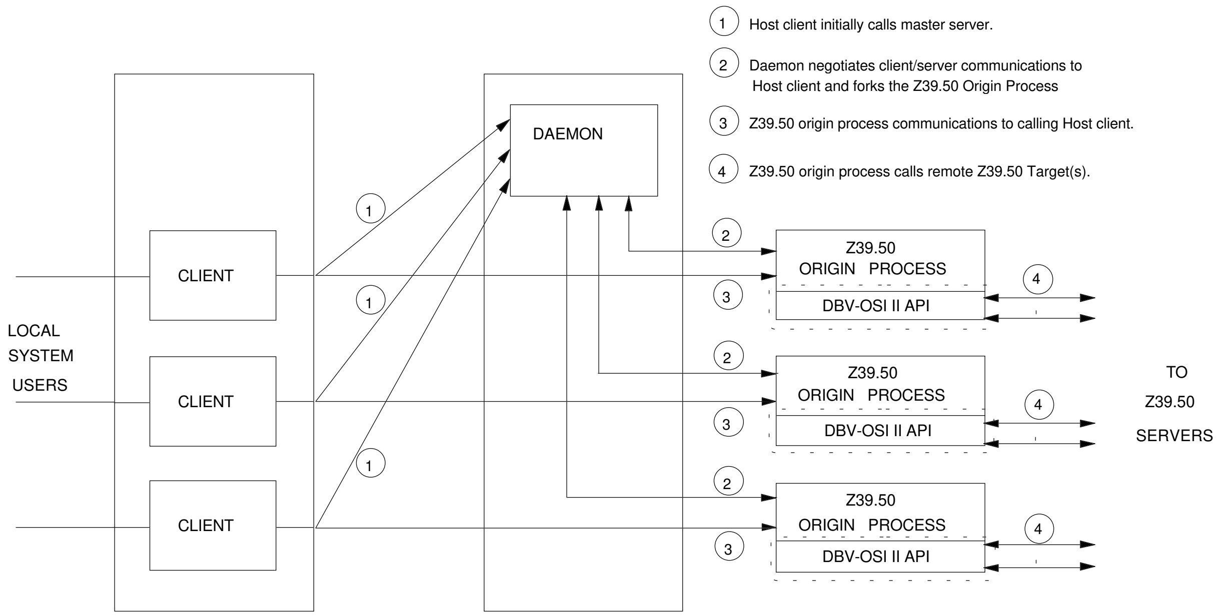


Figure 3.4 Possible Origin Architecture for Local System

The architecture above can be considered as a Colbert-91 Object-Interaction Diagram (OID with each user and its associated Origin process represented as an object. This is illustrated below in Figure 3.5.

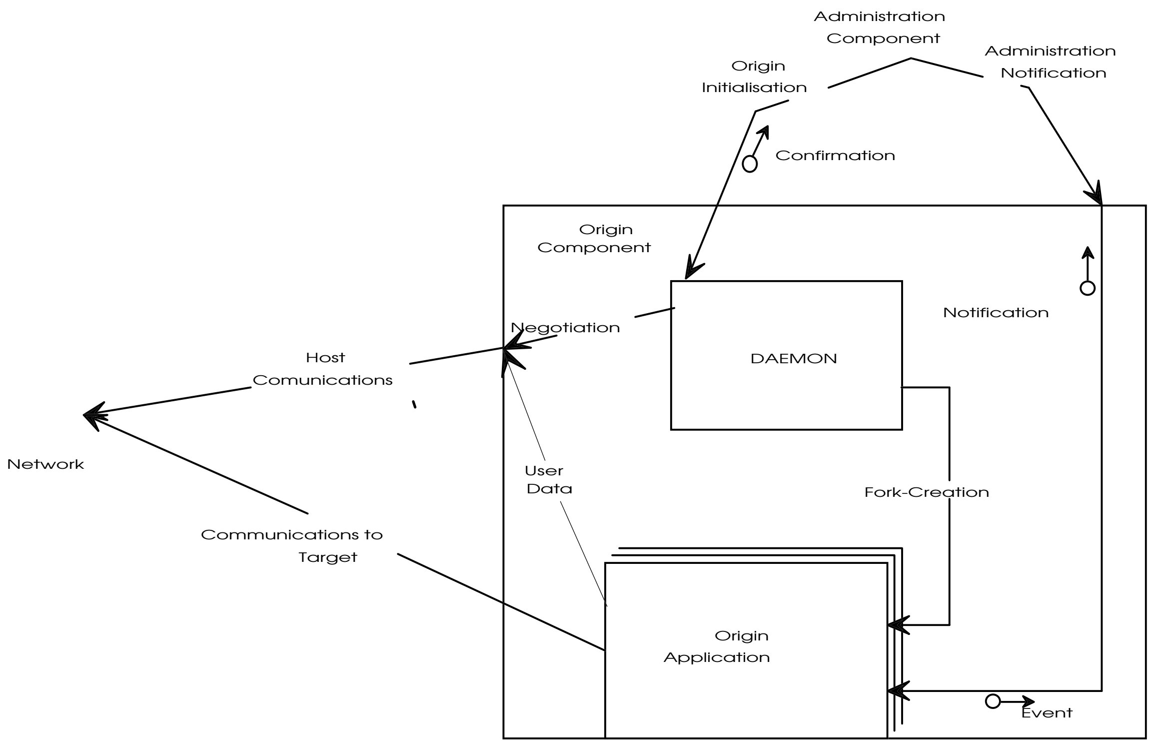


Figure 3.5 Local-Host Origin Architecture

The architecture shows two main objects, namely a TCP/IP daemon to process user session connections from the local system, and the Origin Application object. The architecture shows these as being two distinct objects which will communicate with a standard UNIX mechanism such as pipes. In terms of implementation, the TCP/IP daemon will spawn the Origin Application as a separate program. Each user session is managed by separate process instances, and hence the "collection" symbol for the Origin Application is used.

The Administration Component is shown as a means of stopping or starting the daemon. The design of this daemon is very similar to the Target front end TCP/IP daemon. Using this method allows the administrator to initialise or shut down the service without effecting any other TCP/IP services on the system.

The Origin Application may notify the Administration component of various Event messages. Since any number of concurrent Origin State Machines will exist at any time, each Event message will be tagged by the UNIX process Id of that particular Origin Application.

3.2.3 TCP/IP Z39.50 Target Component Architecture

The design of the Z39.50 Target Component can be quite complicated, depending upon how much of the Z39.50 protocol is to be supported. The design should also take into account development issues such as development debugging and operational reliability. A block diagram of a possible Target architecture is illustrated in Figure 3.6.

The development of the communication between the Target Application and the host is fully the responsibility of the application developer.

Note that the API services are shown in the Target Processes within the dotted lines. The remainder is the responsibility of the Target developer.

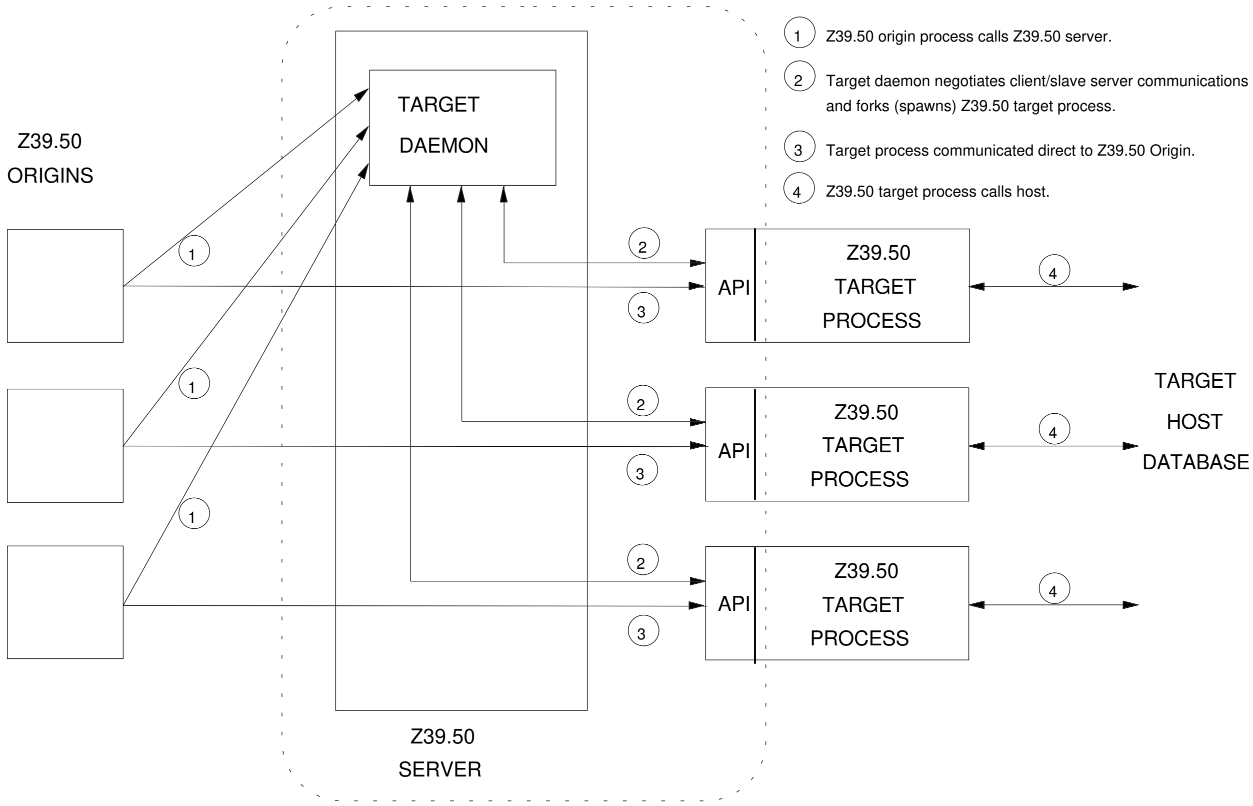


Figure 3.6 Block Diagram of Target Architecture

The trade-off of this architecture is that more memory in the UNIX computer is used when there are multiple concurrent associations, and hence multiple process instances.

By utilising the above architecture, the O-O decomposition of the TCP/IP Target Component is shown in Figure 3.7 below. Referring to this diagram, a brief explanation of its behaviour is given after.

