Satellites International Limited

The Paddock
Hambridge Road
Newbury
Berkshire
United Kingdom

# DBV-OSI II

# Offene Kommunikation der Fachinformations- und Bibliothekssysteme

# SYSTEM

# EXTERNAL SPECIFICATION

468-SES-003

Issue 1

D-020

February 1995

| | | |
|---|---|---|
| Prepared by: | J. Hough | SIL |
| | R. Bull | SIL |
| | C. Attewell | SIL |
| Approved by: | | Project Manger |
| | R. A. Bull | |
| Authorised by: | | Technical Director |
| | A. K. Ward | |
| Accepted by: | B. Hergeth | Danet |
| | e-mail dated 30/1/95 | |

The content of this document is Commercial in Confidence and is the property of Satellites International Limited. The information contained herein must only be used for the purpose for which it is supplied and on request the document will be returned to Satellites International Limited together with any copies made.

**DOCUMENT STATUS SHEET**

| DOCUMENT STATUS SHEET | | | |
|---|---|---|---|
| 1. DOCUMENT TITLE: **DBV-OSI II System External Specification 468-SES-003 - D-020** | | | |
| 2. ISSUE | 3. REVISION | 4. DATE | 5. REASON FOR CHANGE |
| | 1 | 27 Oct 94 | Preparatory draft |
| Draft A | 1 | 9 Dec 94 | Preparatory Draft Comments |
| Draft B | 1 | 11 Jan 95 | Draft A Comments |
| Issue 1 | 1 | 10 Feb 95 | Acceptance + Comments on Draft B |

# Table of Contents

**1. Introduction ..... 5**
1.1 Abbreviations and Nomenclature ..... 5
1.2 References ..... 6

**2. The Responsibilities of Satellites International Limited and the Project Partners ... 7**
2.1 SIL Responsibilities ..... 7
2.2 Project Partner Responsibilities ..... 7

**3. System Architecture ..... 9**
3.1 Context Architecture ..... 9
3.2 Multiple Connections ..... 12
3.2.1 Origin Component Architecture ..... 13
3.2.2 TCP/IP Target Component Architecture ..... 16
3.2.3 OSI Target Component Architecture ..... 19
3.2.4 Different Communication Stacks ..... 20

**4 System Development Considerations ..... 22**
4.1 Origin And Target Component Analysis ..... 22
4.2 Service Provider State Machines for the Origin and Target ..... 23
4.2.1 Utility functions ..... 26
4.3 Origin And Target Application State Machines ..... 27
4.3.1 Origin State Machine ..... 27
4.3.2 Origin Application State Machine Behaviour ..... 33
4.3.3 Target Application State Machine ..... 36
4.3.4 Target State Machine Behaviour ..... 40
4.3.5 Implementation of the State machine ..... 43
4.4 Using The API ..... 44
4.5 Origin and Target Program Operations - Implementation hints ..... 45
4.5.1 Query Language Conversion ..... 45
4.5.1.1 RPN Structure Generator ..... 46
4.5.1.2 Query Attributes ..... 47
4.5.2 Locally Stored Information ..... 48
4.5.2.1 Query Attributes ..... 48
4.5.2.2 Connections From Target To Hosts ..... 49
4.5.2.3 Asynchronous Behaviour ..... 49
4.5.3 Program/Process Errors And Recovery Procedures ..... 49
4.5.3.1 Signal Trapping ..... 49
4.5.3.2 Communication Errors ..... 51
4.6 Determining Stacks to Use ..... 53
4.6.1 TCP/IP Stack ..... 54
4.6.2 OSI Stack ..... 54
4.6.2.1 Building Distributed Applications ..... 54
4.6.2.2 Transport-Independence ..... 55
4.7 The Administration Component ..... 55
4.7.1 Communications from the Origin and Target Applications to the Administration Component ..... 58

4.7.2 Administration Component Errors ..... 60

**5. Program Development ..... 61**
5.1 Development Tools ..... 61
5.1.1 ANSI-C C Source Code Compilation - Using gcc ..... 61
5.1.2 Debugging - Using gdb ..... 62
5.1.3 Use Of Makefiles ..... 64
5.2 Memory Verification And Diagnostic Utilities ..... 67
5.3 Source File Version Control ..... 67
5.3.1 Basic SCCS Commands ..... 68
5.3.2 Typical Basic SCCS Command Sequence ..... 69
5.4 Configuration Management ..... 70

**6. Additional Programs And Development Aids ..... 72**
6.1 Test Programs ..... 72
6.1.1 Prototype 1 ..... 72
6.1.2 Prototype 2 ..... 73
6.2 API Test Tools ..... 76
6.2.1 Test Tool Modes ..... 77
6.2.2 Test Tool Control And Result Analysis ..... 77
6.2.3 Test Tool Operation ..... 78
6.2.4 Origin Test Mode ..... 78
6.2.5 Origin Test Mode Test Sequence ..... 81
6.2.6 Target Test Mode ..... 82

**Appendix A - Colbert Methodology ..... 86**

# Table of Figures

**Figure 3.1 Context Diagram of DBV-OSI II ..... 9**

**Figure 3.2 High Level Decomposition of DBV-OSI II Application Object..... 10**

**Figure 3.3 Possible Client/Origin Architecture ..... 14**

**Figure 3.4 Decomposition of DBV-OSI II Origin Component ..... 14**

**Figure 3.5 Block Diagram of Target Architecture ..... 16**

**Figure 3.6 Decomposition of DBV-OSI II TCP/IP Target Component..... 17**

**Figure 3.7 Decomposition of DBV-OSI II OSI Target Component ..... 19**

**Figure 3.8 Block Diagram Showing use of Different Stacks by Servers..... 21**

**Figure 4.1 State Transition Diagram of Origin Service Provider..... 24**

**Figure 4.2 State Transition Diagram of Target Service Provider ..... 25**

**Figure 4.3 High Level State Transition of Origin Process..... 34**

**Figure 4.4 Detailed State Transition of Origin Process ..... 35**

**Figure 4.5 High Level State Transition of Target Process..... 41**

**Figure 4.6 Detailed State Transition of Target Process ..... 42**

**Figure 4.7 Representation of Communications Stack Components ..... 53**

**Figure 4.8 Representation of Communications Stack Components ..... 56**

# 1. Introduction

This document describes the modules being developed by Satellites International Ltd and their operation, the way they interface to the service user application and recommends an approach for developing DBV-OSI origin and target systems used by each Partner in the DBV-OSI consortium. It is assumed that the reader is familiar with the DBV-OSI II project and the series of D-0xx documents.

## 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 server 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. |
| SIL | Satellites International Limited. |
| 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 SIL. |
| 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. Colbert E, (1989), The Object Oriented Software Development Method: A Practical Approach to Object-Oriented Development, TRI-Ada 1989 ACM.

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

# 2. The Responsibilities of Satellites International Limited and the Project Partners

## 2.1 SIL Responsibilities

For the DBV project, Satellites International will be responsible for developing several software modules:

- **Origin Service Provider** - These are the API functions which provide association control and Z39.50 encoding/decoding/data transfer routines for the origin application. Their functionality is defined in the documents - DBV-OSI II API-Definition Dok.-Nr.: D-008 Section 3.1 and DBV-OSI II Software Internal Specification 486-SIS-004 D-022 Section 6.
- **Target Service Provider** - These are the API functions which provide association control and Z39.50 encoding/decoding/data transfer routines for the target application. Their functionality is defined in the documents - DBV-OSI II API-Definition Dok.-Nr.: D-008 Section 3.2 and DBV-OSI II Software Internal Specification 486-SIS-004 D-022 Section 7.
- **Utilitiy functions** - These are the API functions which provide various functionality useful to the service user programmers of both the origin and target applications. Their functionality is defined in the documents - DBV-OSI II API-Definition Dok.-Nr.: D-008 Section 3.3 and DBV-OSI II Software Internal Specification 486-SIS-004 D-022 Section 7.
- **Administration Tool** - This application is provided for the administration of the completed Z39.50 system. It controls access to and from the system by starting and stopping the master servers. It also stores information about the origin and target applications which are run. This is done via a UNIX FIFO. The administration tool reads information from the FIFO while the origin and target applications write status information to it. It is the job of the Service User to provide information to the FIFO although programming instructions to make this easier will be provided.
- **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 D-023 (SIL Number 486-TTM-005) for complete details.
- **Other Utilities** - These are further useful programs and utilities which will be provided by SIL.
  - A TCP/IP master server which can be used to start the origin and TCP/IP target applications.

## 2.2 Project Partner Responsibilities

The project Partners will then use the developed software to build the following modules:

- **Origin Application** - This application will be executed by a master server when a user wishes to access a remote database. It will include a connection to the local host (provided by the

service user), association handling routines for connection (or multiple connections) to remote databases (target applications) (provided by the service provider origin API), conversion routines to convert data received from the local host from the local query language to DBV Z39.50 structures (routines provided by service user, structures defined by service provider origin API. See DBV-OSI II API-Definition Dok.-Nr.: D-008 Section 4) and routines to encode the DBV Z39.50 structures to a binary stream and transfer them to a remote host (service provider origin API). For responses from host machines (target applications), routines must be provided which decode the incoming binary stream into the DBV Z39.50 structures (provided by the origin service provider API), convert the Z39.50 data structures into the local host query language and provide the results to the local host (both the duty of the service user).

- **Target Application** - This application will be executed by a master server application when an incoming request for a Z39.50 session is received. It 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 to the origin, routines must be provided to receive local host data, translate from the local host format into DBV Z39.50 structures (provided by the service user, structures defined by service provider origin API. See DBV-OSI II API-Definition Dok.-Nr.: D-008 Section 4), encode the DBV Z39.50 structures into a binary stream and transfer them to the origin (provided by the target service provider)

NOTE: The responsibility of adhering to the full Z39.50 state machine for both the origin and target falls to the service user.

# 3. System Architecture

This Section describes the architecture of the DBV-OSI Application software. An initial context analysis is given which is then decomposed to the program architecture. All architectural design is decmposed to a level whereby implementation independence is maintained.

Since all implementations are to be on UNIX systems, then the decomposition to each specific 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 particularly suited. Information on the Colbert methodology can be found in Ref. 1 and Ref. 2. Ref. 2 includes a case study on a previous project undertaken by SIL. In addition, Appendix A includes diagrams pertaining to the Colbert methodology syntax.

## 3.1 Context Architecture

The context architecture of the overall system is described in the Figure 3.1 below. The context of the system must show the system with respect to its external entities. In this case, the system to be decomposed is described as the DBV-OSI II system. Its external entities are the Host databases to which it connects to.

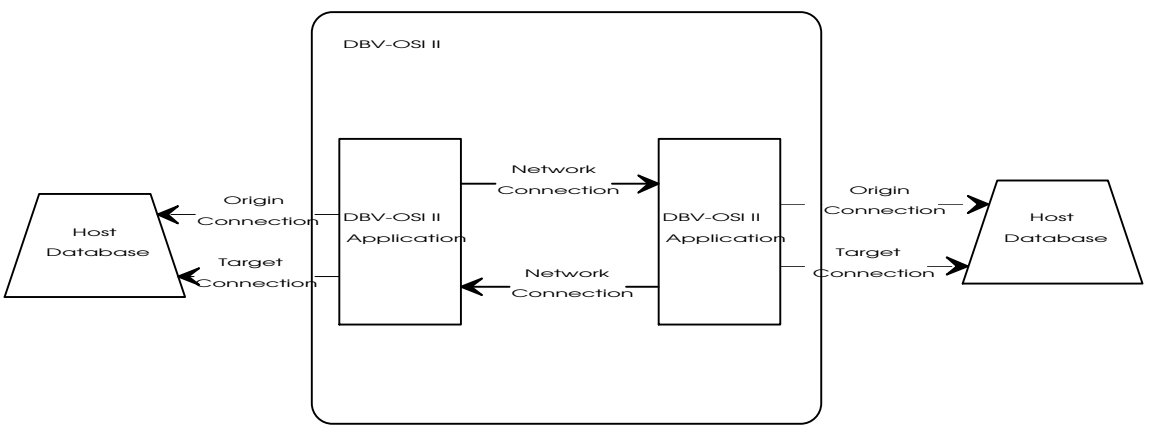



**Figure 3.1 Context Diagram of DBV-OSI II**

In the context analysis, the Host Databases are considered as external entities to the DBV-OSI II itself, since these are not the subject of design in this context. In addition, it should be noted that the direction of the connections from the DBV-OSI II are always towards the external entities. This is because the connection context is also viewed with reference to the DBV-OSI II.

The connections are defined as:

Origin Connection: The DBV-OSI II Application object is looking for a connection from the Host Database

Target Connection: The DBV-OSI II Application is making a connection to the Host Database.

It should be noted that at this stage, the Network connection between the two DBV-OSI II Applications does not consider which network is used, but simply to describe that the two are connected by a network.

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

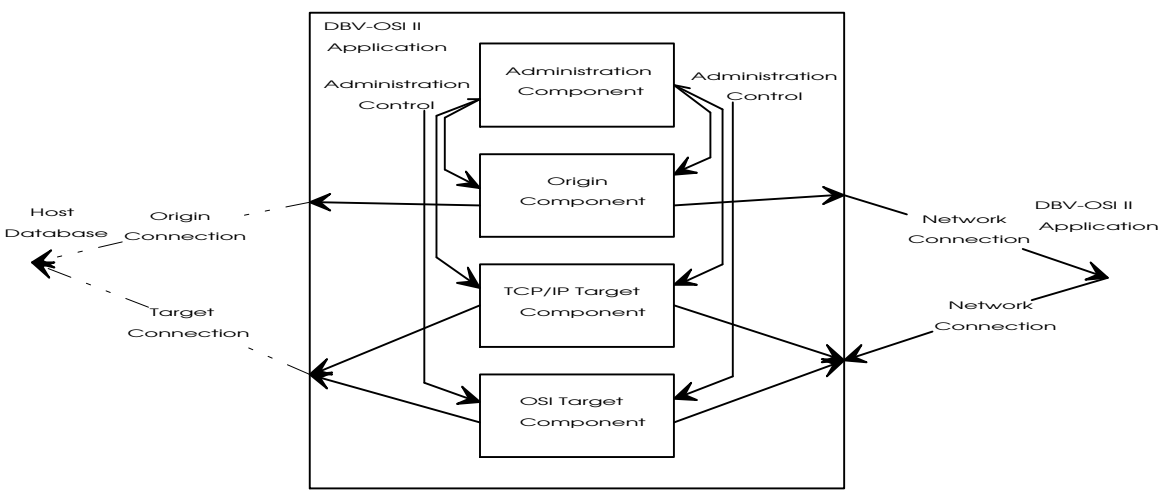



**Figure 3.2 High Level Decomposition of DBV-OSI II Application Object**

In this diagram, it can be seen that the DBV-OSI II application comprises 4 types of object, namely:

Administration Component: This object is responsible for initiating and destructing the other objects defined below. In addition, this object will receive administration messages issued by the other objects.

