Introduction

Welcome to DXSock 3.0 documentation volume 1. This book aims to help you get ready to start developing Internet or intranet servers with the latest version of the DXSock suite. In this introduction, we introduce socket technology in general and talk about prerequisite design goals that can help you prepare for developing Internet or intranet servers.

These books help you understand and appreciate the subjects and materials you need to develop servers. The books are aimed strictly at software engineers. They do not teach you programming or how to implement individual protocols. Instead, we (the authors) present and dissect the questions and problems that you're likely to encounter developing servers. We share with you how you can resolve these questions and problems lie using the DXSock suite as your ally.

We start by addressing the foundation –- discussing the basic concepts of what is the sockets, and what is the client server model. This prepares you for the primary subject of this book the DXSock Winsock Application Programming Interface (DXSock Winsock API).

Introduction – Sockets [1]

Basic concepts

The basic building block for TCP/IP and UDP/IP communications is the socket, to which a name may be bound. Each socket in use has a type and an associated process.

Sockets are typed according to the communication properties visible to a user. Applications are presumed to communicate only between sockets of the same type, although there is nothing that prevents communication between sockets of different types should the underlying communication protocols support this.

Two types of sockets currently are available to a user. A stream socket provides for the bi-directional, reliable, sequenced, and unduplicated flow of data without record boundaries. And the other is a datagram socket, which supports bi-directional flow of data, which is not promised to be sequenced, reliable, or unduplicated. That is, a process receiving messages on a datagram socket may find messages duplicated, and, possibly, in an order different from the order in which it was sent. An important characteristic of a datagram socket is that record boundaries in data are preserved.

Client-server model

The most commonly used paradigm in constructing distributed applications is the client/server model. In this scheme client applications request services from a server application. This implies an asymmetry in establishing communication between the client and server.

The client and server require a well-known set of conventions before service may be rendered (and accepted). This set of conventions comprises a protocol that must be implemented at both ends of a connection. Depending on the situation, the protocol may be symmetric or asymmetric. In a symmetric protocol, either side may play the master or slave roles. In an asymmetric protocol, one side is recognized as the master, with the other as the slave. An example of a symmetric protocol is the TELNET protocol used in the Internet for remote terminal emulation. An example of an asymmetric protocol is the Internet file transfer protocol, FTP. No matter whether the specific protocol used in obtaining a service is symmetric or asymmetric, when accessing a service there is a "client process'' and a "server process''.

A server application normally listens at a well-known address for requests. That is, the server process remains dormant until a connection is requested by a client's connection to the server's address. At such a time the server process "wakes up'' and services the client, performing whatever appropriate actions the client requests of it. While connection-based services are the norm, some services are based on the use of datagram sockets.

Introduction – Design prerequisites

Before you can start developing a server you must have an outline of what you wish to achieve. The outline becomes your road map for your project, giving you the necessary information of where you're going and how to get there. DXSock helps you get there by providing components that encapsulate basic protocols, threading, simplified I/O, and a collection of thread-safe add-ons.

Your outline should first address what type of data your server will be required to handle. For example, will your server be processing data character-by-character, line-by-line, variable length data, or combinations of these? Once you know this information you are ready to document the primary form of communication. Will the server communicate with the client asymmetrically by using a form of request/reply protocol, or symmetrically by using a form of assumption/interruption protocol.

Once you have documented the type of data and the form of communications that you will be using you are ready to start detailing the actual application requirements.

Will your server be stand-alone or embedded into an existing application?

Will your server be GUI, console, service or other?

Will your server need to communicate with other servers?

Will data need to be shared across active client sessions?

If there's a database involved, does it require a single instance per client session?

Will your server need to handle a fixed number of clients sessions?

Will your server need to log some for all of the client session?

Those are the most common questions that you will need to ask yourself before you can finish developing your project outline. In most cases, you'll be able to use one of the existing protocol implementations for your server. DXSock provides you with the ability to modify any of these protocols without changing a single line in the original source code. This is achieved through an internal dynamic parser – which allows you to introduce new commands and their associated callback hooks.

Now you are ready to start learning how to use the communications layer.

DXSock Winsock Application Programming Interface

DXSock implements a common set of communication commands in an object called TDXSock. TDXSock is used to handle everything from sending and receiving of data to accepting or making new connections and of course error handling.

First, we will concentrate on how to send and receive data. TDXSock provides a wide range of methods for handling practically any type of data. Sending data is the easiest of these routines, you simply supply the data and the routine lets you know if it was successful or not.

```
function Write(c: Char): Integer; overload;
function Write(const s: string): Integer; overload;
function Write(buf: Pointer; len: Integer): Integer; overload;
function WriteLn(const s: string): Integer;
function WriteInteger(const n: integer): integer;
function WriteResultCode(const Code: Integer; const Rslt: string): Integer;
function WriteWithSize(S: string): Boolean;
function SendBuf(const Buf; Count: Integer): Integer;
function SendFrom(Stream: TStream): Boolean; overload;
function SendFrom(var Handle: Integer): boolean; overload;
function SendFrom(var Handle: file): boolean; overload;
function SendFromStreamWithSize(Stream: TStream): Boolean;
```

Based upon what type of data you need to send or how you wish to send the data, those 12 routines are what you will need to learn to deliver the data to the client. The philosophy of the result codes is to return the number of characters successfully sent. However, when the size of the source data may be either unknown or potentially different the result codes return true if all the data was successfully sent to the client or false if there was some form of error. In the next chapter, we will address each of these routines individually along with error handling.

DXSock Winsock Application Programming Interface (continued)

```
function Read(buf: Pointer; len, Timeout: DWord): Integer; overload;
function Read(Timeout: DWord): Char; overload;
function ReadInteger(const Timeout: DWord): integer;
function ReadStr(MaxLength: Integer; Timeout: DWord): string;
function ReadString(MaxLength: Integer; Timeout: DWord): string;
function ReadLn(Timeout: DWord): string;
function ReadCRLF(Timeout: DWord): string;
function ReadToAnyDelimiter(Timeout: DWord;Delimiter:String): string;
function ReadNull(Timeout: DWord): string;
function ReadSpace(Timeout: DWord): string;
function ReadWithSize: string;
function ReceiveBuf(var Buf; Count: Integer): Integer;
function SaveTo(Stream: TStream; Timeout: DWord): Boolean; overload;
function SaveTo(var Handle: Integer; Timeout: DWord): boolean; overload;
function SaveTo(var Handle: file; Timeout: DWord): boolean; overload;
function SaveToStreamWithSize(Stream: TStream; Timeout: DWord): Boolean;
function GetChar: Str1;
function GetByte: Byte;
```

As you can see there are many different ways, you can receive your data from the client. The philosophy of the result codes is to return the number of characters successfully received. However, if the routine is commonly used for a line-by-line protocol the result is the actual line of data. Alternatively, if you are using the SaveTo routines, which are the counterparts to the SendFrom routines, then the result is true if successful or false if there was some form of error. In the next chapter, we will address each of these routines individually along with error handling.

Documentation footnotes

[1] The above material was derived from the document "An Advanced 4.3BSD Interprocess Communication Tutorial" by Samuel J. Leffler, Robert S. Fabry, William N. Joy, Phil Lapsley, Steve Miller, and Chris Torek.