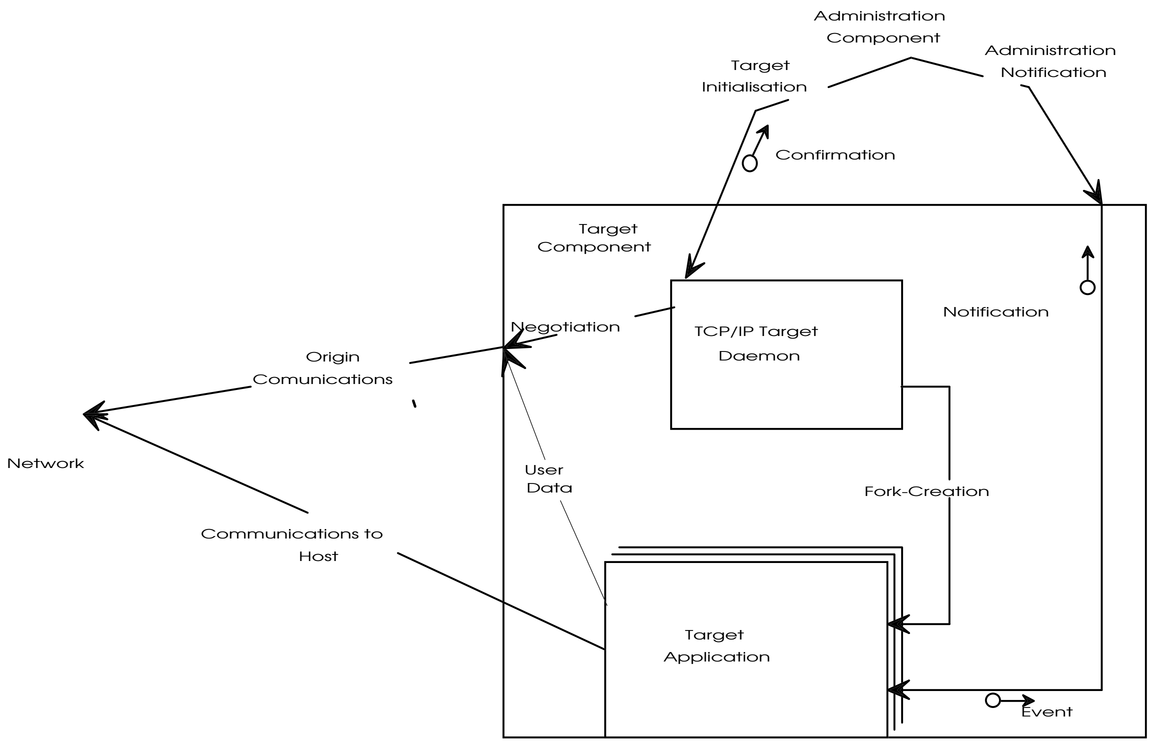


Figure 3.7 TCP/IP Z39.50 Target Architecture

The TCP/IP Target Component comprises two main objects, namely a TCP/IP daemon to receive remote host Origin connections and the Target Application object. The architecture shows these as being two distinct objects which would communicate with a standard UNIX mechanism such as pipes. In terms of implementation, the TCP/IP daemon spawns the Target Application as a separate program. Each user session is managed by separate process instances within the Target Component, and hence the "collection" symbol for the Target Application is used.

The Administration Component is shown as a means of stopping or starting the daemon. The design of this daemon is very similar to the Target front end TCP/IP daemon. Using this method allows the administrator to initialise or shut down the service without effecting any other TCP/IP services on the system.

With respect to the Administration Component, the Target daemon may be started (or stopped) as required by the Administration Component operator. The same mechanism will be used as starting or stopping the Origin front end server.

The Target Application may also notify the Administration component of various Event messages. Since any number of concurrent Target Applications may exist at any time, each Event message should be tagged by the UNIX process Id of that particular Target Application.

The supplied program **targetd** is an example TCP/IP daemon and is provided as an easy means of testing different TCP/IP Target applications without the need for changing the **inetd** related Unix system files. This program can be easily configured to listen on any socket and execute any program with parameters which will be correct for `TargetInitialize()` API function.

targetd can be started by moving to the `release/targetd/` directory and issuing the command:

```
targetd [-p portnumber] [-e path/filename]
```

where `[-e path/filename]` is the Target process to execute.

3.2.4 OSI Z39.50 Target Component Architecture

The architecture of the OSI Target Component is shown in Figure 3.8 below. It is analogous in architecture to the TCP/IP Target Component. Referring to this diagram, a brief explanation of its behaviour is given below.

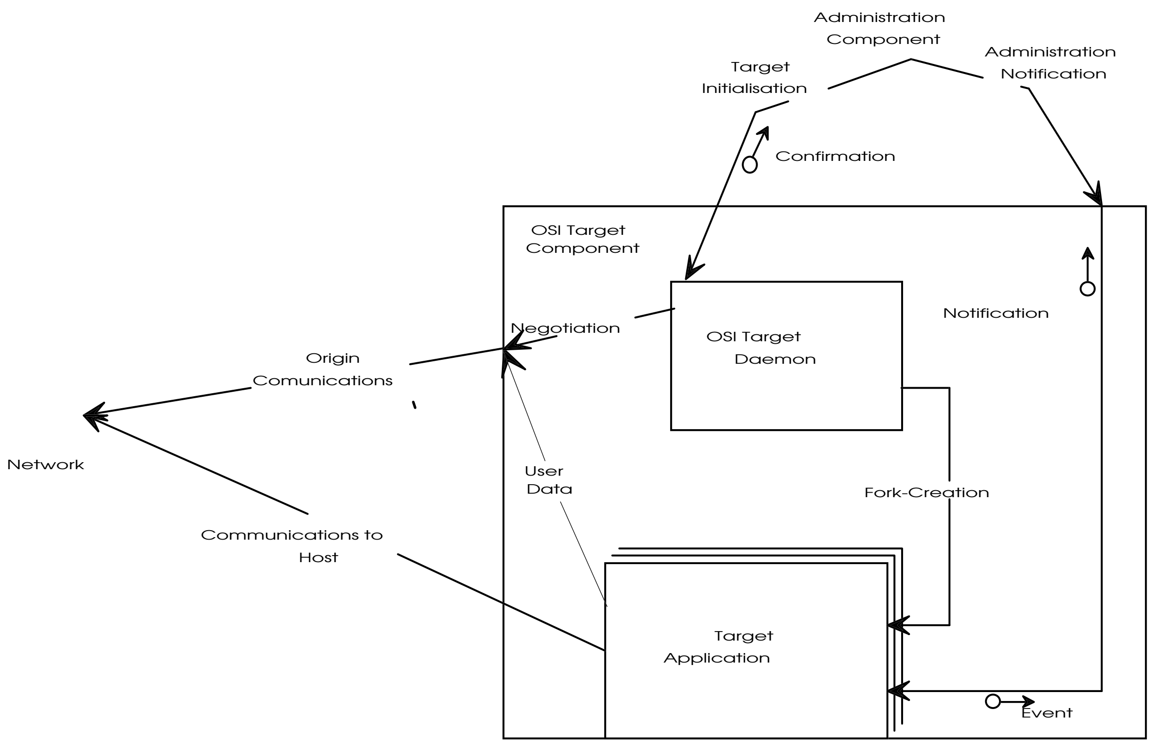


Figure 3.8 OSI Z39.50 Target Architecture

The OSI Target Component comprises two main objects, namely an OSI daemon to receive remote host Origin connections and the Target Application object. The architecture shows these as being two distinct objects which would communicate with a standard UNIX mechanism such as pipes. In terms of implementation, the OSI daemon spawns the Target Application as a separate program. Each user session is managed by separate process instances within the Target Component, and hence the "collection" symbol for the Target Application is used.

With respect to the Administration Component, the Target daemon may be started (or stopped) as required by the Administration Component operator. It is suggested that the daemon provided in the ISODE package is used, (tsapd).

tsapd is provided with the ISODE package. When the Administration Component is started it will check to see if the OSI daemon is already running and report its status. If the administrator wishes to disable OSI connections to the Z39.50 service the process will be stopped and a daemon configuration script file substituted with one which does not include the Z39.50 service. The process would then be re-started. This has the disadvantage that OSI communications will be temporarily made unavailable, but only for a few seconds. For enabling the OSI daemon for Z39.50 services the configuration script would be replaced with one containing the service and the daemon program re-started. If the daemon is not started, a facility to start it will be provided.

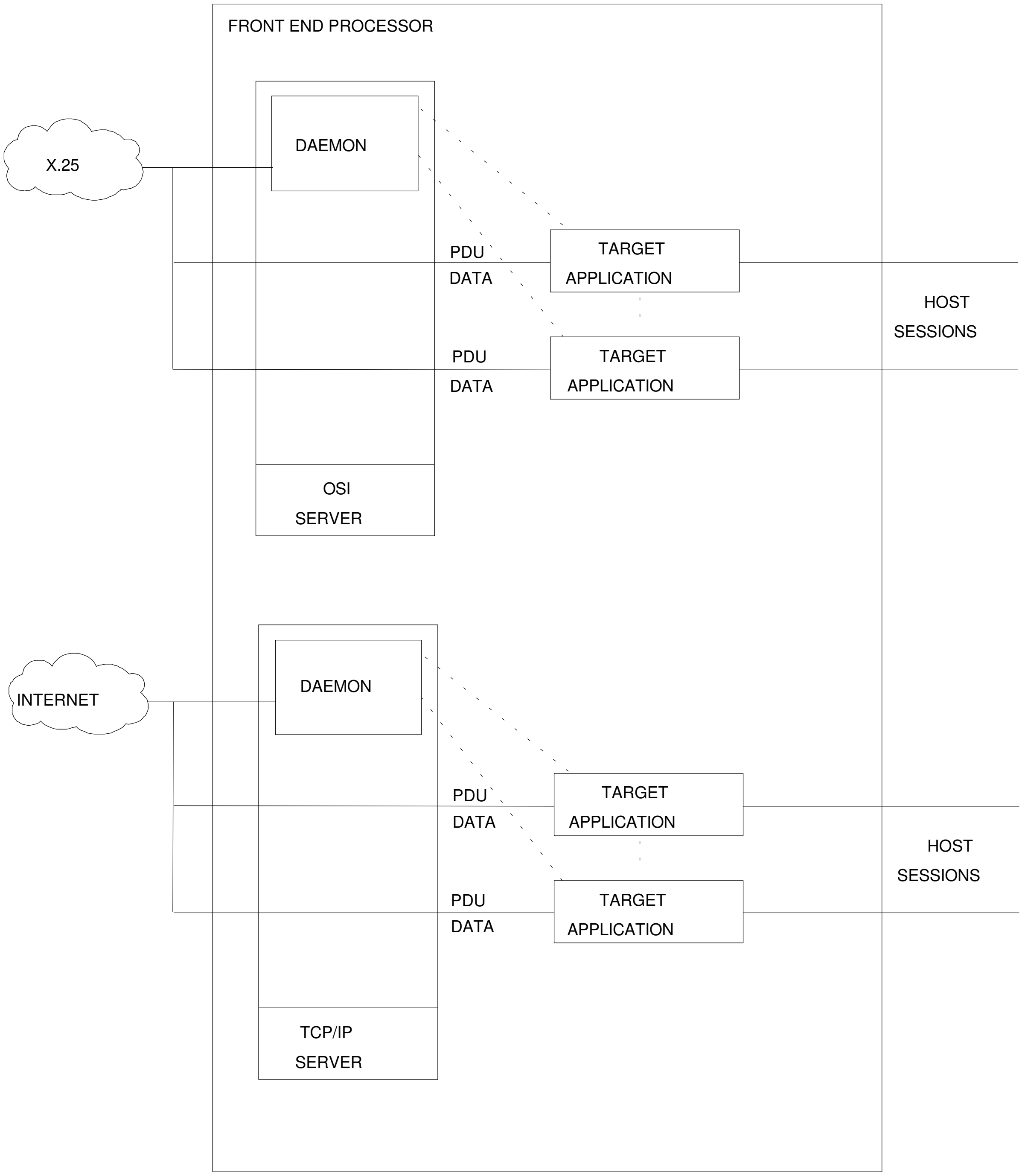
Note that tsapd is provided with the public domain release of ISODE**.** However, for the ISODE Consortium Release 2, an additional daemon has been supplied called **iaed** which is the recommended daemon to use. The use of **iaed** will have no impact upon the Origin or Target processed to be developed by the project Partners. However, since **iaed** has a more automated approach to its administration operation, there will be a slight difference in the operation of the Administration Component.