Origin Component: This object is responsible for processing outgoing connections from the Host Database it is associated with to Target applications in remote DBV-OSI II applications.

TCP/IP Target Component: This object is responsible for processing incoming connections from remote DBV-OSI II Origins over the TCP/IP communication stack.

OSI Target Component: This object is responsible for processing incoming connections from remote DBV-OSI II Origins over the OSI communication stack.

The connection to the Host Database and to the network can now be more clearly defined. However, the exact nature of these connections is better described by further decomposition of each object.

It is the responsibility for each DBV-OSI II Consortium member to develop the service-user part of the Origin/Target programs and the respective connection to the Administration Component. The service-provider part of these programs is in the API.

For the purposes of this document, it is assumed that to satisfy the operation of the Origin Component program; that suitable connection links to the local host will be provided in terms of local network connectivity and also the appropriate software in the Host itself to switch a user session to the Origin Component. A TCP/IP connection is assumed whereby the Host will be a TCP/IP client, and the Origin Component will contain a TCP/IP server. Local system administration will decide upon the network addressing and port numbers to use for this connection. This mechanism will enable multiple connections from the Host 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 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 server process. For both Target components, a local network connection to the Host will be created, and, like the origin Component, TCP/IP is recommended as the communications medium.

Therefore, the **Host database** system need only be modified to have both a TCP/IP client and TCP/IP server to manage outgoing and incoming dialogue respectively.

## 3.2 Multiple Connections

It is recommended that the conventional master-slave architecture is adopted to handle multiple simultaneous connections. The philosophy behind this approach adopts the following procedure:

a) a master target program waits for incoming connection requests from other origins;

b) for each origin process that connects, a slave process is created which communicates with the specific connected origin process. The target master program remains free to accept further connections (upto the physical or system-defined limit of the machine);

c) the target negotiates with the origin specific ports/addresses to use between the slave-target and the origin, and notifies the slave-target on its creation.

Using this master-slave approach, only one set of slave code needs to be written which will be used automatically 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 in-advertantly 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

The Origin Component 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 server.

********** Recommendation **********

The Origin Component itself will contain an TCP/IP server to receive local host sessions. A Master-Slave (parent-child) architecture of this server is advised, described as follows. A master server will receive all incoming client connections on the designated network address. When an incoming client connection is received, the master will designate a legitimate available sub-address (or port) and inform the client that all subsequent communications in this session must use this sub-address (or port). The master then spawns (forks) the slave process which is the Z39.50 Origin process comprising both service-user and service-provider parts. This slave process is then responsible for listening for both the host client communications on the sub-address provided. The Master will resume its permanent responsibility of listening for additional connections from the host clients. In UNIX, the spawning operation is achieved by utilisation of the system call fork(). The slave communications parameters as passed into the spawned process using standard argv and argc mechanism. A slave process is generated for each user connected. This architecture is shown in Figure 3.3 below.

**********

Note that the API shown in the Origin Processes within the dotted lines is the responsibility of the API developer. The remainder is the responsibility of the DBV-OSI II Partner.

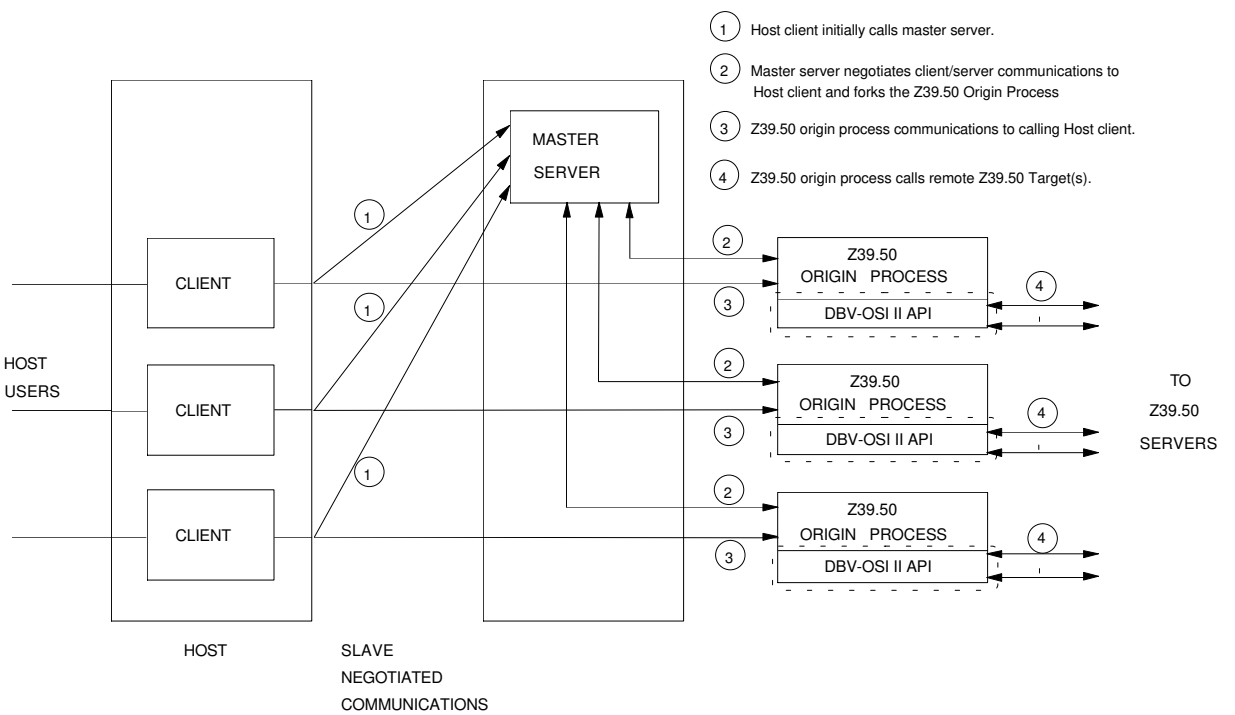


**Figure 3.3 Possible Client/Origin Architecture**

The development of the communication between the Origin Application and the host is fully the responsibility of the Consortium members.

The architecture above can be considered as an O-O architecture with each user and its spawned process represented as an object. This is illustrated below in Figure 3.4 Referring to this diagram, followed by a brief explanation of its behaviour .

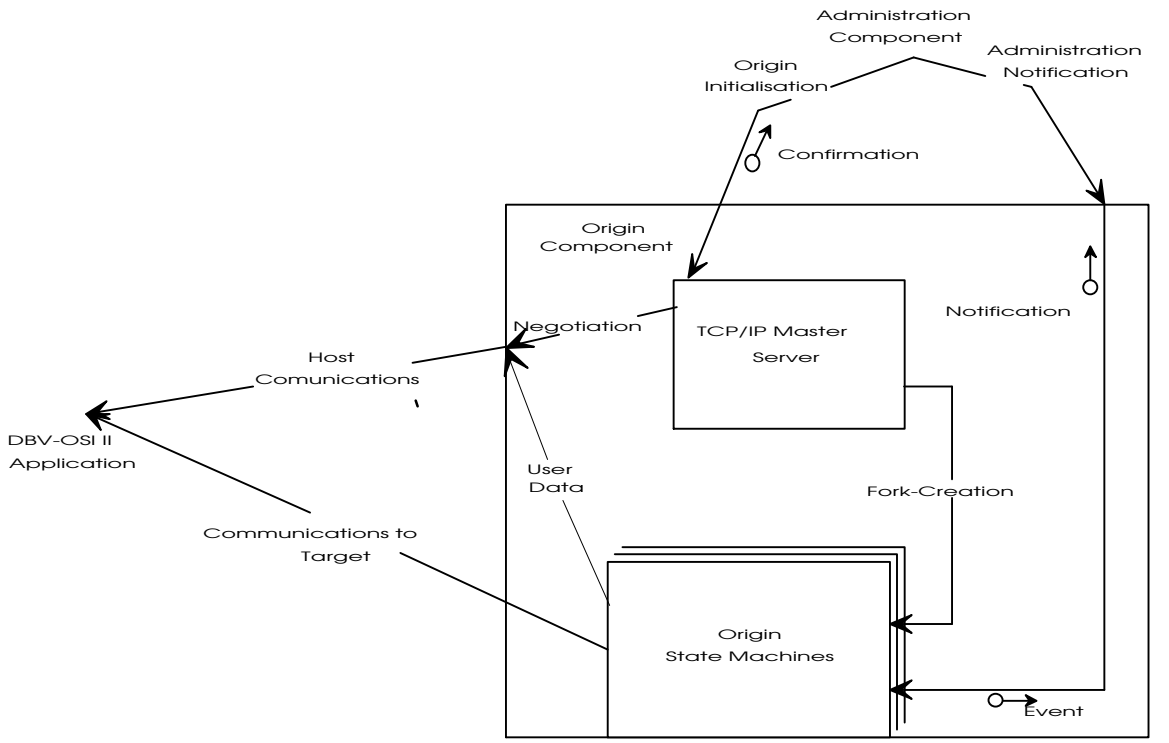



**Figure 3.4 Decomposition of DBV-OSI II Origin Component**

The Origin Component comprises two main objects, namely a TCP/IP server to process user session connections from the Host, and the Origin State Machine 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 Server will spawn the Origin State machine as a separate program. Each user session is managed by separate process instances within the Origin Component, and hence the "collection" symbol for the Origin State Machine is used.

With respect to the Administration Component, the origin master server can be started (or stopped) as required by the Administration Component operator. A TCP/IP master server program will be supplied by SIL and this will be start/stopped by the administration component. The design of the TCP/IP server program will be very similar to the target front end TCP/IP master server. Using this method allows the administrator to initialise or shut down the Z39.50 service without effecting any other TCP/IP services on the system.

The Origin State Machines will 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 State Machine.

### 3.2.2 TCP/IP Target Component Architecture

The design of the server can be quite complicated with respect to the target application and the utilisation of the API. The design should also take into account development issues such as target program complexity, development debugging and operations reliability. A block diagram of a possible server/target architecture is illustrated in Figure 3.5.

The development of the communication between the Target Application and the host is fully the responsibility of the Consortium members.

Note that the API shown in the Target Processes within the dotted lines is the responsibility of the API developer. The remainder is the responsibility of the DBV-OSI II Partner.

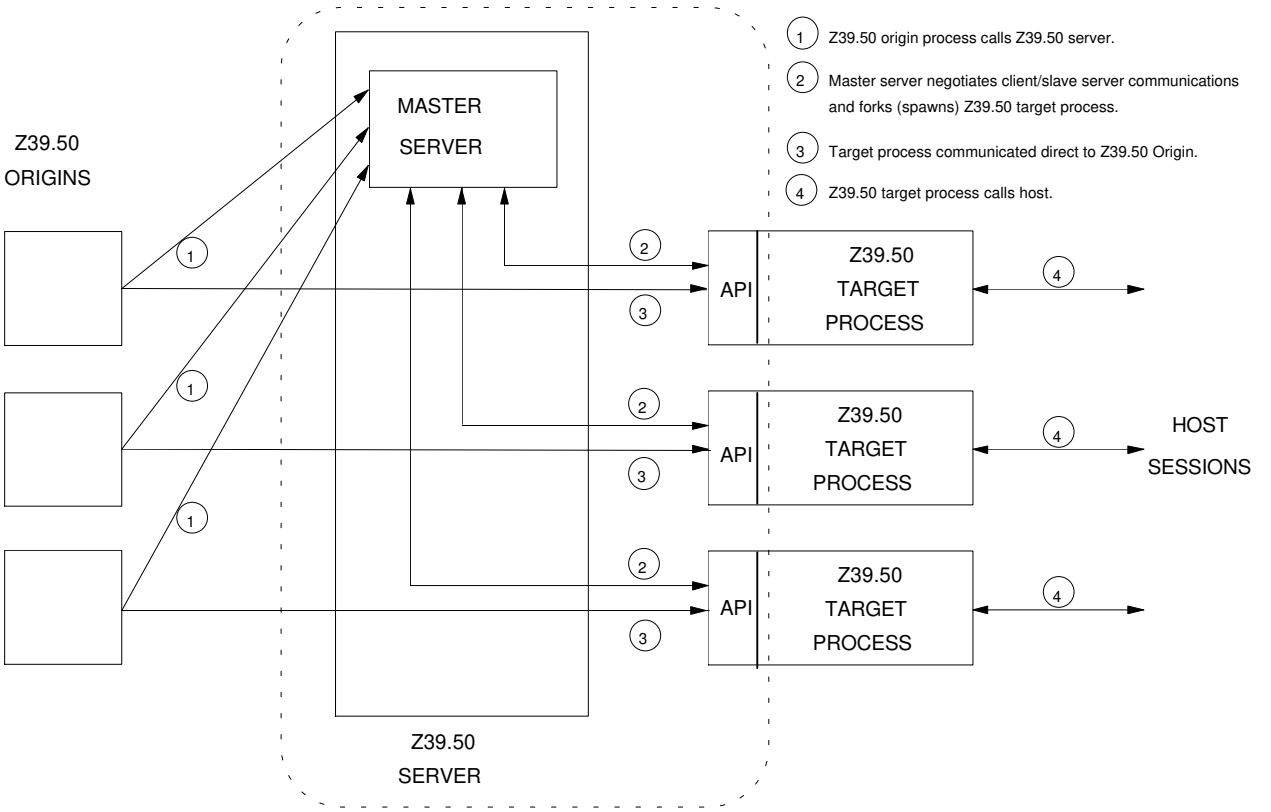



**Figure 3.5 Block Diagram of Target Architecture**

The trade-off of this architecture is that more memory in the UNIX computer is used when there are multiple connections. It is likely however that the cost of additional computer memory would be cheaper than the cost of developing a more complex multi-session program.