The Target Application may notify the Administration component of various Event messages. Since any number of concurrent Target Application will exist at any time, each Event message will be tagged by the UNIX process Id of that particular Target Application .

3.2.5 Targets with Multiple Communication Stacks

The architecture described above can be seen to readily support different communication stacks as illustrated in Figure 3.9. A separate daemon will exist for (say) OSI to that of TCP/IP and both will utilise a similar mechanism to spawn the same Target Application program. This will be achieved by the daemon (Either TCP daemon or OSI Daemon) calling the Target Application program passing a parameter to it implying which daemon invoked the program. In either case, the data between the daemons and the Target applications will be encoded APDUs - the API functions within the Target application are responsible for the encoding and decoding of the data as part of the service-provider component.

The Target architecture presented is also highly desirable if the site wishes to initially support one communications stack protocol and then support another protocol at a later date, since the Target application program is identical for each.



Note: Target Application program is same for either stack.

Figure 3.9 Architecture with Multiple Stack Support

4 System Development Considerations

This section details specific System development considerations of the processes and programs in the DBV-OSI II Applications.

4.1 Origin And Target Application Analysis

The development of the Origin and Target applications should adopt a further decomposition of the architecture defined above. In this sense, the requirements of these programs will need to be fully analysed. From the point of this document, certain inputs to this analysis are formally defined. Other inputs to the analysis are dependent upon any specific operation that the implementor wishes to include. Both inputs are listed below:

Formally defined Inputs:

a) State Machine behaviour of Z39.50;

b) definition of DBV-OSI II API;

c) high level architecture; and

d) Administration Component event notification.

Non-defined Inputs:

a) exact mechanism for Host to communicate with Origin Component;

b) mechanism to incorporate interface to local system;

c) low level architecture for Origin to handle multiple associations;

d) mechanism to associate local Host users with remote host accounts and passwords; and

e) any inter-system charging mechanism.

4.2 Service Provider State Machines for the Origin and Target

The functionality of the service provider API modules follows a well defined path. While the API cannot contain the complete Z39.50 state machine, the API will perform some state machine and error checking, notably:

1) that only valid APDU data fields will be accepted at any state instance;

2) there will be no invalid repetition of API functions being called; and

3) no API function is called before conditions necessary for the function to operate exist

For the latter point, this will include simple state checking such as ensuring an association exists before an APDU encode and send function is called.

In terms of the service-user components, the following are recommended:

a) The Protocol Z-Association and operation functions should not be called until an association connection has successfully been accomplished to the Target;

b) The Protocol O-Association operations should not be called until a valid Z-Associativity has successfully been accomplished by calling the Initialisation function;

c) All structures and parameters for all Association functions should be correctly generated before calling any Z- and O- Association request functions; .

d) If an error is returned by any of the service-provider functions, then appropriate action should be taken.

In the cases a) to c) errors will be returned by the service provider if the rules are not adhered to.

State machine verification that the Origin and Target service provider can perform is illustrated in Figures 4.1 and 4.2 below.

Note that the entry point to the state machine in Figure 4.1 is labelled 'At any Time'. This means that at any time, the service user may create a new association by calling the API function AssociateRequest().

Note that for Version 2 of the API software, that the API supports Access Control and Resource Control which are challenges initiated by the Target to the Origin. In particular, the Resource Control challenge may or may-not require a Resource Control response, which is determined by the responseRequired field in the Resource Control request.

Note also that for Version 2 of the API software, that the API supports Trigger Resource Control Request, to which no response is necessary.

It should be noted that Version 2.1 of the API Software supports Access Control, Resource Control and Trigger Resource Control which are not depicted on the diagrams below. Please refer to Sections 3.2.5, 3.2.6 and 4.2.3 for how these services interact in the state machine model.

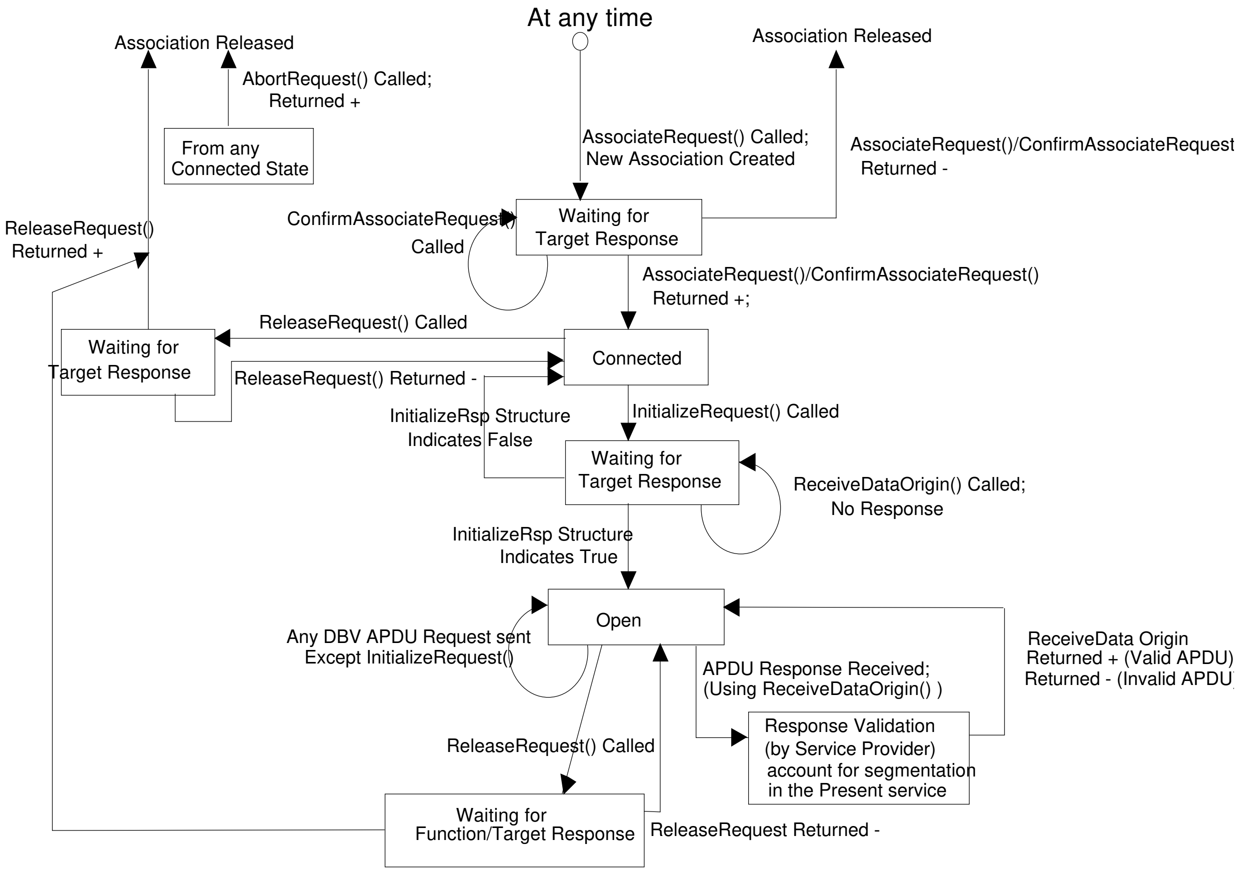


Figure 4.1 State Transition Diagram of Origin Service Provider

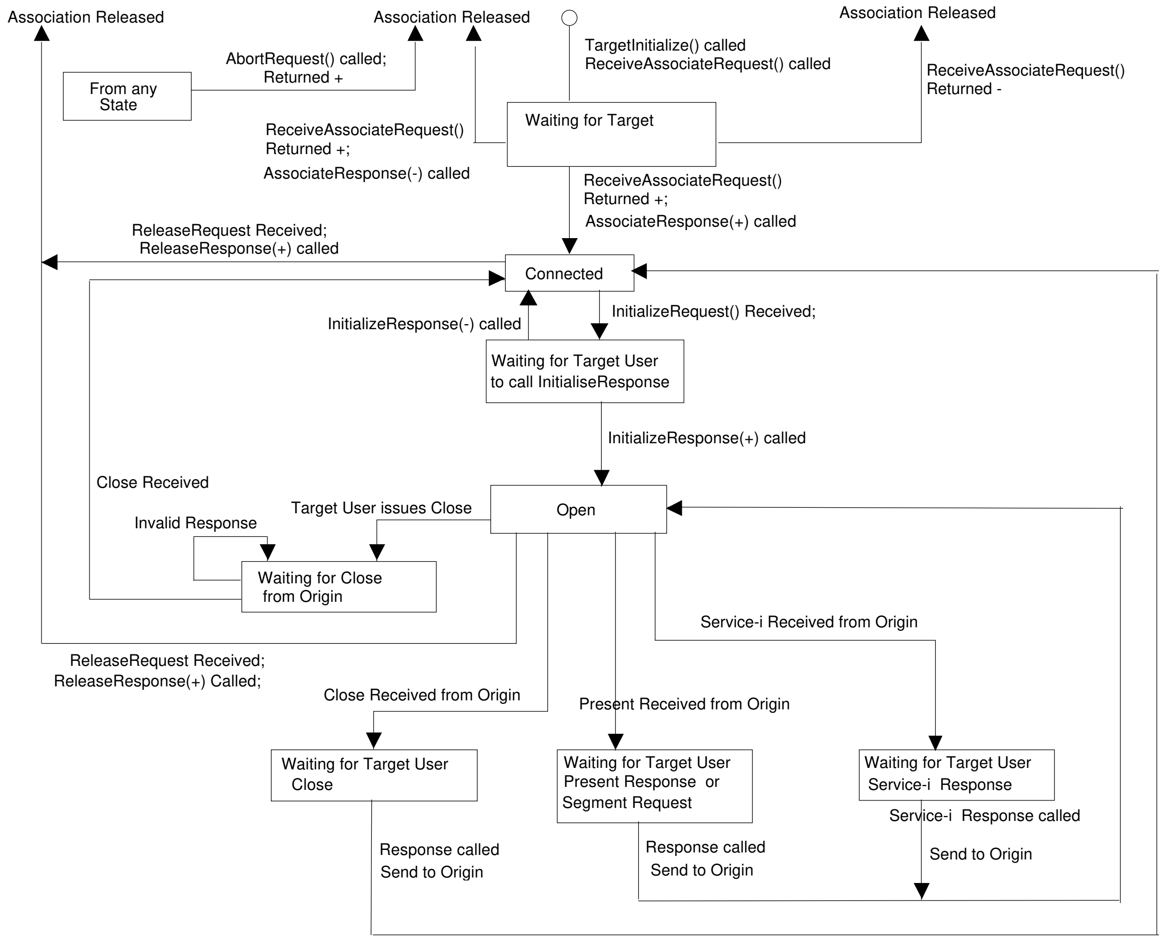


Figure 4.2 State Transition Diagram of Target Service Provider

Figures 4.1 and 4.2 show the state transition diagrams for the Origin and Target service provider. These are the rules which the service user should follow to successfully use the Origin and Target API components. If the service user deviates from these steps then the internal API state machine will give an appropriate error.

Notes on the state machine behaviour:

Note that the state machines only show the valid *service provider* transitions between states. In the states labelled 'Waiting for Origin/Target Response' for example, it is quite valid for there to be a large amount of service user activity.

A call to ReleaseRequest() will fail if there are any application specific responses still to be received. A call to ReleaseResponse() will fail if no ReleaseRequest() has been received from the Origin, or there are outstanding responses to be sent.

Please note that concerning APDUs the API will check for the following :

- Repetition of the InitializeRequest() or InitializeResponse() functions
- Sending an APDU before a valid association
- Calling ReleaseRequest() while there are outstanding responses
- Incorrectly completed or incomplete DBV APDU data fields.

a) Protocol Z-Association and operation functions will only be valid if an association connection has successfully been accomplished, (and not inadvertently lost);

b) Protocol O-Association operations will only be valid if a Z-Associativity has successfully been accomplished, (and not inadvertently lost); the Protocol Initialisation must be the first Z-Association operation to take place;

c) When a protocol O-Association operation is issued at the Origin, only the corresponding Operation response for that Z-Association will be valid. Exceptions to this are when Access or ResourceControl are being used; and when Segmentation is used in the Present Operation.

d) If the service-user calls the service-provider to await for an input event, any loss of service or service error will be immediately indicated to the service-user.

e) When a protocol O-Association request is received at the Target, the Target service provider will verify that it has not been issued before the Initialisation. i.e. InitializeRequest APDU must be received at the Target.

   When a valid O-Association request received and passed to the service user, the service provider will ensure that only the appropriate service response can be sent back to the Origin.

4.2.1 Utility functions

As well as the Origin and Target libraries a set of functions known as the **Utility Functions** will be provided. These functions will be useful to both the Origin and Target programmer. The list below explains at what states the functions may be called (refer to Figures 4.1 and 4.2):

- **SIGetEventLocation()** - Can be used in place of the ReceiveDataOrigin() and ReceiveDataTarget() functions to check for incoming APDUs (although ReceiveDataOrigin/Target() must be used to complete the DBV structures). Can also be used to check service user file descriptors at the same time, or service user file descriptors *only* as the service user sees necessary.

- **SISetACSEFile()** - Used (by the Origin application only) at any time it will effect all subsequent calls to AssociateRequest().
- **SIWhichStack()** - This function can be used at any state in the Origin or Target on a currently valid association.
- **SISetTraceParameter()** - This function can also be used at any state in the Origin or Target on a currently valid association.

All utility functions contain error routines to check for incorrect usage, so an invalid call to any of the above will always be detected and reported.

4.3 Origin And Target Application State Machines

Although the DBV-OSI II API will provide facilities to allow for Z39.50 APDU specification and transfer, it is the responsibility of the application developer to implement the full Z39.50 state change machines within the Origin and Target Components.

The Z39.50 standard clearly defines the state machine behaviour for both Origin and Target. The Z39.50 Origin state machine defines for a given set of states the input events that cause a particular set of Origin outgoing actions to occur. Inputs may be from the Origin application, the Target or be internally generated. The Z39.50 V3 Origin state machine is detailed in section 4.2 of the Z39.50 Specification.

In addition, the Z39.50 standard clearly defines the state machine behaviour that a Target process must conform to. The Z39.50 Target state machine defines for a given set of states the input events that cause a particular set of Target outgoing actions to occur. Inputs may be from the Target application, the Origin or be internally generated. The Z39.50 V3 Target state machine is also detailed in section 4.2 of the V3 Specification.

Code to implement V3 Origin and Target state machines is non-trivial. Most V2 Origin state machines are simple two dimensional arrays. Because of the complexity of the V3 definition with multiple possible state changes from other states, such a simple data structure is not suitable.

The Target Application as a whole must conform to the Z39.50 state machine. This state machine will be built into the service user

The API state machines cater for the specific condition of Resource Control request in which no response is necessary. In addition, it caters for the TriggerResourceControl request, which maybe used to terminate an operation although TriggerResourceControl request may be completely ignored by the Target.

In terms of implementation, the service-provider component will perform verification to a subset of the state machine pertaining to the *network operations*. This will follow the rules defined below:

4.3.1 Origin State Machine

This section discusses particular aspects of the Origin State Machine. In coding the state machine program, each particular event, input and state are given specific names in the Z39.50 Specification. These can be seen in Section 5.2.3 of the Z39.50 specification. It should be noted however, that all the events and actions are abbreviated and ideally the application program should adopt a similar naming convention. What is not apparent from the Specification, is the relationship of each event and action with respect to either Target, Origin, input or output.

The behaviour of the Origin State machine should conform to the description in the ANSI Z39.50 specification. This is an overall state machine comprising service-provider and service-user elements. In terms of state transition, and in terms of the state machine being built into an actual application, the following two diagrams depict an overall program state transition and a first level decomposition.

In the following diagrams, the notation is that a box indicates a state, arrows indicate a change to a different state. The arrow labels specify: the condition for the state-change to take place above the line; and, the actions taken in changing state below the line.

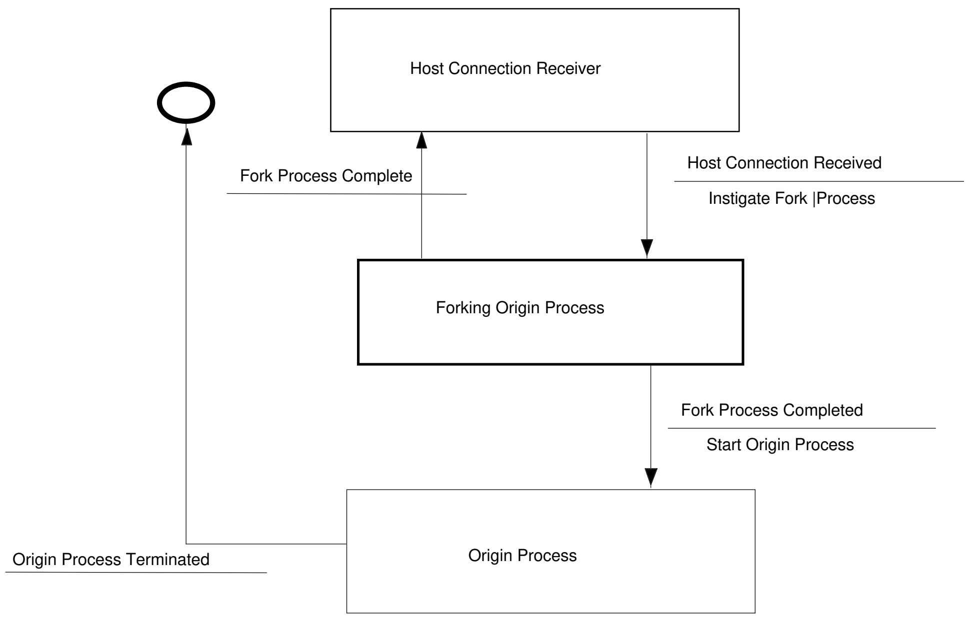


Figure 4.3 High Level State Transition of Origin Process

The high level state transition diagram of Figure 4.3 above refers to the Origin architecture as part of a local system. When the Origin Host connection daemon receiver receives a host/user connection, it will cause a state transition to spawn the Origin program. Note that the spawning transition will return state immediately to the host connection receiving state, and that when the Origin process is complete it has no return state since the program will terminate.

Figure 4.3 below shows the state transition behaviour of the Origin Application process itself. This diagram should be read in conjunction with the state behaviour in the Z39.50 Specification.

Note that the state transition information caters for both service-provider and service-user components.

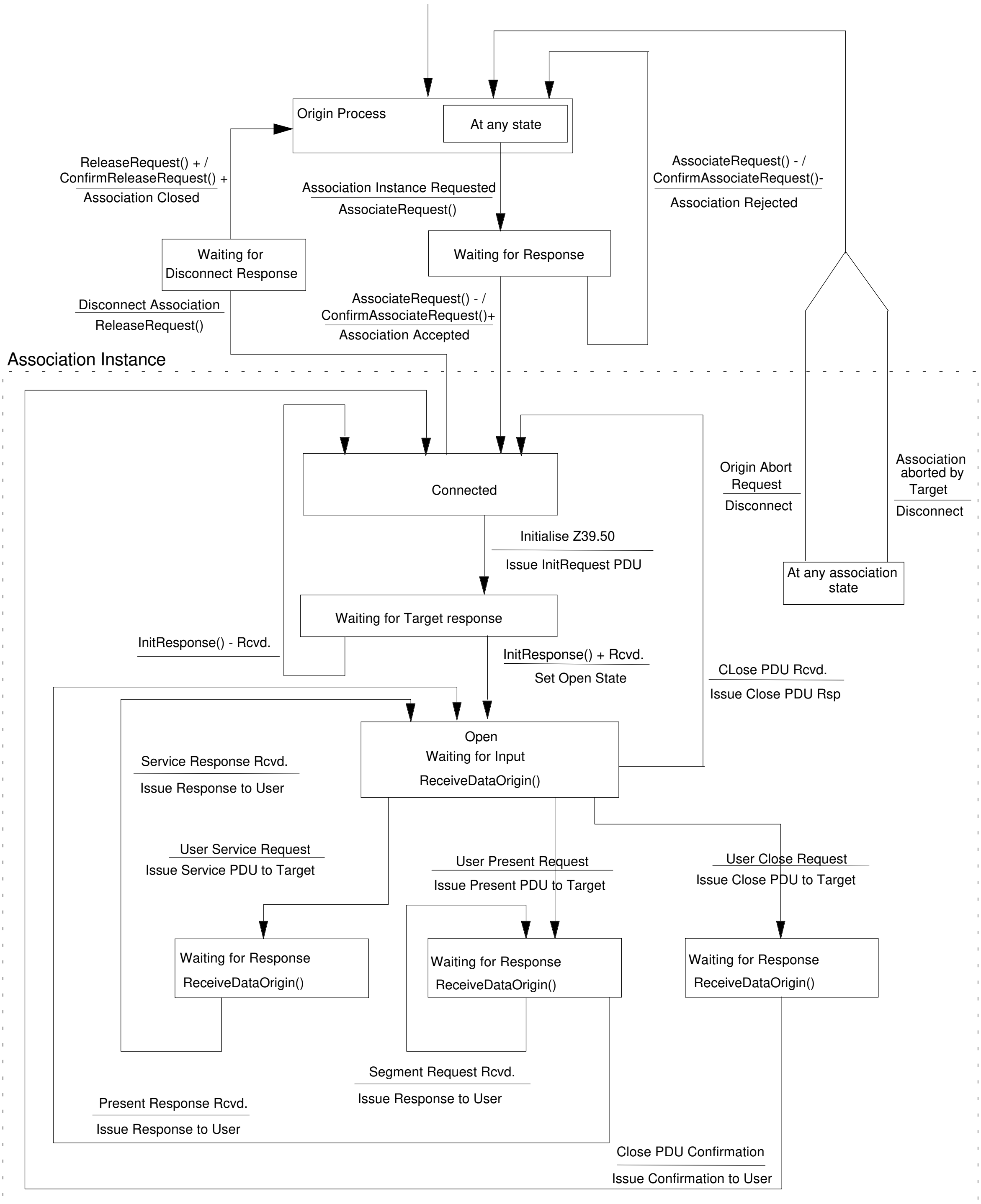


Figure 4.4 Detailed State Transition of Origin Process

4.3.2 Target Application State Machine

This section discusses particular aspects of the Target State Machine. In coding the state machine program, each particular event, input and state are given specific names in the Z39.50 Specification. These can be seen in Section 5.2.3 of the Z39.50 specification. It should be noted however, that all the events and actions are abbreviated and ideally the application program should adopt a similar naming convention. What is not apparent from the Specification, is the relationship of each event and action with respect to either Target, Origin, input or output.