By utilising the above architecture, the O-O decomposition of the TCP/IP Target Component is shown in Figure 3.6 below. Referring to this diagram, a brief explanation of its behaviour is given below.

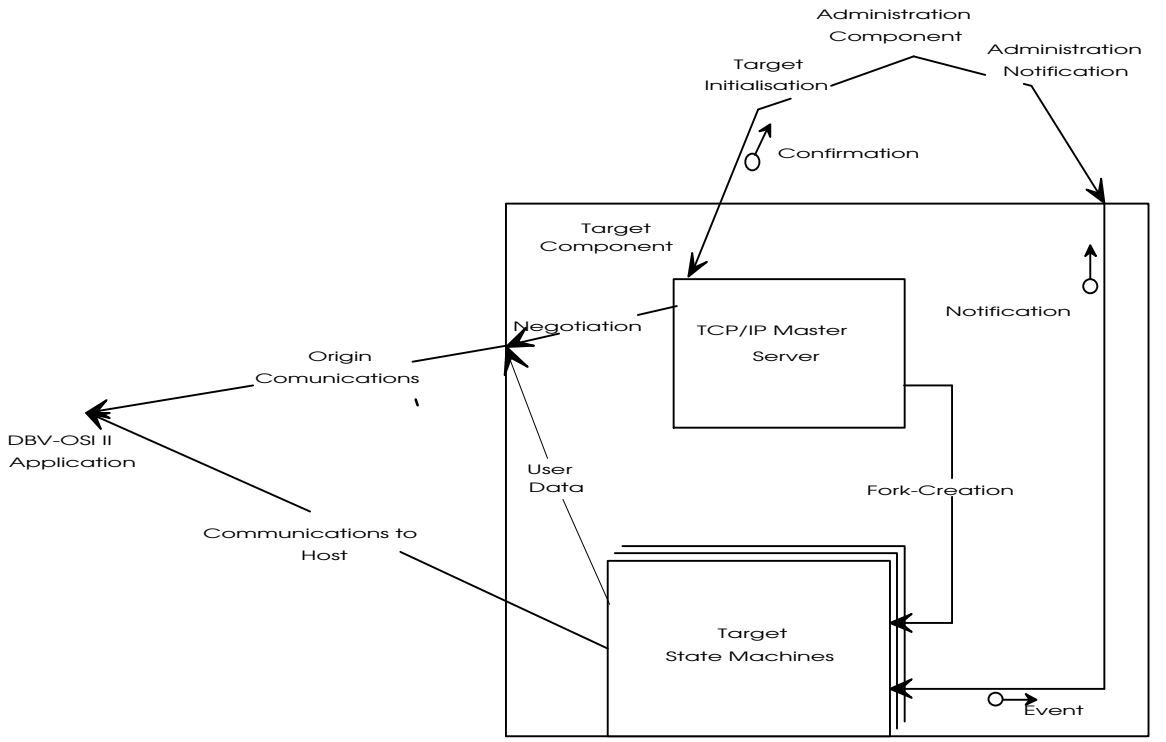



**Figure 3.6 Decomposition of DBV-OSI II TCP/IP Target Component**

The TCP/IP Target Component comprises two main objects, namely a TCP/IP Server to receive remote host Origin connections and the Target State Machine 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 Master Server would spawn the Target State Machine 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 State Machine is used.

With respect to the Administration Component, the TCP/IP Target Component can be initialised (or stopped) as required by the Administration Component operator. The TCP/IP master server will be a program supplied by SIL. Using this method allows the administrator to initialise or shut down the Z39.50 service without effecting any other TCP/IP services on the system. It also

means that the Administration Component has a greater control over access to the whole target component system and can be used to easily allow or deny it.

With respect to the Administration Component, the target master server can 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 State Machine will notify the Administration component of various Event messages. Since any number of concurrent Target State Machines will exist at any time, each Event message will be tagged by the UNIX process Id of that particular Target State Machine.

The program **targetd** is an example TCP/IP server master 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.

### 3.2.3 OSI Target Component Architecture

The architecture of the OSI Target Component is shown in Figure 3.7 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.

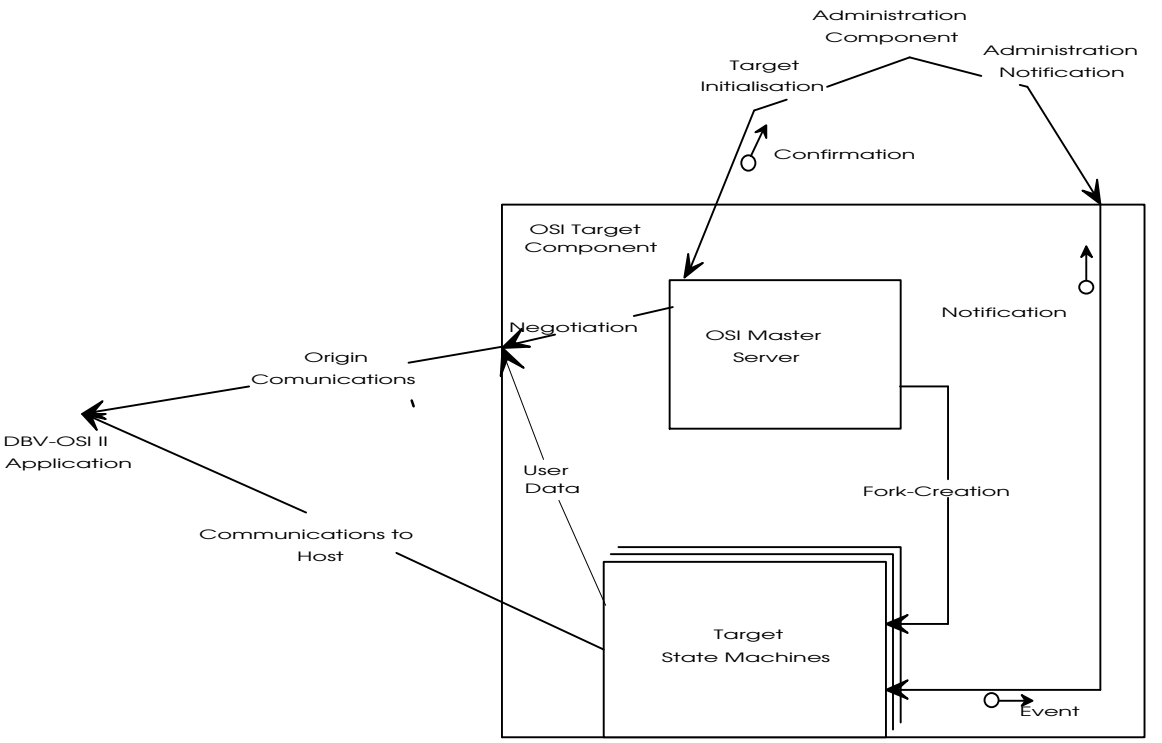



**Figure 3.7 Decomposition of DBV-OSI II OSI Target Component**

The OSI Target Component comprises two main objects, namely an OSI Server to receive remote host Origin connections and the Target State Machine 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 Master Server would spawn the Target State Machine 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 State Machine is used.

With respect to the Administration Component, the OSI Target Component can be Initialised (or stopped) as required by the Administration Component operator. The program used will be the daemon executable 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 server is not started, a facility to start it will be provided.

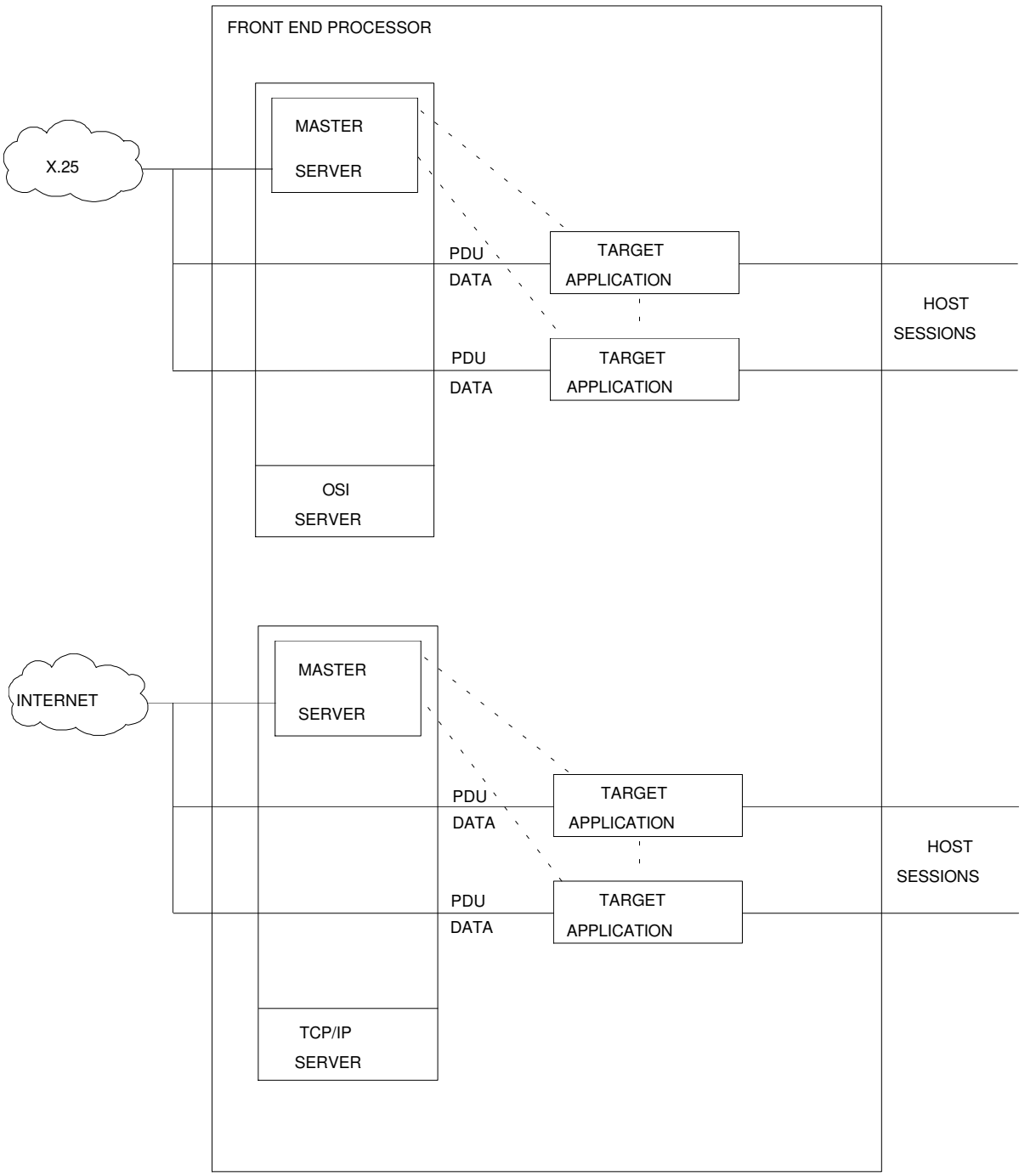
The name of the daemon process used by ISODE is called **tsapd.** However, for the ISODE Consortium Release 2, an additional daemon has been supplied called **iaed** which is the recommended daemon to use. The use of **iaed** will have no impact upon the Origin or Target processed to be developed by 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 State Machines will notify the Administration component of various Event messages. Since any number of concurrent Target State Machines will exist at any time, each Event message will be tagged by the UNIX process Id of that particular Target State Machine.

### 3.2.4 Different Communication Stacks

The architecture described above can be seen to readily support different communication stacks as illustrated in Figure 3.8. A separate server will exist for (say) OSI to that of TCP/IP and both will utilise a similar mechanism to spawn the same Target State Machine program. This will be achieved by the Server (Either TCP Master or OSI Daemon) calls the Target State Machine program passing a parameter to the Target State Machine program implying which server invoked the program. In either case, the data between the servers 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 server 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 3.8 Block Diagram Showing use of Different Stacks by Servers**

# 4 System Development Considerations

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

## 4.1 Origin And Target Component Analysis

The development of the Origin and Target components will 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 language translation;

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 PDU 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 a PDU encode and send function is called.

********** Recommendation **********

In terms of the service-user components, at the very least it should conform to the following:

a) Do not attempt to call Protocol Z-Association and operation functions until an association connection has successfully been accomplished to the target;

b) Do not attempt Protocol O-Association operations 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.