In order to assist the developer in these matters, some suggested names are included below, together with the relationship and the direction of the data.

Note that both TCP/IP and OSI Target programs must conform to the same state machine behaviour, and therefore their construction and design may be very similar.

The behaviour of the Target State machine should conform to the description in the ANSI Z39.50 specification. This is an overall state machine comprising service-provider and service-user elements. In terms of state transition, and in terms of the state machine being built into an actual application, the following two diagrams depict an overall program state transition and a first level decomposition.

In the following diagrams, the notation used is the same as for the Origin Transition diagrams above.

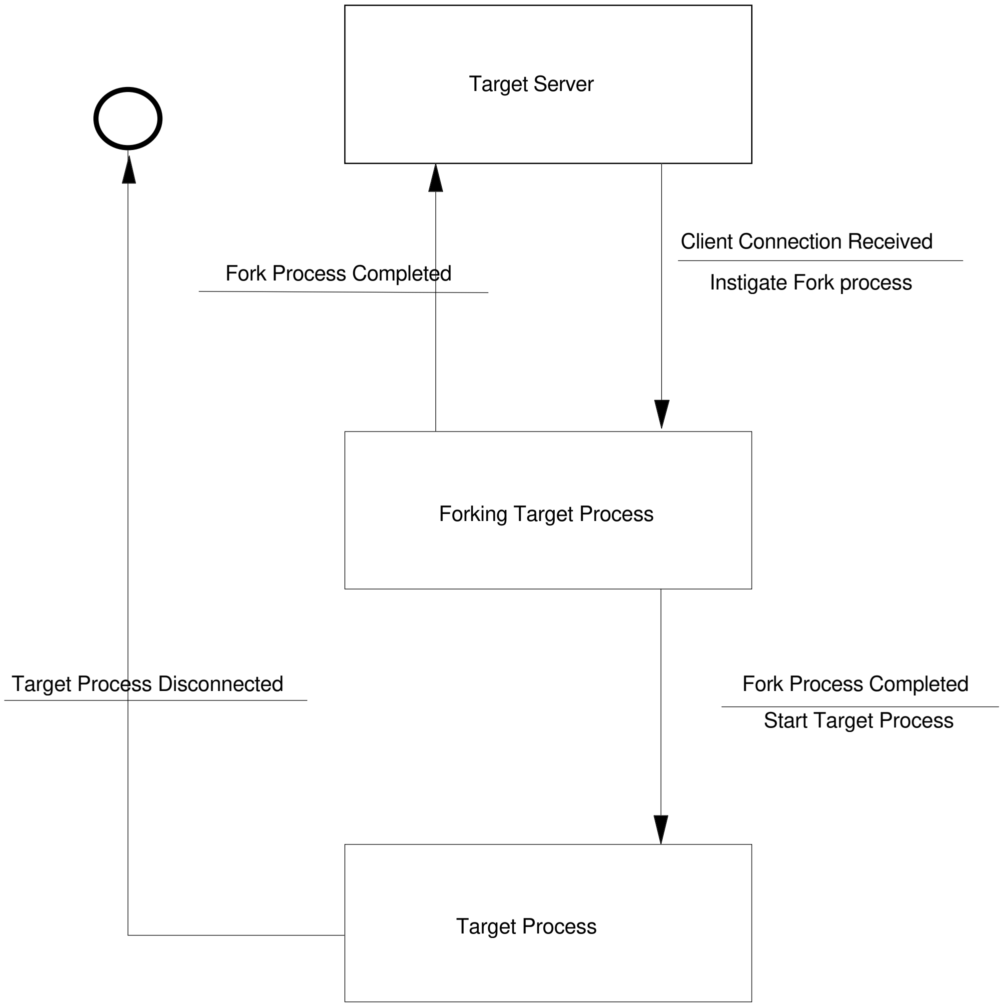


Figure 4.5 High Level State Transition of Target Process

In the high level state transition diagram of Figure 4.5 above, when the Target daemon receives an Origin connection, it will cause a state transition to spawn the Target Application program. Note that the spawning process will return state immediately to the Target daemon, and that when the Target Application process is complete it has no return state since the program will terminate.

Figure 4.6 below shows the state transition behaviour of the Target Application process itself. This diagram should be read in conjunction with the state behaviour in the Z39.50 Specification.

Note that the state transition information caters for both service-provider and service-user components.

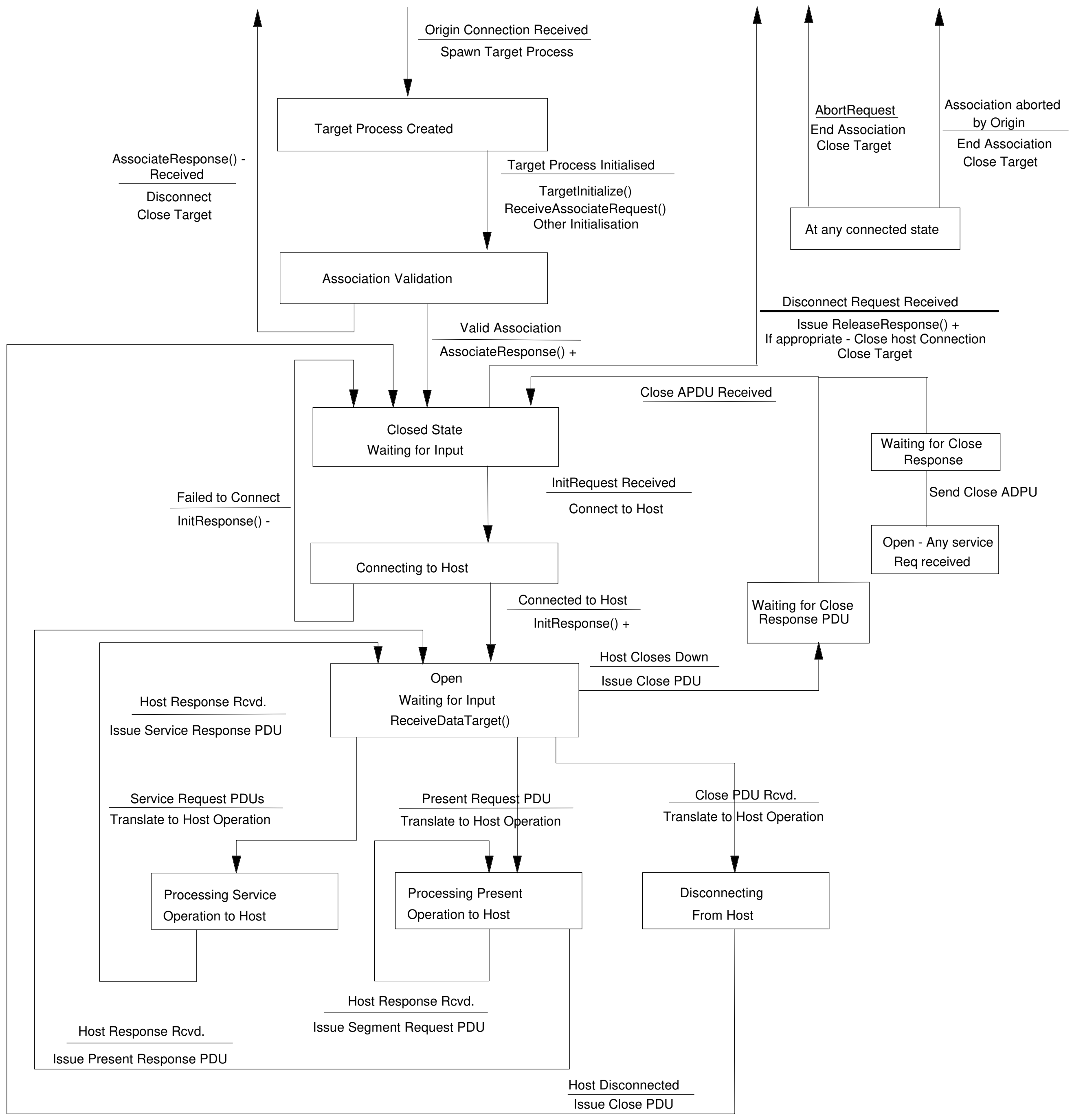


Figure 4.6 Detailed State Transition of Target Process

4.3.3 Implementation of the State Machine in Applications

Some guide-lines to the implementation of the Application level within the Origin and Target service-user components are given below:

The program may follow the following operations, namely:

a) waiting for an event (either from Application or network);

b) identification of the received event in terms of state machine input; and

c) validation of the event through the state machine.

When the incoming event signifies a Z39.50 Operation, the Operation may be given a reference identifier and the appropriate O-machine is started. All subsequent input events, whether from the Application or the remote Origin or Target will be identified and will be given to either the Z-machine or the referenced O-machine. In this context, the Z-machine and O-machine are running in parallel.

In practice, the implementation in the C language can be satisfied by several methods such as multiple and nested switch statements. However, one mandatory aspect of the program design is its ability to accept asynchronous inputs from both the application or the network. This particularly applies to the Z39.50 Close service, which can be initiated from either the application or the network.

4.4 Using The API

To the application programmer, the DBV-OSI II API consists of the following components:

1) A shared library containing DBV-OSI II API functions common to the Origin and Target. This library will be called : **libdbv**.

2) Two shared libraries, one for Origin specific functions, another for Target specific functions. These libraries will be called: **libdbvorigin** and **libdbvtarget** respectively.

3) A static library provided by snacc containing encoding and decoding functions called **libasn1csbuf.a**. The Origin and Target program developer need not be concerned about this library since the functions within it are called only by the API. The user only needs to ensure that this library is included in the program linkage.

4) Isode Communication stack libraries for developers implementing on OSI stack.

5) The following header files:

sidefs.h	- DBV OSI II standard definitions
sitypes.h	- DBV OSI II APDU structure type definitions
sitypes2.h	- DBV OSI II additional structure definitions
itemordr.h	- DBV OSI II item order extended service and ILL structure definition
iord_ext.h	- DBV OSI II SUBITO additional information structure definition
dbvproto.h	- prototypes of the API functions and definitions of errors.

Notes:

a) In order to include the API library functions in Origin and Target processes, these header files should be included in the Origin and Target program source files, and the shared library **libdbv** should be included in the linker command line with just the reference:

-ldbv and either **-ldbvorigin** or **-ldbvtarget** and also **-lasn1csbuf**

(list ISODE libraries also)

b) Note: The file extensions for shared libraries differ depending on the implementation of Unix being used. On SunOs 4 the extension is **.so.M.m** where M.m is the major version and minor version of the library respectively. Solaris 2 does not include the minor version while HP-UX has no version information in the filename, only a **.sl** extension.

c) The DBV OSI shared libraries outlined above will contain both OSI and TCP/IP association functionality. This allows origins to connect to both TCP/IP and OSI targets concurrently and for the same Target application to be used for both communication types. The type of communication stack to be used by the Target is passed in as a parameter to the application itself. The input parameters are then parsed by the `TargetInitialize()` function.