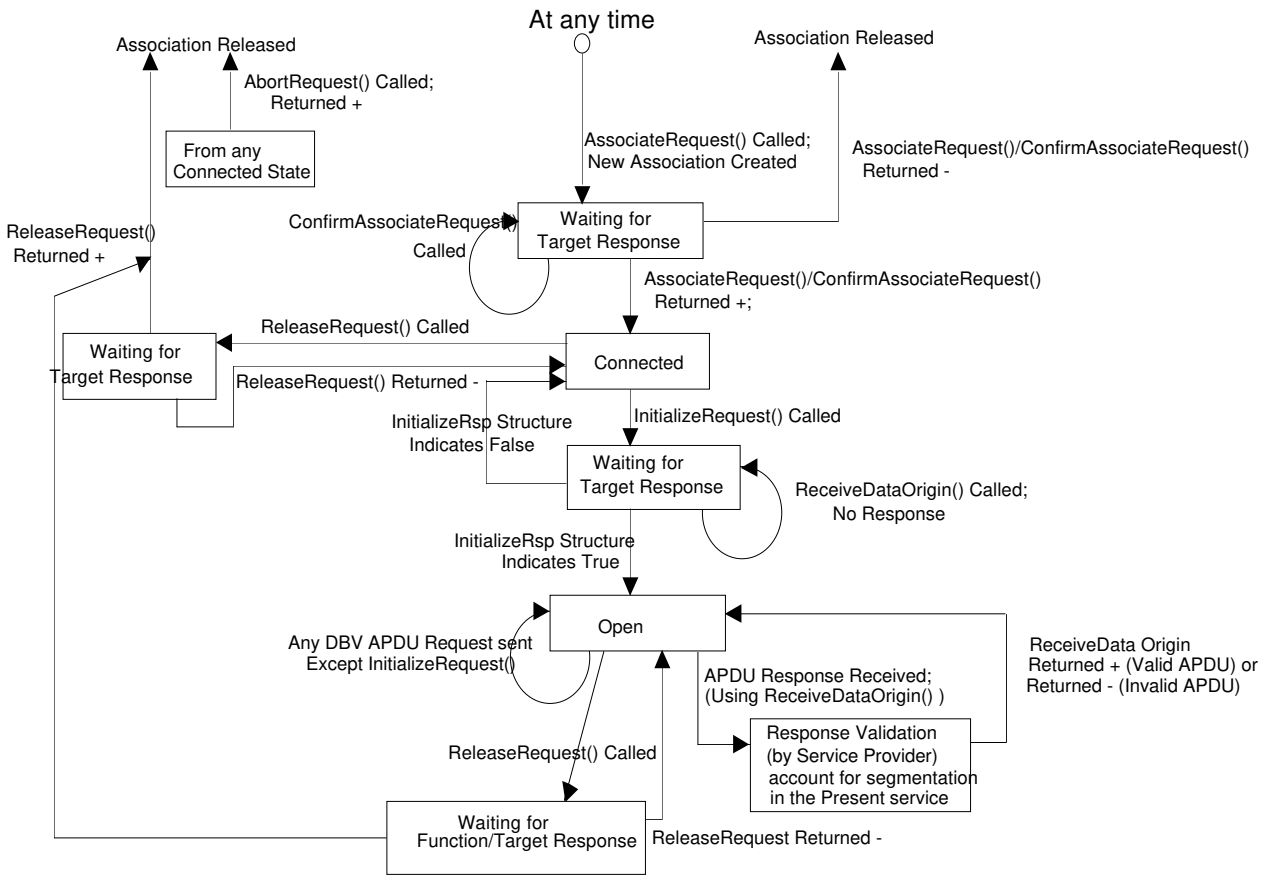



**Figure 4.1 State Transition Diagram of Origin Service Provider**

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().

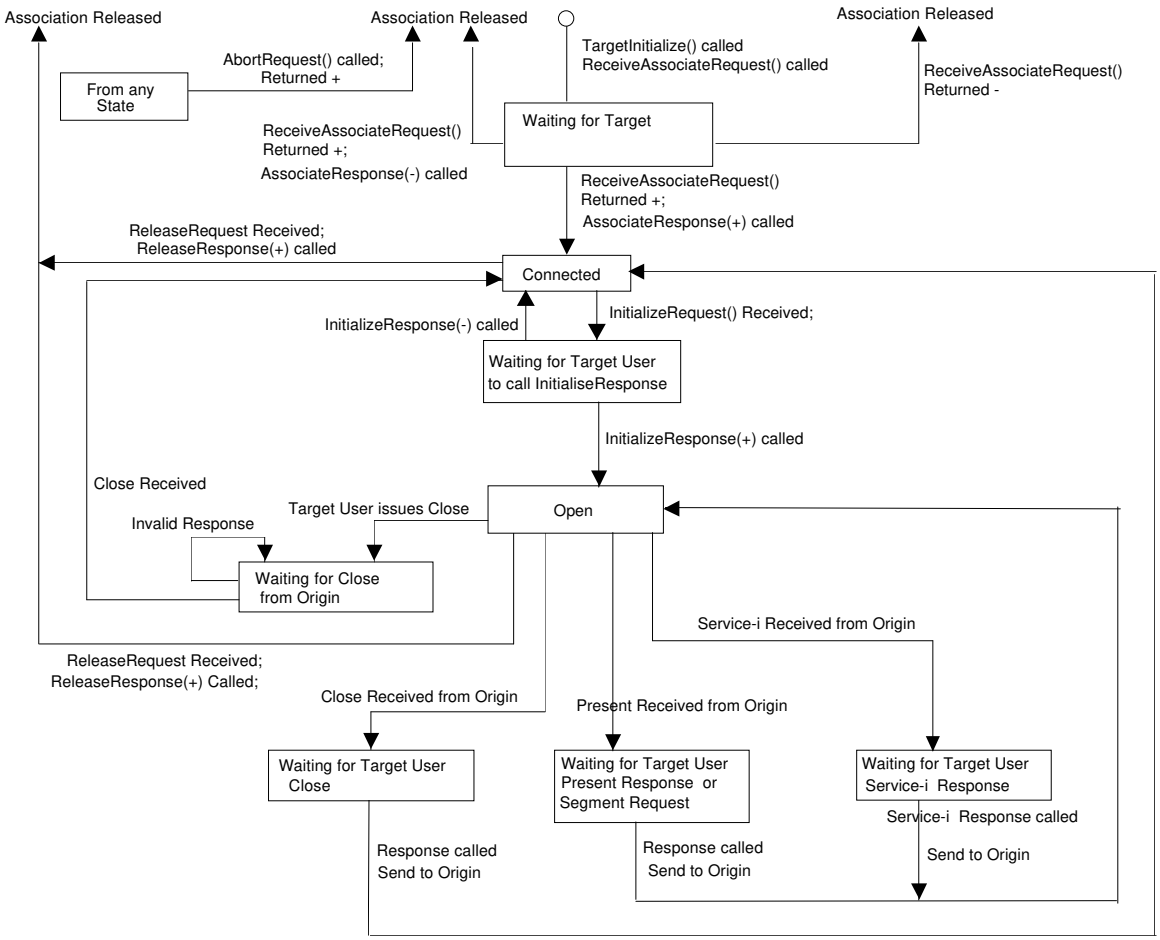



**Figure 4.2 State Transition Diagram of Target Service Provider**

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.

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:

a) Protocol Z-Association and operation functions will only be valid if an association connection has successfully been accomplished, (and not inadvertantly 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;[1] The exception for this is the Present Operation, in which valid Responses are the Present response and Segment requests.

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

[1] This is because Access and Resource control are not being implemented

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 PDU 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

### 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. (1)

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.

(1) (These relationships were however included in Draft 9 of Z39.50)

********** Recommendation **********

Below are listed the valid incoming events for the Origin state machine. These have been enumerated and the comments indicate the source of the input. The prefix SMI in this case is abbreviation from State Machine Input. In addition, the comments indicate from where the input comes from, and distinguishes between the Z- and O- associativity. It is recommended that these are used by the project partners.

**********

```
/* valid client incoming events */
typedef enum sm_input   {
 SMI_InitReq,                 /* Init request (from application) */
 SMI_InitRespPDUAcc,          /* Init_response PDU accept (from target) */
 SMI_InitRespPDURej,          /* Init response PDU reject (from target) */
 SMI_OpReq,                   /* Operation request (from application) */
 SMI_EndOpInd,                /* End-operation indication (from application) */
 SMI_CloseReq,                /* Close request (from application) */
 SMI_ClosePDU,                /* Close PDU (from target) */
 SMI_CloseResp,               /* Close response  (acknowledgement from application) */
 SMI_AnyOpPDU,                /* Any Op PDU (from target whilst waiting to close) */
 SMI_SegPDU,                  /* Segment PDU (from target) */
 SMI_PresentRespPDU           /* Present response PDU (from target) */
} SM_Input;
```

Below are listed the valid outgoing actions for the Origin state machine. These have been enumerated and the comments indicate the direction of the Output. The prefix OOA in this case is an abbreviation from Origin Outgoing Action. In addition, the comments indicate where the output goes to.

```
/* valid client outgoing actions  */
typedef enum sm_OO_action  {
OOA_Init_Req_PDU,            /* Init PDU (to target) */
OOA_Init_Conf_APP,           /* Init confirmed accept (to application) */
OOA_Init_Rej_APP,            /* Init confirmed reject (to application) */
OOA_Close_PDU,               /* Close PDU (to target) */
OOA_Close_Ind_APP,           /* Close indication (to application) */
OOA_Close_Conf_APP,          /* Close confirm (to application) */
OOA_Seg_Ind_APP,             /* Segment indication (to application) */
OOA_Pres_Ind_APP,            /* Present response indication (to application) */
OOA_Op_Ind_APP               /* Operation response indication (to application) */
} SM_OO_Action;
```

Below are listed the valid states for the Origin state machine. These have been enumerated. The comments identify the name of the states. Note that an additional pseudo state SMS_QUIT has been added. This may be useful if when the Z-associativity has closed, the program may terminate. The prefix SMS is abbreviated from State Machine State.

```
/* valid client states */
typedef enum sm_state  {
SMS_Closed,             /* Closed */
SMS_Init_Sent,          /* Init Sent */
SMS_Serial_Idle,        /* Serial operations idle */
SMS_Serial_Active,      /* Serial operations active */
SMS_Close_Sent,         /* Close sent */
SMS_Close_Rcvd,         /* Close received */
SMS_Present_Sent,       * Present sent */
SMS_Rsc_Recvd,          /* Rsc received */
SMS_Op_Sent,            /* Operation sent */
SMS_QUIT                /* Termination of Program */
} SM_State;
```

Below are listed the valid state machine actions defined in the Z39.50 state tables. Some of these are more difficult to interpret from the tables than are the events, inputs or outputs. The abbreviation SMA is for State Machine Action.

```
/* valid client state-machine actions */
typedef enum sm_action  {
SMA_StateChange,                /* change to the new state */
SMA_SetReturnState,             /* set the return state when actions are completed */
SMA_GoToReturnState,            /* change state to the return state */
SMA_OutgoingAction,             /* client outgoing actions */
SMA_ClearOpCount,               /* set the number of operations to zero */
SMA_TestOpMode,                 /* test if serial or concurrent operations mode */
SMA_Serial,                     /* initiate program parameters for serial mode */
SMA_InitiateOperation,          /* initiate a Z39.50 O-machine operation */
SMA_KillOperations,             /* Kill all O-machine operations */
SMA_DecrementOpCount,           /* decrement the O-machine count */
SMA_TestOpCount,                /* test the operations count for > 0 */
SMA_OpCountZero,                /* process behaviour - operation count = 0 */
SMA_OpCountNonZero,             /* process behaviour - operation count > o */
SMA_FlagEndOp                   /* internally generate notificaion that an opreation has completed */
} SM_Action;
```

Below are listed the valid modes for Z39.50 operations, either Serial or Concurrent. The abbreviation SMOP is for State Machine Mode.

```
/* serial or concurrent operations  */
typedef enum sm_op_type {
                        SMM_Serial,
                        SMM_Concurrent
} SM_OpType;
```

Below are listed the valid operation types for Z39.50 operations. The abbreviation SMO is for State Machine Operation.

```
/* internal client operations */
typedef enum sm_operation {
SMO_Search,
SMO_Present,
SMO_DeleteSet,
SMO_Scan,
SMO_Sort,
SMO_ResourceReport,
SMO_ExtendedService,
SMO_PresentSegment
} SM_Operation;
```

The behaviour of the state machine must conform to the state tables in Z39.50.

### 4.3.2 Origin Application State Machine Behaviour

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.

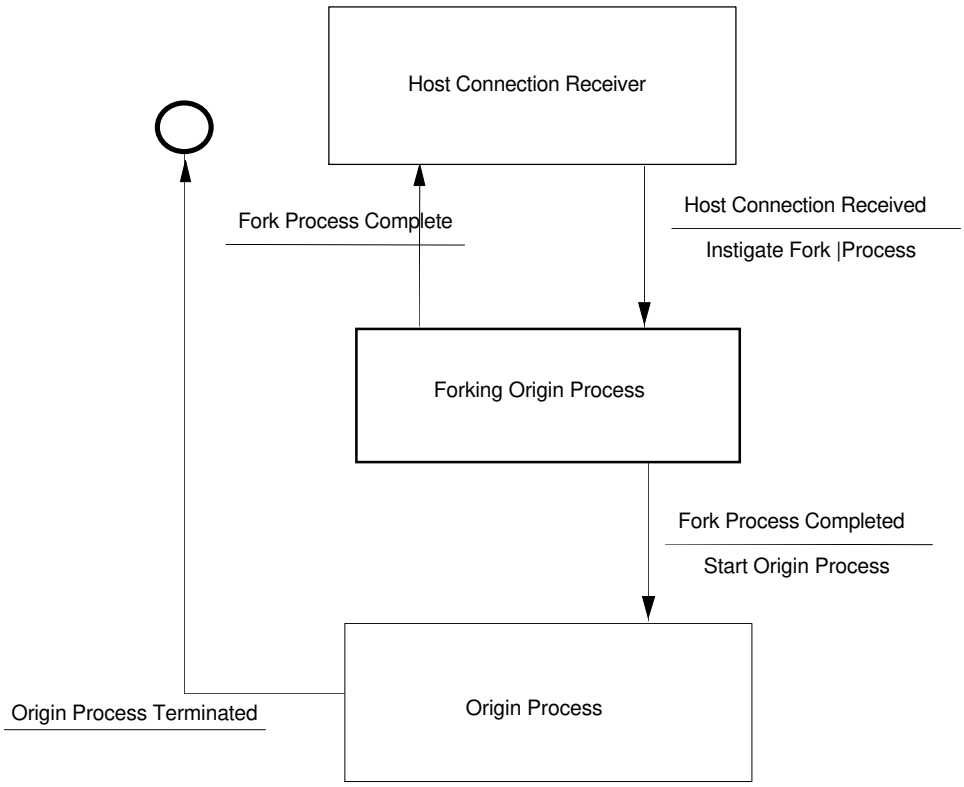



**Figure 4.3 High Level State Transition of Origin Process**

In the high level state transition diagram of Figure 4.3 above, when the Origin Host connection receiver receives a host/user connection, it will cause a state transition to start the forking process. This in turn will cause the spawning of the Origin program. Note that the forking 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 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.

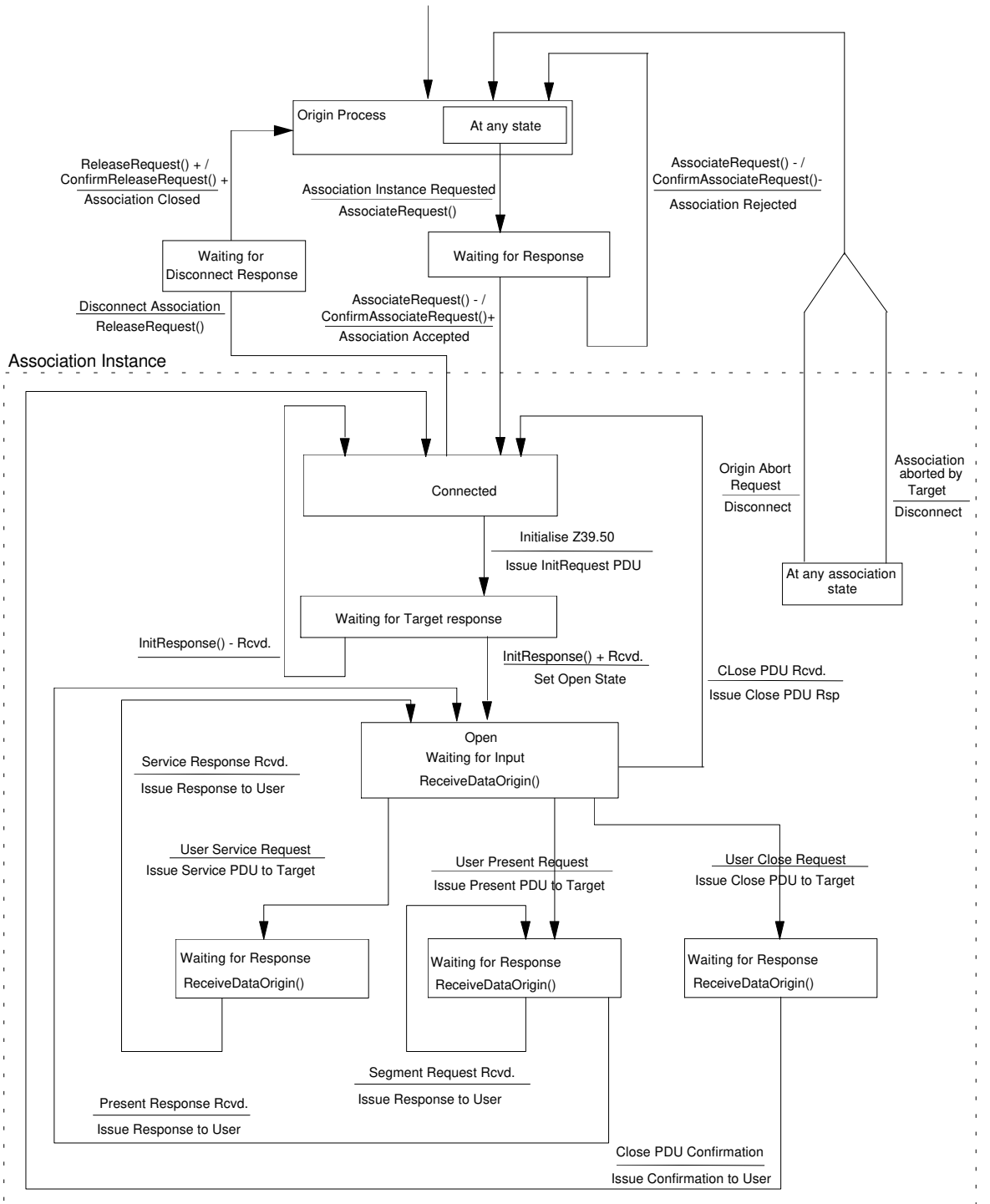



**Figure 4.4 Detailed State Transition of Origin Process**

### 4.3.3 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. (1)

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.

(1) (These relationships were however included in Draft 9 of Z39.50)

********** Recommendation **********

Below are listed the valid incoming events for the Target state machine. These have been enumerated and the comments indicate the source of the input. The prefix SMI in this case is abbreviation from State Machine Input. In addition, the comments indicate from where the input comes from, and distinguishes between the Z- and O- associativity.

**********

```
/* valid target incoming events */
typedef enum sm_input   {
 SMI_InitPDU,                  /* Init PDU (from origin) */
 SMI_InitRespAcc,              /* Init_response accept (from application) */
 SMI_InitRespRej,              /* Init response reject (from application) */
 SMI_OpPDU,                    /* Operation PDU (from origin) */
 SMI_CloseReq,                 /* Close request (from application) */
 SMI_ClosePDU,                 /* Close PDU (from origin) */
 SMI_CloseResp,                /* Close response  (acknowledgement from application) */
 SMI_AnyPDU,                   /* Any PDU (from origin) */
 SMI_EndOpInd,                 /* End-operation indication (from application) */
 SMI_SegReq,                   /* Segment request (from application) */
 SMI_PresentResp,              /* Present response (from application) */
 SMI_OpResp                    /* Operation response (from application) */
} SM_Input;
```

Below are listed the valid outgoing actions for the Target state machine. These have been enumerated and the comments indicate the direction of the Output. The prefix TOA in this case is an abbreviation from Target Outgoing Action. In addition, the comments indicate where the output goes to.

```
* valid target outgoing actions  */
typedef enum sm_TO_action  {
TOA_Init_Ind_App              /* Init indication (to application) */
TOA_Init_Conf_PDU,            /* Init confirmed accept (to origin) */
TOA_Init_Rej_PDU,             /* Init confirmed reject (to origin) */
TOA_Op_PDU                    /* Operation response PDU (to origin) */
TOA_End_Op                    /* Internally generated End-operation */
TOA_Close_PDU                 /* Close PDU (to origin) */
TOA_Close_Ind_APP             /* Close indication (to application) */
TOA_Close_Conf_APP,           /* Close confirm (to application) */
TOA_Seg_PDU                   /* Segment request PDU (to origin) */
TOA_Pres_PDU                  /* Present response PDU (to origin) */
} SM_OO_Action;
```

Below are listed the valid states for the Target State Machine. These have been enumerated. The comments identify the name of the states. Note that an additional pseudo state SMS_QUIT has been added. This may be useful if when the Z-associativity has closed, the program may terminate. The prefix SMS is abbreviated from State Machine State. Note that since Origin and target systems are independently built, any similarity to State Machine naming for the Origin listed in the previous section should not be of concern.

```
/* valid server states */
typedef enum sm_state  {
SMS_Closed,             /* Closed */
SMS_Init_Recvd,         /* Init Request received */
SMS_Serial_Idle,        /* Serial operations idle */
SMS_Serial_Active,      /* Serial operations active */
SMS_Close_Sent,         /* Close sent */
SMS_Close_Rcvd,         /* Close received */
SMS_Present_Rcvd,       /* Present received */
SMS_Op_Recvd,           /* Operation received */
SMS_QUIT                /* QUIT Application */
} SM_State;
```

Below are listed the valid Target state machine actions defined in the Z39.50 state tables. Some of these are more difficult to interpret from the tables than are the events, inputs or outputs. The abbreviation SMA is for State Machine Action.

```
/* valid target outgoing actions */
typedef enum sm_action  {
SMA_StateChange,                /* change to the new state */
SMA_SetReturnState,             /* set the return state when actions are completed */
SMA_GoToReturnState,            /* change state to the return state */
SMA_OutgoingAction,             /* server outgoing actions */
SMA_ClearOpCount,               /* set the number of operations to zero */
SMA_TestOpMode,                 /* test if serial or concurrent operations mode */
SMA_Serial,                     /* initiate program parameters for serial mode */
SMA_ResponseReqd,               /* process behaviour for resource response */
SMA_ResponseNotReqd,            /* process behaviour - no resource response required */
SMA_InitiateOperation,          /* initiate a Z39.50 O-machine operation */
SMA_KillOperations,             /* Kill all O-machine operations */
SMA_DecrementOpCount,           /* decrement the O-machine count */
SMA_TestOpCount,                /* test the operations count for > 0 */
SMA_OpCountZero,                /* process behaviour - operation count = 0 */
SMA_OpCountNonZero,             /* process behaviour - operation count > o */
SMA_FlagEndOp                   /* internally generate notificaion that an opreation has completed */
} SM_Action;
```

The valid modes for Z39.50 Target operations and the valid operation types for Z39.50 Target operations are the same as for the Origin Component described in the previous section.

### 4.3.4 Target State Machine Behaviour

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.

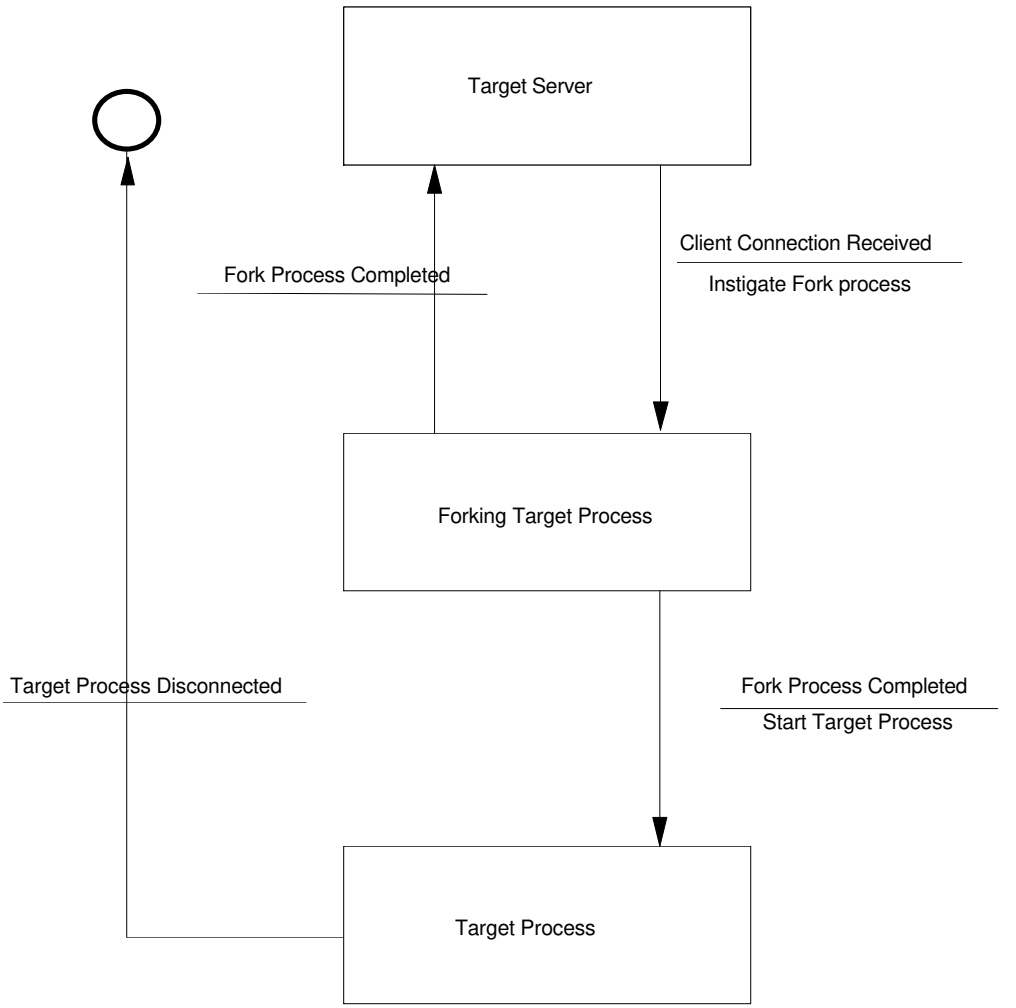



**Figure 4.5 High Level State Transition of Target Process**

In the high level state transition diagram of Figure 4.5 above, when the Target Server receives a Client connection, it will cause a state transition to start the forking process. This in turn will cause the spawning of the Target program. Note that the forking transition will return state immediately to the Target Server, and that when the Target 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 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.

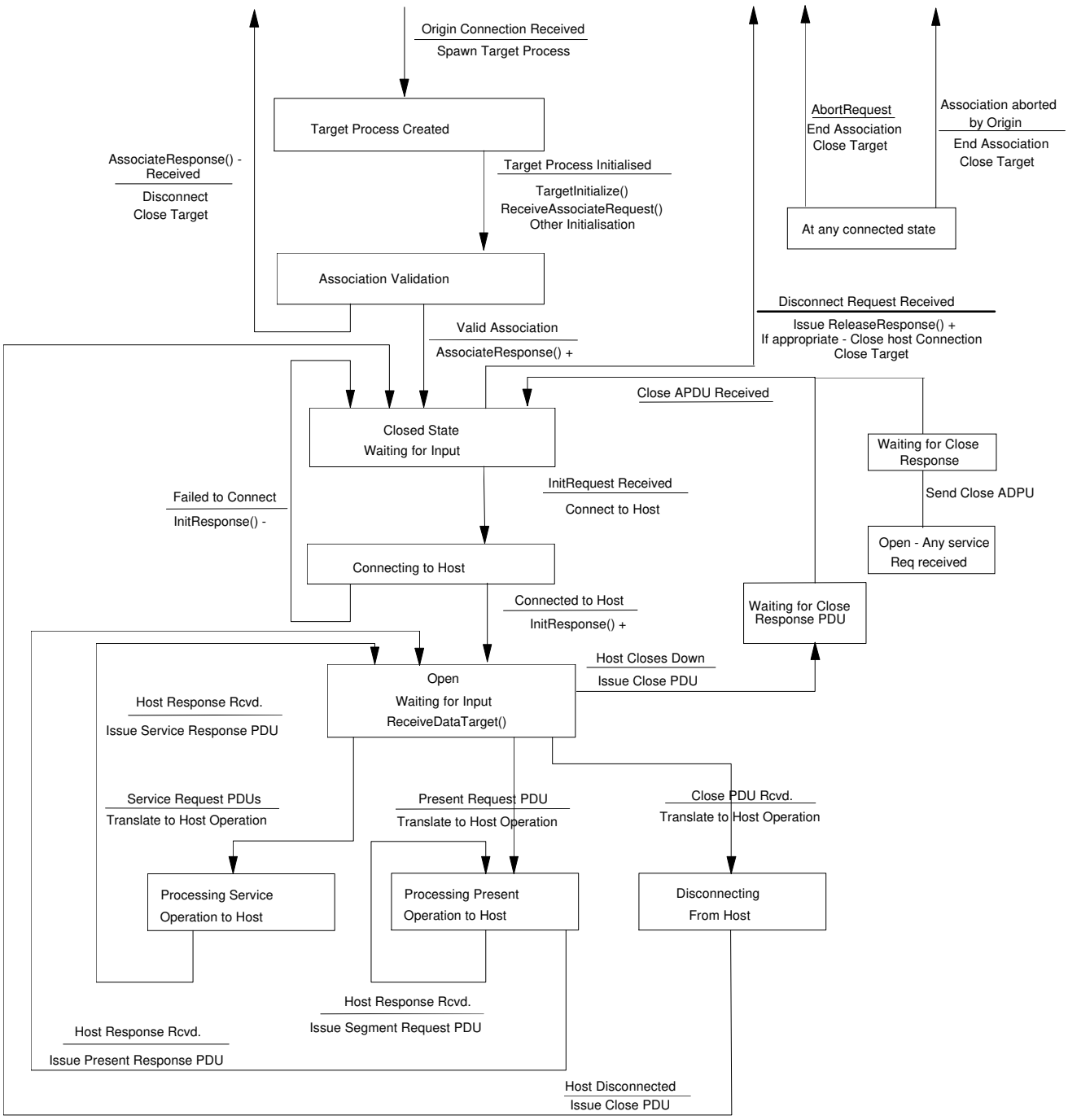



**Figure 4.6 Detailed State Transition of Target Process**

### 4.3.5 Implementation of the State machine

********** Recommendation **********

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

The program can follow a three stage operation, 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) feed event through state machine.