An example of including the header files is shown in the following C code:

If a Target process consists of one C file, target.c, which has a state machine containing calls to DBV-OSI II functions, it must have the following structure

```
/*Top of C Code File*/
#include "sidefs.h"
#include "sitypes.h"
#include "sitypes2.h"
#include "itemordr.h"
#include "siord_ext.h"
#include "dbvproto.h"

/* code to implement a target application which
 * includes calls to DBV-OSI II functions>
 */
{
.....
}
```

In order to link with the API library, the file could be compiled with the GNU gcc compiler like:

% gcc target.c -ldbv -ldbvtarget -lasn1csbuf

Or:

% gcc origin.c -ldbv -ldbvorigin -lasn1csbuf

4.5 Origin and Target Program Operations - Implementation hints

The following subsections discuss practical approaches to some important aspects of the Origin and Target operations and are implementation hints only.

4.5.1 Query Language Conversion

One of the most complex operations of the Origin program is the conversion of command line queries to the structure representations used within Z39.50. In addition, the developer will have to determine the exact mapping and behaviour of the query command language to the appropriate Z39.50 APDU and to the necessary elements within the APDU structure.

In the DBV-OSI II programs, this will be a two way mapping. The Origin component must map the local query language to Z39.50 requests, and also map the Z39.50 responses to the responses analogous to the query language. The more accurate this can be done, the greater the transparency of a remote host will appear to a user.

The Target component also has a translation aspect. It must map incoming Z39.50 APDU requests to its local host query language, and translate the local host responses to Z39.50 response APDUs. Some potentially difficult aspects are:

a) Some commands of the query language may not have an appropriate Z39.50 APDU to convert to - a compromise solution may need to be considered.

b) It should be assumed that the remote host will have a different query language - in this case some commands cannot be suitably passed to the remote host, and the Origin program may need to emulate these.

c) Some query languages have short-cut representations of the query language syntax - a special query expansion pre-processing operation may be needed before appropriate conversion to Z39.50 is possible.

d) Certain query language operations will require a degree of user parameter information to be retained and used by the Origin program.

An example of a) above would be for a command to display set information. There is no Z39.50 APDU to request information on the users current search history.

An example of b) above is if the remote host supports a database service that is not supported by the local host.

An example of c) above could be a range operator, such as a date range. IN this case, the query translation has to expand the date range to be date < year1 AND date > year2.

An example of d) above would be the utilisation of a "limitall" command, that, (say) restricted all future queries to a particular attribute. The Origin program would need to modify subsequent user queries appropriately.

The format of the query in the Z39.50 Search APDU is by far the most complex to deal with. The query and attributes are represented in a complex structure containing query terms and attributes. This structure, a form of reverse polish notation (RPN) is discussed below.

4.5.1.1 RPN Query Formation

The Z39.50 Type-1 and Type-101 queries use the Reverse Polish Notation of a query, RPN. It will be necessary for Origin developers to map the user defined query to RPN.

The Origin test program supplied with the API software contains a simple RPN generator software.

For example, the following command uses Boolean and positional operators:

SELECT ARTIFICIAL(W)INTELLIGENCE AND COMPUTER(W)APPLICATIONS

This command maps onto the following Z39.50 RPN structure. The direct mapping of elements to RPN structure fields are not shown.

QUERY101	- Z39.50 query type
1.2.840.10003.3.1	- OID
1,0 2,3 3,0 4,2 5,100 6,0 ARTIFICIAL	- term and bib-1 attributes for word "artificial"
1,0 2,3 3,0 4,2 5,100 6,0 INTELLIGENCE	- term and bib-1 attributes for word "intelligence"
PROX EMPTY 1 TRUE 2 0 2	- proximity relationship between "artificial" and "intelligence"
1,0 2,3 3,0 4,2 5,100 6,0 COMPUTER	- term and bib-1 attributes for word "computer"
1,0 2,3 3,0 4,2 5,100 6,0 APPLICATIONS	- term and bib-1 attributes for word "applications"
PROX EMPTY 1 TRUE 2 0 2	- proximity relationship between "computer" and "applications"
AND	- AND relationship between words.

4.5.1.2 Query Attributes

A number of query attributes sets are referenced in Z39.50 (e.g. Bib-1). However there is no specified standard attribute set or combination of attributes to be used for search requests. If a

Target doesn't support an attribute list passed in a search request the search must be rejected. To minimise this problem, some freely available Z39.50 Origin packages (e.g. Willow) have a huge range of selectable predefined attribute combinations. This allows an Origin to work successfully with different Z39.50 Targets. In addition, some Origins often allow attribute combinations to be determined by a user for a search. This is because the sets used by some Targets differ due to the interpretation of the Z39.50 specifications by the implementors.

4.5.2 Asynchronous Behaviour

The Origin and Target programs allow requests to arrive in any order at any time from both the local host and the TCP/IP or OSI daemon connected.

Within TCP/IP, this asynchronous behaviour can be achieved by using synchronous I/O multiplexing - using the programming level association identifiers of both the local host and TCP/OSI Target process associations within the select() system call. This call can be made to wait until data appears on any of the associations specified, either indefinitely, for a specified period or for a zero length period (in which case the select() acts as a poll).

The Z39.50 State Machine explicitly defines asynchronous behaviour by being able to cater for inputs from either the network or the host application.

4.5.3 Program/Process Errors And Recovery Procedures

The DBV-OSI II programs developed should have robust error checking and incorporate error recovery procedures as discussed below.

4.5.3.1 Signal Trapping

The occurrence of an unrecoverable error in the proposed master-slave architecture of the Targets be extremely rare.

It is recommended that unrecoverable errors in the Application processes should be trapped, and should produce a relevant diagnostic message followed by an orderly exit of the Application process. A "reaper" function could be added to automatically terminate such orderly terminated processes. This will ensure that no defunct processes, called zombies, are left.

Unrecoverable errors can be trapped and reaper functions can be specified using the UNIX "signal()" function call. Essentially, this function allows a 'signal' and an associated function to be specified. If the specified signal should occur, the specified function is executed.

Signals are specified using the following syntax:

signal(<signal to trap>, <function to execute when the signal occurs>)

An example of a program fragment which could be used to catch segmentation faults and bus errors is given below. In this case, the function program_terminate() will be called when such an error occurs.

```
#include <signal.h>

void program_terminate_segment();
void program_terminate_bus();

main()
{

/* When a segmentation fault occurs jump to function  program_terminate_segment
*/
      (void *) signal(SIGSEGV, program_terminate_segment);

/* When a bus error occurs jump to function program_terminate_bus
 */
      (void *) signal(SIGBUS, program_terminate_bus);

<rest of main>
}

void program_terminate_segment()
{
      printf("Program died because of segmentation fault\n");
      exit(1);
}

void program_terminate_bus()
{
      printf("Program died because of bus error\n");
      exit(1);
}
```

Catching and acting upon reaper signals is more complex than catching unrecoverable program faults since the signal function must return. A reaper signal should be specified in Target daemon code. When this is set up; if an Application process created by a daemon should terminate (ideally through the signal trap, exit(1) sequence specified above), the reaper function will be automatically called. This will terminate the exited Application process in an orderly fashion if needed; and will then return to the location from which it was called. If a system call is interrupted due to a reaper signal, on return from the reaper function, the interrupted system call returns -1 and sets the global error variable errno to EINTR.

Consequently, if using the master-slave architecture, every master system call must check for a -1 return and a setting of errno.

The structure of a Target daemon which uses a reaper function could be the following:

```
#include <signal.h>
#include <sys/errno.h>
extern int errno;
void reaper();
main()
{
      (void *) signal(SIGCHLD, reaper);
      <code which accepts connections from an origin and spawns the application>
}

void reaper()
{
      int wpid, status;

      while ((wpid = wait(&ststus)) != -1)
            printf("Reaped and killed process %d\n", wpid);
}
```

Note: it should be checked to see if the this mechanism is supported on your UNIX system. It is known to work on Sun Solaris 1 (BSD), Sun Solaris 2 (System V) and IBM RS6000 AIX. Some published reaper processes such as:

```
#include <sys/wait.h>
union wait status;
while (wait3(&status, WNOHANG, (struct rusage *) 0) >0)
      continue;
```

are BSD specific and are not portable to system V, since System V does not support wait3().

4.5.3.2 Communication Errors

If a DBV-OSI II API communication function returns an error indication, appropriate action must be taken. The following could be done:

a) issue an error message to the Administrator; and,

b) perform an orderly exit.

Checking communication errors is particularly important for Target association functions if a master-slave architecture is used. It will most likely be the case that the association acceptance and Application creation functions will be in an infinite loop. If connection errors are ignored, it could be possible for the program to enter a situation whereby instances of the Application program are being created continuously; thereby filling the UNIX system process table.

4.6 Determining Stacks to Use

Both TCP/IP and OSI stacks will be supported by the API. Both requires some system requirements in order to be supported. By convention, the UNIX platform will readily support TCP/IP communications. OSI communications requires the use of additional software products, namely the protocol stack software, and the X.25 interface software.

A diagrammatic representation of the stacks is given in Figure 4.7 below.

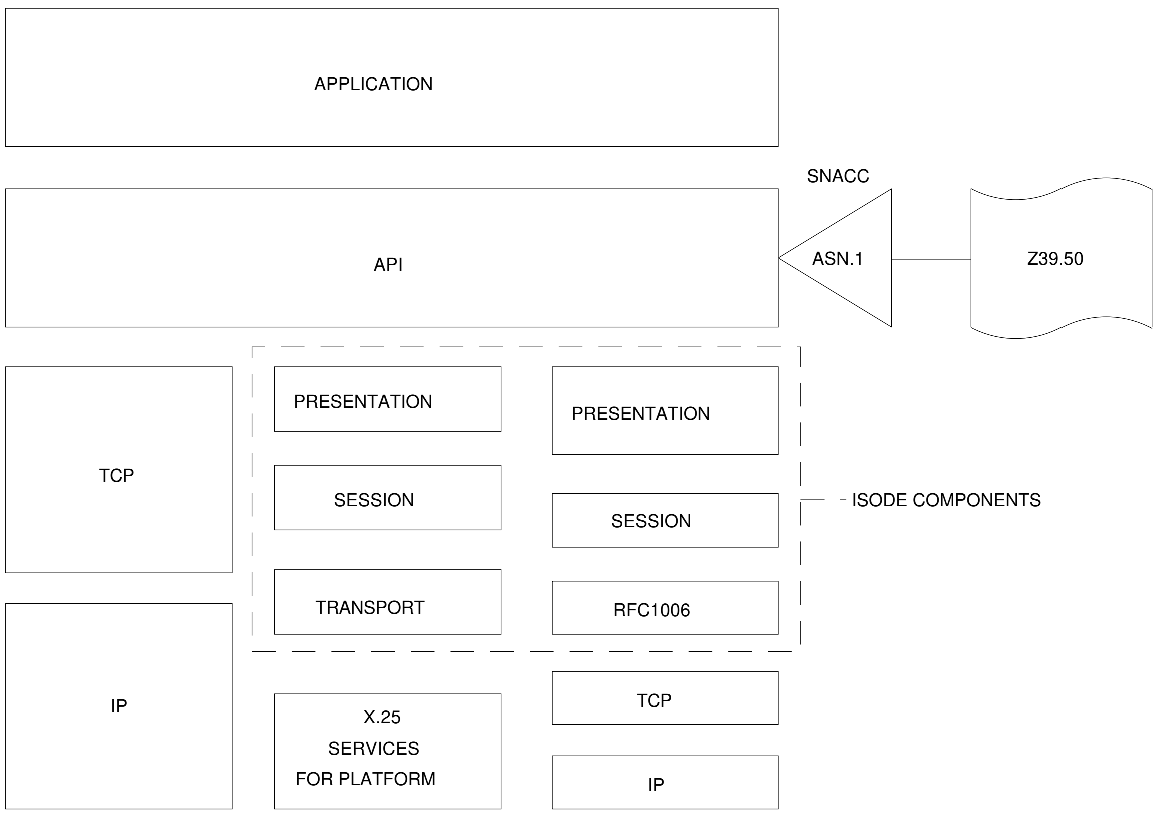


Figure 4.7 Representation of Communications Stack Components

4.6.1 TCP/IP Stack

All vendor platforms to be ported to support BSD 4.x Socket TCP/IP communications software. This will be the basis upon which TCP/IP communications software will be developed. System functions must be correctly used in order for the State machine programs to correctly operate in the asynchronous behaviour.