The implementation of the State machine can follows a three level operation namely:

a) identify State;

b) validate incoming Event; and

c) process Action(s) to take and identify transition state.

**********

When the incoming Application 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 client/server 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 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 will consist 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[2]

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:

| | |
|---|---|
| **si_defs.h** | - DBV OSI II specified definitions |
| **si_types.h** | - DBV OSI II specified types |
| **dbv_proto.h** | - prototypes of the API functions |

Notes:

a) In order to include the API library functions in origin and target processes, the three header files above 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.

Pure TCP/IP libraries will also included for those implementors which will not be using OSI communications. These will be named **libdbvtcp**, **libdbvorigintcp** and **libdbvtargettcp**.

---

[2] During initial development all libraries will be provided as .a static libraries

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 "si_defs.h"
#include "si_types.h"
#include "dbv_proto.h"

/* code to implement a target state machine 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 PDU and to the necessary elements within the PDU 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.

One consideration of the system is the interoperability with other developments in the DBV-OSI consortium. The behaviour, supported operations and interpretation of commands should be clearly defined as a high priority input to the program development. This process is termed "profiling" the Z39.50 services.

The target component has the reverse translation aspect. It must map incoming Z39.50 PDU requests to its local host query language, and translate the local host responses to Z39.50 response PDUs. Some potentially difficult aspects are:

a) Some commands of the query language may not have an appropriate Z39.50 PDU 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 PDU 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 PDU 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 Structure Generator

********** Recommendation **********

It is recommended that the developer create a RPN structure conversion function to convert the user query syntax to the structure format for the Query API function.

For simple queries, the RPN structure will be easier to create and initialise. For a query that has the slightest complexity, setting up the RPN structure will become difficult and time-consuming.

**********

Consider the following command which includes Boolean and positional operators represented in the ESA-Quest command language:

**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. |

********** Recommendation **********

It is advised that a generic translation mechanism be developed. Where possible, specific definitions of query language syntax should be represented in tabular format dynamically loaded at run-time. This will facilitate future program operations and maintenance aspects.

**********

### 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 server 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 clients (e.g. Willow) have a huge range of selectable predefined attribute combinations. This allows a client to work successfully with different Z39.50 servers. In addition, some clients often allow attribute combinations to be

determined by a user for a search. This is because the sets used by some servers differ due to the intpretation of the Z39.50 specifications by the implementors.

For example, some American V2 servers require a Bib-1 search term attribute list with the following format

1,<Use Attribute> 2.0 3,3 4,6 5,0 6,0

Note that the attribute set tables for which no attribute is to be specified are given a zero attribute.

Other examples are:

a) a Title search:

the bib-1 attributes are: 1,4 2,0 3,3 4,6 5,0 6,0

b) a Subject search:

the bib-1 attributes are: 1,21 2,0 3,3 4,6 5,0 6,0

### 4.5.2 Locally Stored Information

It is highly likely that the Origin and Target programs will need to retain information about the user during the user session. Some examples of this follow.

#### 4.5.2.1 Query Attributes

As mentioned previously, the Origin may need to retain particular user information. for the current user session. Such information may include:

- query limit parameters;
- selected database; and
- record of sets generated.

This information may be discarded on user session termination.

### 4.5.2.2 Connections From Target To Hosts

********** Recommendation **********

Each target process must use an appropriate mechanism for connection to its local host. This is usually solved by adopting a script file mechanism that defines the host login procedure. The script file should contain special parameters for dealing with events such as:

**********

a) parsing in the user Id and password,

b) recognition of going on line;

c) recognition of failure situations; and,

d) recognition of going off-line.

### 4.5.2.3 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 server 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. The Master Server code is typically an infinite loop with less than a page of C code in length.

********** Recommendation **********

Any unrecoverable errors in the slave processes should be trapped, and should produce a relevant diagnostic message followed by an orderly exit of the slave process. A "reaper" function should 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 master code. When this is set up; if a slave process created by a master should terminate (ideally through the signal trap, exit(1) sequence specified above), the reaper function will be automatically called. This will terminate the exited slave 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 master 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 creates a slave>
}

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 communication function returns an error indication, appropriate action must be taken. At the very least, the following should be done:

a) issue an error message to the Administration Component; 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 slave 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 erroneous slaves are being created continuously; thereby filling the UNIX system process table resulting in a system crash.. From experience, the only recourse to this situation is to reboot the machine.

## 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.

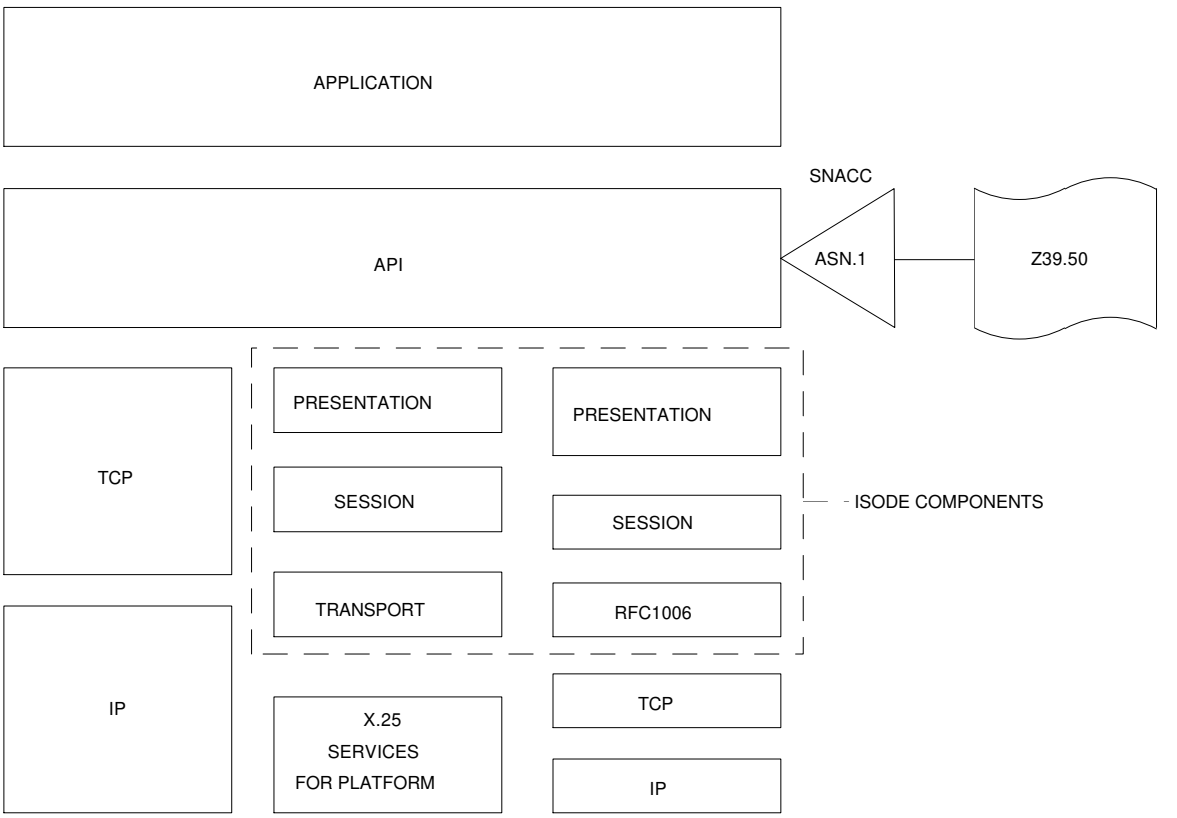



**Figure 4.7 Representation of Communications Stack Components**

### 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.

## 4.7 The Administration Component

An administration component will be provided as part of the DBV-OSI II application. The aim of this component will be to start, monitor, control and provide diagnostic information about origin and target programs. Figure 4.8 shows the flows of data and control concerning the administration tool.

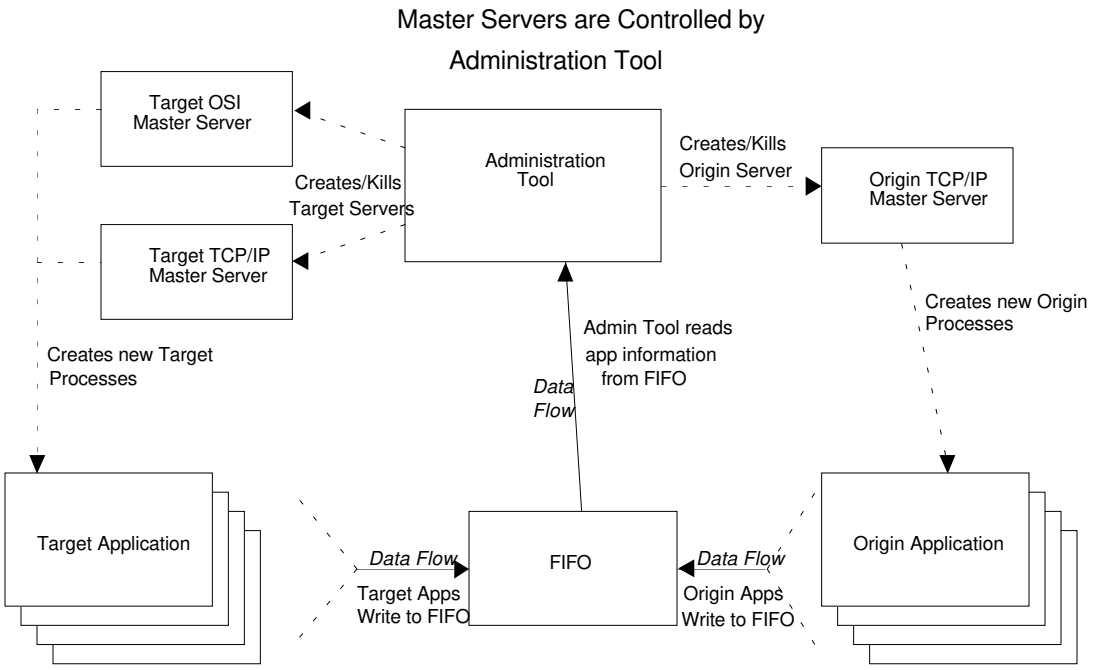



**Figure 4.8 Representation of Communications Stack Components**

The administration component will provide the following:

a) A means of starting and stopping origin and target server processes. A menu approach will be used, and the server processes will notify the administration component with a status message when started. Stopping a server process will be a more difficult operation, particularly if there are users accessing the system. In this context, the administration tool will be configured so as to know how many users are on the system, and the UNIX processes associated with the users. Each user related UNIX process can also be stopped by the administration component if necessary.

   The administration tool will start the TCP/IP master server processes using the UNIX C commands fork() and execlp() and will stop processes by using the UNIX command Kill. The master server will be started and stopped using the daemon specific UNIX commands. The administration component will allow the administrator to stop the servers at any time, but will check for users accessing the system beforehand and prompt accordingly.

   The administration component will be manipulating the system OSI daemon to remove/include the Z39.50 service. This means that while the Z39.50 service is being reset, all OSI communications on that machine will be made unavailable.

b) A means of identifying the system usage, and who the users are. Both origin and target processes can be monitored.

c) A means of diagnosing origin and target process problems. Even if these programs crash for any reason, the UNIX operating system can enable system signals to be trapped and special signal handling subroutines can be used to notify the administration tool prior to the program exiting. This will be achieved by the origin and target signal handlers sending messages to the administration component before exiting.

d) A means for allowing the addition, deletion and modification of address and port details of remote target systems. The administration component will allow editing of this file through a standard text editor. The origin application program may make use of this file to identify the target addresses which may be passed to the AssociateRequest() function.

The format of the file will be straighforward and will be similar to:

<DatabaseName>, TCP, <Address>, <PortNumber>;

<DatabaseName>, OSI, <PSel>, <SSel>, <TSel>, <NSap>;

Additional details can be found in the document "Software Internal Specification", SIS.

Other features such as the monitoring of user commands to form statistical and accounting information, access Id and password storage, unique retained associativity numbers and parameter configuration (such as network address modification) may be added.

It is not essential to have the Administration Component continually running. If, however the FIFO mechanism is used in the origin or target applications, then a check mechanism will need to be utilised in case the FIFO write fails. In this context, the FIFO write may fail for two reasons, namely:

1. The administration component is not running,

2. The administration component is runing but for some reason is not reading the FIFO which fills up.

In order to distinguish these, the administration component will generate a PID file in the /tmp system directory. If writing to the FIFO fails and the PID file exists, then this is a genuine error situation. If writing to the FIFO fails and the PID file does not exist, then the administration component is assumed to be stopped.

(Note, the administration component will have inbuilt signal traps to prevent the PID file existing without the administration component running.)

The tool will be OSF-Motif based with appropriate menus and windows for the display of events.

### 4.7.1 Communications from the Origin and Target Applications to the Administration Component

When origin and target processes are created, they will have no direct connection to the administration component. Therefore, all origin and target processes will communicate with the administration component through the UNIX System V FIFO inter-process communication mechanism. FIFOS (which are in fact named pipes) can be written to by any process which has the correct permission, irrespective of its relationship to other processes.

The following code should be used by all origin and target processes which wish to send data to the administration component - a structure of type FIFOMESSAGE should be created then sent to the administration component by calling write_to_fifo(), passing in as a parameter a FIFOMESSAGE structure. write_to_fifo() can be called at any time by one or multiple processes, the system fifo management ensures that no data is lost.

The default FIFO from which the administration component will be **/tmp/dbvfifo**

The following code and FIFOMESSAGE structure are a suitable example of this mechanism. It is the exact format for reading by the administration component:

```
/*    Description: FIFO Interface definition
 */

#define FIFONAME  /tmp/dbvfifo
#define DBV_ADMIN_STRING_LEN 1000
#define FIFO_MESSAGE_SIZE 8 * DBV_ADMIN_STRING_LEN + 4 longs

/* Enumerated types of message which can be sent
 * from processes sent to the administration tool via a fifo
 */

enum {
        FM_origin_process_id,       /* origin notifies process id */
        FM_target_process_id,       /* target notifies process id */
        FM_user_command,            /* command entered by origin user */
        FM_origin_user_response,    /* Response given back to user */
        FM_error,                   /* origin/target error message */
        FM_originuser_id,           /* local user connected to origin */
        FM_targetuser_id,           /* Id of user connected to target */
        FM_account_of_target,       /* origin notifying target account used */
        FM_process_complete          /* the origin/target process has finished */
};

typedef struct {
        long process_id;
        long messagetype;
        char uid[DBV_ADMIN_STRING_LEN];
        char request[DBV_ADMIN_STRING_LEN];
        char response[DBV_ADMIN_STRING_LEN];
        char error[DBV_ADMIN_STRING_LEN];
        char origin_user_id[DBV_ADMIN_STRING_LEN];
        char target_user_id[DBV_ADMIN_STRING_LEN];
        char target_host_name_used[DBV_ADMIN_STRING_LEN];
        char target_host_account_used[DBV_ADMIN_STRING_LEN];
} FIFOMESSAGE;


int write_to_fifo(FIFOMESSAGE m)
```

```
{
        char    send_string[FIFO_MESSAGE_SIZE];
        int     fifofd = -1;

        memset(send_string, 0, sizeof(send_string));
        sprintf(send_string,%ld@@@@%ld@@@@%s@@@@%s@@@@%s@@@@%s@@@@%s@@@@%s
@@@@%s@@@@%s",
                m.process_id,
                m.messagetype,
                m.uid,
                m.request,
                m.response,
                m.error,
                m.origin_user_id,
                m.target_user_id,
                m.target_host_name_used,
                m.target_host_account_used);

        fifofd = open(FIFONAME, O_WRONLY);
        write(fifofd, send_string, MESSAGE_SIZE);
}
```

Note that the sequnce of @@@@ symbols are purely a field delimitor pattern.

Below is a list of the message types, their meaning and usage:

- **FM_origin_process_id** - The message is used to relay the process id of a newly created origin process. The `FIFOMESSAGE` field `process_id` is associated with this message type.

- **FM_target_process_id** - Used to relay the process id of a newly created target process. The `FIFOMESSAGE` field `process_id` is associated with this message type.

- **FM_user_command** - Used to relay the commands entered by the local user so that they can be recorded (in the local host format). The `FIFOMESSAGE` fields `process_id`, uid and request are associated with this message type.

- **FM_origin_user_response** - Used to relay the responses which the origin has received from the remote target so that they can be recorded (in the local host format). The `FIFOMESSAGE` fields `process_id` and `response` are associated with this message type.

- **FM_error** - Used to relay error messages created by the origin or target application. The contents of the error messages is to be defined by the service user. The `FIFOMESSAGE` fields `process_id` and `error` are associated with this message type.

- **FM_originuser_id** - This relays the user id of the user associated with an origin process. The `FIFOMESSAGE` fields `process_id` and `origin_user` are associated with this message type.

- **FM_targetuser_id** - This relays the user id of the user associated with a target process (a remote user accessing the local host). The `FIFOMESSAGE` fields `process_id` and `target_user` are associated with this message type.

- **FM_account_of_target** - Used to relay the account being used by the origin on the target machine. This message also includes the address of the target machine. The `FIFOMESSAGE` fields `process_id`, `target_host_name_used` and `target_host_account_used` are associated with this message type.

- **FM_process_complete** - Signals to the administration tool that the origin or target process has finished is closing down. This message should be sent in the last few instructions before the program exits. The `FIFOMESSAGE` fields `process_id` is associated with this message type

Notes:

Some of the messages need not be relayed to the administration tool. However, the messages FM_origin_process_id, FM_target_process_id and FM_process_complete messages must be used if the tool is to keep track of origin and target processes and function correctly.

For most message types, when the process id is displayed by the administration component the associated user id will be automatically displayed if it has been received.

### 4.7.2 Administration Component Errors

The administration component itself will incorporate signal trap error detection for unrecoverable program faults. If any of these are detected, all target programs running will be killed, reaped and diagnostic log files will be written.

# 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 slave 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 slave 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

It is without doubt, that in development of UNIX system software, the majority of program errors are 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.

Such routines are not usually available on the latest UNIX implementations, indeed they have been completely removed from Sun Solaris-2 (System V). If such routines do exist they should always be used.

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 DBV-OSI II prototype 1, prototype 2 and final deliverable software, test programs for testing the origin and target API functions will be developed. The programs will illustrate typical programmer use of the API including connection and query to known V2 Z39.50 servers such as those running in certain US Institutes, Universities, and any V3 Z39.50 servers (if and when these become available). These programs will be available to consortium members.

### 6.1.1 Prototype 1

The menu components for prototype 1 will be very similar to the following:

Origin Menu:

1) Specify Target Internet Address
2) Connect To Target
3) Send InitializeRequest to Target
4) Exit

Target Menu:

1) Send InitializeResponse (True) to Origin
2) Send InitializeResponse (False) to Origin
3) Exit

Demonstration of prototype 1 will use a designated origin (client) program and target (server) program. Only the TCP/IP stack will be supported. Althougth the descriptions below refer specifically to target and origin machines, these may physically be the same computer. The prototype will be demonstrated by the following sequence of user operations:

1) Z39.50 target master server program started on the target machine,

   <Target master server starts and waits for a connection from the origin>

2) Z39.50 origin program started on the origin machine,

   <The origin starts and waits for a menu selection from the user>

3) Option 1 from origin menu program is chosen and the target master server Internet address and port numbers are entered,

<Origin menu is re-displayed>

4) Option 2 from origin menu program chosen,

<A TCP/IP connection to the target master server is attempted, when the connection is successful, the target application is created by the target master server and the origin menu is re-displayed. When started, the target application automatically waits for an incoming APDU.>

5) Option 3 from origin menu chosen,

<The InitializeRequest APDU containing well formed default values is sent from the origin to the target application. The origin waits for a response from the target application which accepts the request APDU data sent by the origin, displays it and then produces the target menu>

6) Option 1 from target menu chosen,

<The InitializeResponse APDU containing well formed values including a TRUE result field is sent from the target slave process to the origin. The target application will now wait for the next request from the origin. The origin accepts the response PDU data sent by the target slave process, displays it and then produces the origin menu>

7) Option 4 from origin menu chosen,

<Origin process exits, the TCP/IP connection is lost by the target application and the target application closes>

The above sequence (1 - 7) will be repeated. In step 6, option 2 from the target menu will be chosen to send back an InitialiseResponse PDU containing a FALSE result field. The sequence must be repeated from the beginning since the service user part or Prototype 1 does not allow for two InitializeRequest PDU's to be sent during one association.

To demonstrate multi-user functionality, the tests above will be repeated using two origin/clients running simultaneously, each in a separate virtual terminal, both connecting to the same target daemon. The architecture developed for the target will allow for this with no further code modifications.

### 6.1.2 Prototype 2

The objectives of prototype 2 will be to demonstrate basic use of a medium set of developed API functions allowing TCP/IP connectivity and Z39.50 PDU transfer between an Origin and Target. This prototype will implement Z39.50 InitializeRequest-InitializeResponse, SearchRequest-

SearchResponse and PresentRequest-PresentResponse PDU transfer and reception. Prototype 2 will be a logical development extension of prototype 1.

Prototype 2 will involve enhancements of the following software programs developed for prototype 1:

a) the simple origin (client) application program with simple API function testing;

b) the simple target (server) application test program with simple API function testing;

c) the DBV-OSI II API libraries containing a relevant set of functions; and

d) the test tool which implements comprehensive API function testing.

The prototype 2 Origin program will be a further developed version of the Origin developed for prototype 1. In addition to those functions listed in the prototype 1 Origin description, the prototype 2 Origin will call the following additional functions:

| | | |
|---|---|---|
| TCP/IP Association Functions: | None | |
| SR-/Z39.50 Functions: | SearchRequest() | Send SearchRequest to Target |
| | PresentRequest() | Send PresentRequest to Target |
| Utility Functions: | None | |

The menu component of the prototype 2 client will be very similar to the following:

1) Specify Target Internet Address
2) Connect To Target
3) Send InitializeRequest to Target
4) Send SearchRequest to Target
5) Send PresentRequest to Target
6) Exit

New Option 4 will attempt to send a well formed SearchRequest PDU (with default values) to the most recently specified Target. The SearchRequest PDU smallSetUpperBound and largeSetUpperBound fields will be set to specify no record retrieval (which will be tested through the PresentRequest PDU). Send errors will be trapped and reported. When a PDU has been sent, the Origin program will wait and receive data until a complete response has been returned by the target slave program. If the response is a SearchResponse PDU, the fields returned by the

target slave program will be verified and if valid displayed on the terminal. Any errors will be trapped and reported. After this display the menu will be re-displayed.

New Option 5 will attempt to send a well formed PresentRequest PDU (with default values) to the most recently specified target. The PresentRequest PDU resultSetStartPoint and numberOfRecordsRequested fields will be set to retrieve all records found by the most recent searchRequest. Send errors will be trapped and reported. When a PDU has been sent, the Origin program will wait and receive data until a complete response has been returned by the target slave. If the response is a PresentResponse PDU, the fields returned by the target slave will be verified and if valid displayed on the terminal. Any errors will be trapped and reported. After this display the menu will be re-displayed.

New Option 6 will exit the Origin program and will also shut-down by port closure any target slave process to which the Origin is connected.

The prototype 2 target slave program will be a further developed version of the target slave developed for prototype 1. In addition to those functions listed in the prototype 1 target slave description, the prototype 2 target slave will call the following additional functions:

| | | |
|---|---|---|
| TCP/IP Association Functions: | None | |
| SR-/Z39.50 Functions: | SearchResponse() | Send SearchResponse to Origin |
| | PresentResponse() | Send PresentResponse to Origin |
| Utility Functions: | None | |

The proposed architecture for the prototype 2 target slave program will be identical to the architecture used for the prototype 1 target slave.

When a target slave is started, it will wait for and receive data until a complete Z39.50 request has been received. If the received request is an InitializeRequest PDU, the fields in the received PDU will be verified and then displayed to the virtual terminal. Any errors will be trapped and reported. If a valid InitializeRequest is received, a menu very similar to that below will be presented:

1) Send InitializeResponse Accept to connected Origin
2) Send InitializeResponse Reject to connected Origin
3) Send SearchResponse to connected Origin
4) Send PresentResponse (UnstructuredText) to connected Origin
5) Send PresentResponse (OpacRecord) to connected Origin
6) Exit

New option 3 will send a well formed SearchResponse PDU containing test fields to the connected Origin. Immediately after the PDU has been sent, the target slave process will wait for another request from the connected Origin.

New option 4 will send a well formed PresentResponse PDU containing test unstructuredText External retrieval records. Immediately after the PDU has been sent, the target slave process will wait for another request from the connected Origin.

New option 5 will send a well formed PresentResponse PDU containing test OpacRecord External retrieval records. Immediately after the PDU has been sent, the target slave process will wait for another request from the connected Origin.

New option 6 will exit the target slave process which will also cause the connected Origin program to exit because of the port close. Note that this process exit will not effect the target server master process which will be free to accept further connections from more Origins (client).

The prototype 2 API will be an enhanced and expanded version of the UNIX library archive (.a) file developed for prototype 1. The library archive will be linked with the client and server programs at compilation time. The following additional functions will be implemented:

| | | |
|---|---|---|
| Origin TCP/IP Association Functions: | None | |
| Server TCP/IP Association Functions: | None | |
| Origin SR/Z39.50 Functions: | SearchRequest() | Send SearchRequest to target |
| | PresentRequest() | Send PresentRequest to target |
| Server SR-/Z39.50 Functions: | SearchResponse() | Send SearchResponse to origin |
| | PresentResponse() | Send SearchResponse to origin |
| Utility Functions: | None | |

## 6.2 API Test Tools

A comprehensive set of test tools will be developed to prove and verify the API functions. These tools will be available to consortium members.

After development and testing of the prototype origin and target programs, a first version of the DBV-OSI II Test Tools will be created and then used to test the developed API functions rigorously.

The test tool will be a dynamically linked executable written in GNU ANSI C which will allow 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 (client) testing application to allow testing of origin related API functions with both valid and invalid parameter combinations,

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

### 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-PDU functions only:

3) 'Request At Target To' files - these files will be created by the target upon reception of a PDU from the origin and will contain the PDU 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-PDU functions only:

3) 'Request At Origin To' files - these files will be created by the origin upon reception of a PDU from the target and will contain the PDU 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 Satellites International Ltd. The SIL 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>

There may be a variable number of lines in each tbl file. Each tbl file may have a list of parameters such that the function is called more than once (for a number of associations for example).

An example of part of a tbl file structure used in a project developed by Satellites International Ltd is reproduced below to show the use of mixed constant, string, numeric and comment types.

```
1, "AU=", a, a, "Z39.50", 67, "QUESTEL", 23;    /* Author */
2, "CS=", a, a, "Z39.50", 2, "QUESTEL", 27;     /* Corporate source */
3, "CC=", a, a, "Z39.50", 20, "QUESTEL", 25;    /* Classification Code */
4, "CN=", a, a, "Z39.50", 1000, "QUESTEL", 1000;
5, "JN=", a, a, "Z39.50", 1000, "QUESTEL", 1000;
6, "DT=", a, a, "Z39.50", 95, "QUESTEL", 30;    /* Document type */
7, "BN=", a, a, "Z39.50", 7, "QUESTEL", 1000;
8, "SN=", a, a, "Z39.50", 8, "QUESTEL", 41;     /* ISSN Source */
```

### 6.2.3 Test Tool Operation

Before the test tools can be used correctly master servers for TCP and OSI connections must be started and configured to launch the test tool application called **target** in the `release/testtools/` directory.