4.6.2 OSI Stack

The OSI stack will utilise the ISODE package. The ISO Development Environment is a collection of library routines and programs that implements an extensive set of OSI upper-layer services. The ISODE implementation of the upper-layers of OSI is interesting in four respects:

a) it provides extensive automatic tools for the development of OSI applications;

b) it supports OSI applications on top of both OSI and TCP/IP-based networks;

c) it provides a novel approach to the problems of OSI coexistence with and transition from the Internet suite of protocols; and,

d) it is openly available (non-proprietary), although the Consortium version will be used in this application.

ISODE contains four parts:

a) a set of application service elements;

b) a collection of ASN.1 tools;

c) presentation and session services; and,

d) interfaces to transport and network layer realisations.

4.6.2.1 Building Distributed Applications

Although ISODE can be programmed at several levels, it provides a set of semi-automatic tools, termed The Applications Cookbook which can be used to construct OSI applications rapidly and easily.

Whilst the ISODE package contains an ASN.1 compiler to produce C functions that link to the OSI stack libraries, the DBV-OSI II Application will utilise the snacc ASN.1 compiler instead of the ISODE ASN.1 compiler. This is because the snacc compiler produces far more efficient encoding and decoding routines.

4.6.2.2 Transport-Independence

When the ISODE was originally developed in January of 1986, there were very few networks and systems which supported the OSI transport service. In contrast, the Internet suite of protocols, and in particular, its reliable transport protocol, the TCP, were widely deployed even at that time (and are even more so today).

To solve this problem, a transport service convergence protocol (TSCP) between the TCP and the OSI transport service was developed, known as the RFC1006 method.

This particular TSCP is a simple protocol that runs over the TCP and makes the service it offers appear to be identical to the OSI transport service. This is an important abstraction in that it allowed the development of native OSI applications, that behaved as if they were running in a pure OSI environment. Later, when some pure OSI environments became available, the same applications can run ran without even being re-compiled: they need only be reloaded with the new transport library.

In support of this, there is an important abstraction in the ISODE called the transport switch. When the ISODE is configured, one or more transport-stacks are defined. A transport-stack is simply a combination of protocols which offer the OSI transport service.

5. Program Development

This section of the document recommends approaches for developing applications using the DBV-OSI II API.

5.1 Development Tools

High quality, and highly portable freeware tools provided by the GNU project in America. are recommended for program development. It is recommended that the following GNU tools which are freely available should be obtained and installed on your UNIX system. Full installation instructions are provided with each tool.

1) gzip-1.2.4 (or higher) - Efficient file compression/ decompression utility
2) gcc-2.5.7 (or higher) - ANSI C compiler
3) gdb-4.11 (or higher) - Source code debugger
4) make-3.70 (or higher) - Make utility

Note that not mentioned above is glibc, a GNU replacement for standard C libraries and header files. Although gcc exists for many machines, glibc, which is still at version 1 only exists for a few at the moment. The recommended approach is to use gcc with the proprietary libraries and files provided by the machine manufacturer. This is the default operation of gcc.

5.1.1 ANSI-C C Source Code Compilation - Using gcc

gcc is a popular fully ANSI-C compliant compiler. Unlike some proprietary ANSI C compilers (such as Sun's acc) this compiler supports the entire ANSI standard. The compiler provides a wide range of configuration options and supports multiple user concurrent compilation. In its most basic form, gcc is used as follows:

% gcc < filename>

e.g.

% gcc myfile.c

Provided it has been installed, full details of gcc can be found by entering

% man gcc

5.1.2 Debugging - Using gdb

gdb is a popular source code debugger which supports a wide range of source languages including ANSI-C.

It is similar to the dbx debugger found on many UNIX environments but is superior and has more useful facilities. In addition to standard debugging facilities such as breakpoint setting, line stepping, variable printing and identification of source code lines which cause segmentation faults or bus errors, it has several sophisticated features attachment to including running processes .

There are three usual scenarios for using gdb, described below.

1) To locate the line in a program that causes a core dump.

Example Sequence:

Initially, recompile the source file(s) with the "-g" compiler option. This will add symbolic information to the compiled file. Then, re-run the program and repeat the sequence of events which causes the program crash. Finally, invoke gdb and display the source file line which causes the core dump.

% gcc -g myfile.c

% a.out

% gdb a.out core

gdb> where

2) To execute a program within the gdb environment.

This option will allow full control over the program execution. gdb will catch segmentation faults and bus errors, allow breakpoints to be set, variables to be examined etc.

Example Sequence - compile the source code files with the "-g" option to create the gdb symbolic information. Invoke gdb and then run the program within gdb. Full information about gdb is available in the on-line manual. A X-Interface to gdb is also available in the utility xxgdb.

% gcc -g myfile.c

% a.out

% gdb a.out

% run

3) To attach gdb to a running process.

This provides control to gdb to the running program. This includes catching segmentation faults and bus errors, breakpoint setting, variables to be examined etc.

Example Sequence:

% gdb program 01234

where 01234 is the process id to attach to.

The latter example, is particularly useful in an X-Window based development environment when debugging master-slave architectures described in previous sections. It is recommended that each Application is executed in its own x-term and that a gdb debugger process (which will run in its own x-term) is automatically attached. When the system is operational, the code that invokes the gdb debugger and the xterm environments can be removed.

The following C code fragment shows how a gdb debugger can be specified at compile time to automatically attach at run-time to a running process. The process assumes that the code fragment is compiled into an executable called

targetslave

```
char gdb_str[100];

dummy()
{ }

main()
{
      printf("Executing: xterm -e gdb targetslave %d\n", getpid());
      printf("When the window appears, do:   b dummy\n");
      printf("                            c\n");
      printf("Sleeping 20 seconds.....\n");
      sprintf(gdb_str, "%s%d%s", "xterm -e gdb targetslave ", getpid(), " &");
      system(gdb_str);
      sleep(20);
      dummy();

      < rest of target application process>
}
```

The behaviour of this code is as follows:

It is assumed that the executing program is displaying to an x-term window. The printf statements will notify that the gdb process is about to be invoked, and also instructs what to do when the gdb window appears. The sprintf statement generates a command line to invoke another x-term window, and to initiate the gdb command. The process id for gdb to attach to is this process, as retrieved by the getpid() function.

The system() function invokes the command, and then this process will sleep, allowing for the gdb window and process to appear. When this window does appear, the instructions in the printf statements must now be executed. (remember, you have 20 seconds to do this !!). The "b" command instructs gdb to put a breakpoint at the function dummy(). The "c" command instructs gdb to continue with execution of the program. Now, the program being debugged will pass after the sleep() function and call the function dummy(). Here it will stop since there is a breakpoint. The user now has control over the program through gdb to debug the program.

5.1.3 Use of Makefiles

All source code compilation should use a 'makefile' utility. The purpose of this utility is to determine automatically which source files or other components of a program need to be recompiled and automatically issue the commands to recompile them. Using this utility is essential when developing a large program consisting of multiple source files as it always produces the most time-efficient compilation; if one file from a source set of (say)fifty files is modified only this one file is recompiled and relinked.

It is recommended that the GNU makefile utility (version 3.70 or higher) be used rather than a manufacturer's makefile utility.

There are two common approaches for makefile creation and usage, namely:

1) Raw makefile creation and usage:

In this approach, the following are required:

a) Create a file called Makefile (or makefile) in an editor specifying all source files and build rules.
b) Invoke the makefile utility by entering the command:

% make

2) 'Automatic' makefile creation and usage:

In this approach, the following are required:

a) Create a file called Imakefile specifying source files (An Imakefile must have a standard format)

b) Automatically create a makefile called makefile which specifies source files and build rules from the specification in Imakefile by entering the command:

% xmkmf

(xmkmf is a utility provided as part of X11 system software)

c) Invoke the makefile utility by entering the command:

make

Imakefiles are standard format files which were created to simplify the creation of makefiles. Although popular, they are not always the automatic choice, especially when only a small number of source files are to be compiled, as they are less flexible than raw makefiles and tend to produce large makefiles as output.

An extremely simple raw makefile is reproduced below which builds an executable program called shapes from five sources, circle.c, square.c, rectangle.c, oval.c and arc.c.

```
#----------------------------------------------------------------
# Example makefile
#
# Usage:
#
# make
#----------------------------------------------------------------
#
#                 Program name
#
PROGRAM= shapes
#----------------------------------------------------------------
#       Directory for object files

OBJDIR= object_files

#----------------------------------------------------------------
#       Specify Compiler

CC = gcc

#----------------------------------------------------------------
#     C source files

# Program (shapes) gcc ANSI C source files
CSRCS= circle.c \
      square.c \
      rectangle.c \
      oval.c \
      arc.c

#----------------------------------------------------------------
#     Define object files

MEMB= $(CSRCS:.c=.o)
OBJS= $(MEMB:%=$(OBJDIR)/%)

#----------------------------------------------------------------
#     Principal dependencies

all:  $(PROGRAM)

$(PROGRAM): $(OBJS)
      $(CC) -o $(PROGRAM) $(OBJS)
      chmod a+x $(PROGRAM)

#----------------------------------------------------------------
#     Implicit build rules

$(OBJDIR)/%.o: %.c
      $(CC) -c -o $@ $<
```

Full details of the default supported make utility can be found by entering the command:

% man make

5.2 Memory Verification And Diagnostic Utilities

In the development of UNIX system software, many program errors can be due to incorrect usage of memory related system functions, such as malloc() and free().

Some platforms, (such as Sun Solaris-1 provide additional memory verification routines which are linked with the program. These routines, from system file /usr/lib/debug/malloc.o, include memory checking and diagnostic replacements for memory functions such as malloc() and free() which automatically detect memory corruption errors which can be overlooked.

In addition, a facility called checker is available on the LINUX platform.

On Sun Solaris-1, details about the /usr/lib/debug/malloc.c memory routines can be obtained by entering the command:

man malloc_debug

5.3 Source File Version Control

It is recommended that any source files developed are maintained using a version control system.