### 6.2.4 Origin Test Mode

After the origin test tool has been selected the menu below will appear:

```
Choose Origin API Function Group To Test:
1) Association Handling
2) SR-/Z39.50 And Communication Utility Functions
3) Non Communication Utility Functions
4) Exit Program
Choice (1/2/3/4) :
```

Option 1 will produce menu 'Choose Origin Association Handling Function'.

Option 2 will produce menu 'Choose Origin SR-/Z39.50 / Communication Utility Function'.

Option 3 will produce menu 'Choose Origin Non Communication Utility Function'.

Option 4 will exit the program.

The menus:

'Choose Origin Association Handling Function',

'Choose Origin SR-/Z39.50 / Communication Utility Function' and

'Choose Origin Non Communication Function'

are shown below. In all menus and all sub-menus, if a function chosen to test has not been implemented, an error message will be output and the menu will be re-displayed.

```
Choose Origin Association Handling Function:
1) Valid AssociateRequest()
2) Invalid AssociateRequest()
3) Valid ConfirmAssociateRequest()
4) Invalid ConfirmAssociateRequest()
5) Valid ReleaseRequest()
6) Invalid ReleaseRequest()
7) Valid ConfirmReleaseRequest()
8) Invalid ConfirmReleaseRequest()
9) Valid AbortRequest()
10) Invalid AbortRequest()
11) Previous Menu
Choice (1/2/3/4/5/6/7/8/9/10/11) :
```

Each option will read from the appropriate .tbl filenames. For example, AssociateRequest() will read from the file assocreq_valid.tbl. The first menu option displayed will be the following:

1) Valid AssociateRequest()

Input From: assocreq_valid.tbl

Result To : assocreq_valid_result.tbl

Options 1-10 inclusive from this menu will execute the related function, reading parameters for the function from the related 'Input From' file. File names including the text "valid" will contain valid parameters, file names including the text _"invalid" will contain invalid parameters. By varying the contents of the input files, a wide range of tests will be possible.

The functions in this group are concerned with origin-target session communication aspects, not data transfer. Consequently, no data will be stored or retained at the target end related to execution of any of these functions. Function status and result information will be written to each relevant 'Result To' file. In all cases, any existing result files will be overwritten.

If any errors occur during origin-target association connection they will be immediately reported, the server and client processes will be destroyed and menu 'Choose Test Mode' will be re-displayed.

Option 11 will reproduce menu 'Choose Origin API Function Group To Test'.

```
Choose Origin SR-/Z39.50 / Communication Utility Function:
SR-/Z39.50 Functions
1) Valid InitializeRequest()
2) Invalid InitializeRequest()
3) Valid SearchRequest()
4) Invalid SearchRequest()
5) Valid PresentRequest()
6) Invalid PresentRequest()
7) Valid DeleteResultSetRequest()
8) Invalid DeleteResultSetRequest()
9) Valid ScanRequest()
10) Invalid ScanRequest()
11) Valid ResourceReportRequest()
12) Invalid ResourceReportRequest()
13) Valid SortRequest()
14) Invalid SortRequest()
15) Valid ExtendedServicesRequest()
16) Invalid ExtendedServicesRequest()
17) Valid CloseRequest()
18) Invalid CloseRequest()
19) Valid SIGetEventLocation()
20) Invalid SIGetEventLocation()
21) Valid ReceiveDataOrigin()
22) Invalid ReceiveDataOrigin()
23) Previous Menu
Choice (1/2/3/4/5/6/7/8/9/10/11/12/13/14/15/16/17/18/19/20/21/22/23):
```

Options 1-22 inclusive from this menu will execute the related function, reading parameters for the function from the related 'Input From' file. File names ending _valid will contain valid parameters; file names ending _invalid will contain invalid parameters . By varying the contents of the input files, a wide range of tests will be possible. Function execution status and result information will be written to each relevant 'Result To' file.

For each valid PDU related parameter function test chosen, relevant PDU data transfer to the target will occur. The user will then choose a combination of the communication related utility functions from the menu (19 - 22, typically, 19 followed by 21) in order to receive a response from the target.

On reception of a request, the target will write the received PDU data and related information to the associated 'Request At Target to' file. If the response is a segment request which by definition is the first of multiple presentation requests, the origin will perform multiple block and read

sequences using the utility communication functions until the entire request sequence has been received. Each PDU read will be appended to the segment file.

Depending on how it is configured, the target will either automatically or after user-interaction send back a well formed response PDU containing values read from the 'Target Response From' tbl file. When received, the PDU data together with related information will be written by the origin to the associated 'Response From Target To' file.

When the test tool is first started the result files will be cleared, but while the origin test tool is being executed all API result files will be appended to. This will allow the complete results of an origin test run to be viewed. The same system will also apply to the target test tool.

Option 23 will reproduce menu 'Choose Origin API Function Group To Test'.

```
Choose Origin Non Communication Utility Function:
1) Valid SISetACSEFile()
2) Invalid SISetACSEFile()
3) Valid SIWhichStack()
4) Invalid SIWhichStack()
5) Valid SISetTraceParameter()
6) Invalid SISetTraceParameter()
7) Previous Menu
```

Options 1-6 inclusive from this menu will execute the related function, reading parameters for the function from the related 'Input From' file. File names ending _valid will contain valid parameters, file names ending _invalid will contain invalid parameters. By varying the contents of the input files, a wide range of tests will be possible. Option 7 will reproduce menu 'Choose Origin API Function Group To Test'.

### 6.2.5 Origin Test Mode Test Sequence

Testing of the origin API functions will be done in the following sequence:

1) All functions tested with valid parameters.
2) All function tested with invalid parameters.
3) Sequence combinations tested including basic PDU order testing.

### 6.2.6 Target Test Mode

The target processes are started by a call to the AssociateRequest() function in the origin test tool.. After the target has been started the user will be given the option `debug (y/n)`. Selecting `y` will start the Gnu gdb debugger.

The menu below will then be presented:

```
Choose Interactive Mode:
1) Non-Interactive Mode
2) Interactive Mode
3) Exit
Choice (1/2/3) :
```

Selecting 1 will cause the process to initialise itself correctly and wait for incoming PDUs. When a PDU is received, it will automatically reply with a valid response.

If the interactive mode is chosen, the menu below will be presented allowing selection of a target API function group for testing:

```
Choose Target API Function Group To Test:
1) Association Handling
2) Non Communication Utility Functions
3) Previous Menu
Choice (1/2/3) :
```

Option 1 will produce menu 'Choose Target Association Handling Function'.

Option 2 will produce menu 'Choose Target Non Communication Utility Function'.

Option 3 will destroy the background server process then re-display menu 'Choose Test Mode'

Menu 'Choose Target Association Handling Function' is shown below. In both the menu and all sub-menus, if a function chosen to test has not been implemented, an error message will be output and the menu will be re-displayed.

```
Choose Target Association Handling Function:
1) Valid ReceiveAssociateRequest()
2) Invalid ReceiveAssociateRequest()
3) Valid AssociateResponse()
4) Invalid AssociateResponse()
5) Valid ReleaseResponse()
6) Invalid ReleaseResponse()
7) Valid AbortRequest()
8) Invalid AbortRequest()
9) PDU Reception From Client
10) Previous Menu
Choice (1/2/3/4/5/6/7/8/9/10) :
```

Options 1-8 inclusive from this menu will execute the related function, reading parameters for the function from the related 'Input From' file. File names including _"valid" will contain valid

parameters, file names including _"invalid" will contain invalid parameters. By varying the contents of the input files, a wide range of tests will be possible.

The first association handling functions always tested when in target test mode will be either Valid TargetInitialize() or Invalid TargetInitialize(), followed by Valid ReceiveAssociateRequest() or Invalid ReceiveAssociateRequest().

The functions in this group are concerned with origin-target session communication aspects, not data transfer. Consequently, no data will be stored or retained at the origin end related to execution of any of these functions. Function status and result information will be written to each relevant 'Result To' file. In all cases, any existing result files will be overwritten.

Option 9 will block the target and cause it to wait until a request PDU arrives from the origin. This option will only be valid and useful if a valid association has been previously set up between origin and target.

If a PDU arrives from the origin, it will be output to a relevant file then menu 'Choose Target SR-/Z39.50 Function' which is shown below will be displayed to allow a PDU response to be returned. The name of the file to which the received PDU will be written is shown in the following table. Note how the file names correlate to those in the related origin menu 'Choose Origin SR-/Z39.50 Function'.

| **PDU Received** | **File To Which PDU Is Written** |
|---|---|
| InitializeRequest | initreq_valid_received.<tcpip \| ositcpip \| osi> |
| SearchRequest | srchreq_valid_received.<tcpip \| ositcpip \| osi> |
| PresentRequest | presreq_valid_received.<tcpip \| ositcpip \| osi> |
| DeleteResultSetRequest | delreq_valid_received.<tcpip \| ositcpip \| osi> |
| ScanRequest | scanreq_valid_received.<tcpip \| ositcpip \| osi> |
| ResourceReportRequest | resreq_valid_received.<tcpip \| ositcpip \| osi> |
| SortRequest | sortreq_valid_received.<tcpip \| ositcpip \| osi> |
| ExtendedServicesRequest | extreq_valid_received.<tcpip \| ositcpip \| osi> |
| CloseRequest | closereq_valid_received.<tcpip \| ositcpip \| osi> |

Option 10 will reproduce menu 'Choose Target API Function Group To Test'.

```
Choose Target SR-/Z39.50 Function:
Request PDU Just Received From Origin:    xxxxxxxxxxxxxxxxxxxxxxxx
File Written Containing PDU Data:         xxxxxxxxxxxxxxxxxxxxxxxx
1) Valid InitializeResponse()
2) Invalid InitializeResponse()
3) Valid SearchResponse()
4) Invalid SearchResponse()
5) Valid PresentResponse()
6) Invalid PresentResponse()
7) Valid DeleteResultSetResponse()
8) Invalid DeleteResultSetResponse()
9) Valid ScanResponse()
10) Invalid ScanResponse()
```

```
 11) Valid ResourceReportResponse()
 12) Invalid ResourceReportResponse()
 13) Valid SortResponse()
 14) Invalid SortResponse()
 15) Valid SegmentResponse()
 16) Invalid SegmentResponse()
 17) Valid ExtendedServicesResponse()
 18) Invalid ExtendedServicesResponse()
 19) Valid CloseResponse()
 20) Invalid CloseResponse()
 21) Valid SIGetEventLocation()
 22) Invalid SIGetEventLocation()
 23) Valid ReceiveDataTarget()
 24) Invalid ReceiveDataTarget()
 25) Previous Menu
Choice (1/2/3/4/5/6/7/8/9/10/11/12/13/14/15/16/17/18/19/20/21/22/23/24/25):
```

Options 1-20 inclusive from this menu will execute the related function, reading parameters for the function from the related 'Input From' file. File names including _"valid" will contain valid parameters, file names "including" _invalid will contain invalid parameters. By varying the contents of the input files, a wide range of tests will be possible.

Function execution status and result information will be written to each relevant 'Result To' file.

For each valid parameter function test chosen, relevant PDU data transfer to the origin will occur. The user will then select a combination of utility functions from menu options 21 to 24 (typically 21 followed by 23) and if a valid sequence the target will then wait for the next request from the origin for the time specified in the tbl file. On reception of a request, the menu will be re-displayed.

While the target test tool(s) is being executed all API result files will be appended to. This will allow the complete results of a target test run to be viewed. If multiple targets are being executed, the result files will be given a unique tag to distinguish them. The filename tag which refers to a particular target test tool instance will be displayed to the user at startup.

Option 25 will reproduce menu 'Choose Target API Function Group To Test'.

```
Choose Target Non-Communication Utility Function:
1) Valid TargetInitialize()
2) Invalid TargetInitialize()
3) Valid SISetACSEFile()
4) Invalid SISetACSEFile()
5) Valid SIWhichStack()
6) Invalid SIWhichStack()
7) Valid SISetTraceParameter()
8) Invalid SISetTraceParameter()
9) Previous Menu
```

Options 1-8 inclusive from this menu will execute the related function, reading parameters for the function from the related 'Input From' file. File names including _"valid" should read in valid

parameters, file names including_"invalid" should read in invalid parameters. By varying the contents of the input files, a wide range of tests will be possible.

Option 1 (TargetInitialize()) should be the first function called by the target test tool. Note that there is no associated tbl input file for a Valid TargetInitialize() as it is not possible to know what the command line arguments would be.

Option 9 will reproduce menu 'Choose Target API Function Group To Test'.

# 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 motiviation 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.

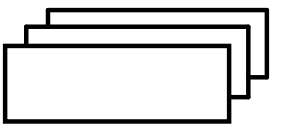
Colbert (OOSDM) Notation
Object Interaction Diagram (OID)

Objects

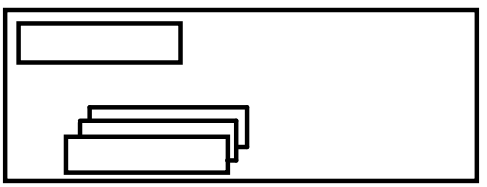
passive object

external object

active object

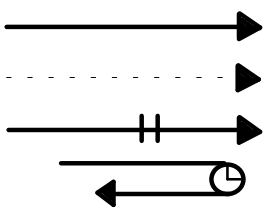


collection of objects



nested objects

Service Calls



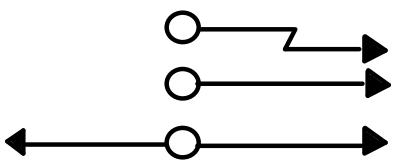
simple internal synchronous service calls (like function calls)

simple external synchronous service calls

asynchronous service calls (interrupts, mailbox, etc)

timed synchronous calls (handshakes, etc)

Data Flows

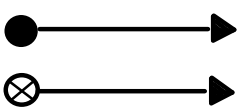


error flow

in/out flow

in and out flow

Object Flows



Grant Access

Revoke Access