SCCS, which is an abbreviation for 'Source Code Control System' is a version control tool available in most UNIX environments which allows files to be securely maintained, modified and updated. Any file including binaries can be held under SCCS.

The steps involved in setting up an SCCS system are listed below. All steps should be performed from the source directory containing the files to control. For a set of files, setting up SCCS is only required once as the setting remains 'permanent' - when the system is next used; hours, days, possibly months later and after any number of logins, the SCCS system is immediately ready to respond to SCCS commands.

1) Create a directory called SCCS (note the use of upper case) in the directory containing sources which are to be controlled.

NOTE: LEARNING USERS SHOULD NEVER ENTER AN SCCS DIRECTORY OR MODIFY ANY FILES CONTAINED WITHIN.

2) 'Put' each file required to be maintained by SCCS under the tool's control using the command

% sccs create <filename>

Once an SCCS directory has been set-up and files entered under control of the tool, the files may be 'retrieved' as required for further editing, compilation and testing. Once retrieved from SCCS, files may be put back under SCCS control at any time, normally this is only done after a useful change (such as correcting an error).

When a file is initially put under SCCS control and each time it is 'put back' after being edited, a 'new version' of the file is created which is assigned a unique version number. Version numbers start at 1.1 - this is assigned to the version of the file which is initially put under SCCS control - and incremented logically. The use of version numbers allows great control as any version of any file may be retrieved and used. Baselineing of versions is possible which is extremely useful when producing releases of software (e.g. the current version of all source files in a directory can be made to be version 3.1 etc.)

By default after all operations, the most recent version of a file is left in the directory from which an SCCS command is issued.

5.3.1 Basic SCCS Commands

The basic commands required after setting up source files in an SCCS directory are given below. All commands should be issued from the directory containing the SCCS directory.

% sccs edit <filename>

Retrieve a file from SCCS control for editing (the file will have both read and write permission).

% sccs get <filename>

Retrieve a file from SCCS control for reading only (the file will have read permission only).

% sccs delget <filename>

Return a file (possibly modified) to SCCS control. SCCS will automatically assign a new version number to the file. When a this command is entered, the SCCS system will issue a:

Comments?

prompt, allowing entry of comments which describe changes made to the file. Any relevant comments should be entered. After the comments are accepted, a new version of the file will be created under SCCS control and assigned a new version number. A copy of the file version will be left in the directory with read-only file protection attributes.

% sccs prt <filename>

Print the SCCS 'history' of a file including version numbers and comments.

% sccs unedit <filename>

If a file is out of SCCS for editing, this command will discard the file and retrieve the previous version. NOTE: any edits to the file whilst it was out for editing will be lost.

% sccs info

Display a list of files being edited plus who is editing them and what versions are being edited.

5.3.2 Typical Basic SCCS Command Sequence

A typical SCCS command sequence is given below. The sequence will create an SCCS directory, put a file under SCCS control, retrieve it from SCCS for editing then put it back under SCCS control.

Assuming there is a single file hello.c in directory /home/hough/code:

% cd /home/hough/code

% mkdir SCCS

% sccs create hello.c

% sccs edit hello.c

Edit hello.c in an editor, add lines, compile and test until working.

% sccs delget hello.c

Add any comments at the Comments? prompt issued by the delget command.

The following day....

% cd /home/hough/code

% sccs edit hello.c

Add lines to hello.c, compile and test until working then...

% sccs delget hello.c

% sccs prt hello.c

Full details of SCCS can be found on any UNIX environment which supports it by entering the command:

% man sccs

5.4 Configuration Management

A logical structure for source, object and executable files is recommended. In most software projects, The following directory structure is suitable for this:

```
<root directory of product>
|
|---documentation
|---exedir
|---cshellscripts
|---src

src
|
|--- src directory 1
|--- src directory 2
```

The contents of the exedir directory would typically be:

1) Executables and files required at run time built and copied from src directories.

The contents of the cshellscripts directory would typically be:

1) shell scripts used during development, (such as porting scripts, source code baseline scripts etc.).

2) SCCS directory for version control of the shell scripts

The contents of the src directory would typically be:

1) Sub directories

src directory 1 contents:

1) Source files used to make a distinct component of the project.

2) Makefile to build the project component into exedir

3) SCCS directory for version control of the source files and Makefile.

src directory 2 contents:

1) Source files used to make a distinct component of the project.

2) Makefile to build the project component into directory exedir

3) SCCS directory for version control of the source files and Makefile.

6. Additional Programs And Development Aids

6.1 Test Programs

In the course of developing the DBV-OSI II API test programs for testing the Origin and Target API functions have been developed and are included in the software package. The programs will illustrate typical programmer use of the API including connection and query to known V2 Z39.50 Targets.

6.2 API Test Tool

A comprehensive test tool is included with the API package which has been used to prove and verify the API functions.

The test tool will is written in GNU ANSI C which allows portability across hardware platforms It will use simple virtual terminal based text menus.

6.2.1 Test Tool Modes

The Test tool will provide the following two applcations:

1) Origin testing application to allow testing of Origin related API functions with both valid and invalid parameter combinations,

2) Target testing application to allow testing of Target related API functions with both valid and invalid parameter combinations.

3) Menu and Batch modes exist whereby menu mode requres that each action of the test tool is manually operated and batch mode allows a control file to determine the set of actions that the test tool will follow.

6.2.2 Test Tool Control And Result Analysis

To enable analysis of parameters passed to API functions and resultant data transfer from the use of such parameters, text files will be used to control all testing. Parameters for function testing will be read from relevant text files and data received by both Origin and Target functions will be written to relevant files as text strings. There will be up to five types of diagnostic files associated with the execution of each API function.

1) 'Input From' files - these files will contain parameters to be passed to API functions when they are executed.

2 'Result To' files - these file will contain information about the success/failure of the function execution.

For Origin API-APDU functions only:

3) 'Request At Target To' files - these files will be created by the Target upon reception of a APDU from the Origin and will contain the APDU data and other related information.

4) 'Target Response From' files - these files will hold API function parameters to be used when creating a response to send back to the Origin.

5) 'Response From Target To' files - these files will be created by the Origin upon reception of a response from the Target.

For Target API-APDU functions only:

3) 'Request At Origin To' files - these files will be created by the Origin upon reception of an APDU from the Target and will contain the APDU data and other related information.

4) 'Origin Response From' files - these files will hold API function parameters to be used when creating a request to send back to the Target.

5) 'Response From Origin To' files - these files will be created by the Target upon reception of a response from the Origin.

It will be assumed that all input files will always contain correctly formatted fields. All files will use a flexible table based format which has been previously developed at Crossnet systems Ltd. The TBL table file (.tbl) format has been used successfully in several commercial projects and a large range of ANSI C support functions exist to control creation, update and maintenance of such files. Each line entry in a file having this table based format has the following structure.

<Key field><variable length list of mixed format data items>

6.2.3 Test Tool Operation

The test Tool has its own documentation for describing the operation of the menu and batch modes.

Appendix A - Colbert Methodology

The Colbert methodology is a very practical approach to object-oriented analysis and design. The method concentrates on objects (or instances) initially. Classes are very much a secondary consideration to ensure that the commonality between objects is identified and duplications are therefore eliminated.

An object is considered "active" if it displays independent motive power, otherwise it is considered to be "passive" . Passive objects act only under the motivation of active objects.

The method uses a consistent object model throughout the life-cycle stages, covering requirements analysis, preliminary design and detailed design.

Four activities can be performed to create the model of the application from the problem statement:

1) Object Interaction specification, (OIS)
2) Object Class specification, (OCS)
3) Behaviour specification, (BS)
4) Attribute specification. (AS)

These activities can be performed in any order. OIS identifies objects, their interactions and the hierarchical relationship of objects. The Object Interaction and Object Hierarchy diagrams are used.

OCS identifies the classes of objects and the relationships between the classes. The class relationships primarily depict component type relationships (is_part_of).

BS identifies the dynamic behaviour of an object. Typically state transition diagrams are used to depict the life states of an object.

AS identifies the quantitative and qualitative measures and resources for each object in the system. An attribute specification table enables the analyst to record quality criteria such as portability, performance and reusability criteria which must be associated with an object.

The objectives of the preliminary design phase are to refine the model developed during analysis and add sufficient rigour to create an implementable solution. It creates a language independent description of the software architecture of the system. The object interaction, object hierarchy, object class and state transition diagrams are all refined from those generated during the analysis phase.

During the detailed design, the implementation dependent representation of the software architecture is created. The same models and techniques are used. decisions can now be made on how to represent the objects identified in previous stages in a programming language.

Object interaction diagrams represent the interactions between objects. An interaction involves an operation and optional information data flows. The symbols for active, passive and external objects are used. External objects are considered as those outside the scope of the system to be implemented.

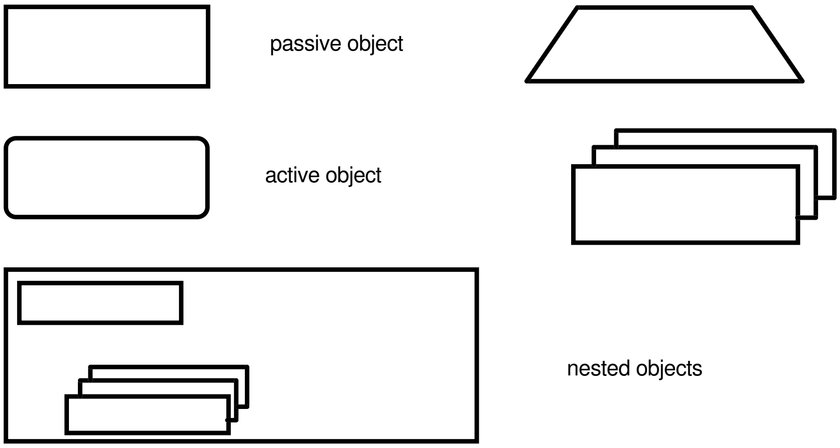
Object hierarchy diagrams represent the decomposition of a system into its components. In this sense, hierarchy is not the inheritance of object classes. Generally there is a top level diagram providing the system to be implemented in its context with all external objects. This is decomposed/exploded to a more detailed diagram which describes the whole system to be implemented. Each of these objects can then be decomposed to finer and finer levels of detailed objects.

Object class diagrams show relationships between classes and corresponding objects and information flows defined in object interaction diagrams. Class relationships to other classes are also shown.

The following diagram shows the syntax of the Colbert 91 methodology Object Interaction diagrams.

Colbert (OOSDM) Notation
Object Interaction Diagram (OID)

Objects



passive object

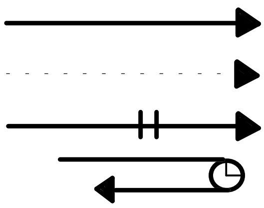
external object

active object

collection of objects

nested objects

Service Calls



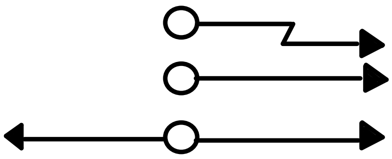
simple internal synchronous service calls (like function calls)

simple external synchronous service calls

asynchronous service calls (interrupts, mailbox, etc)

timed synchronous calls (handshakes, etc)

Data Flows

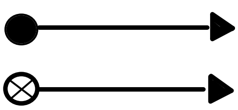


error flow

in/out flow

in and out flow

Object Flows



Grant Access

Revoke Access