Similarly, the second phrase in document 10, "Internet...", allows the user to access document 14 by link 20. In other words, if the user clicks on the HyperText phrase "Internet" document 14, or a portion of document 14, is displayed on the user's screen. HyperText documents can have nested linking. Document 12 is the target of a HyperText link and, itself, have HyperText links within it as shown in FIG. 3 where link 18 is associated with HyperText "Internet" in document 12. It refers that the same word or phrase may function as a link in different documents and different documents may link to a common document. Many variations on HyperText linking from the above shown in FIG. 3 are possible.

C. Overview of Sockets API and Secure Sockets Layer (SSL) Interface

A current embodiment of the invention provides a security protocol layered beneath an application protocol used by an application program to communicate over a network. The security protocol is implemented through a "Secure Sockets Layer" library (the SSL library) which is bound to the application program. The SSL library modules emulate the widely known "sockets" application program interface (API). The sockets API is supported by most major operating systems including UNIX and Microsoft Windows. For a general discussion of the sockets interface, refer to, *Inter-networking with TCP/IP*, volume 1, by Douglas E. Comer, Prentice-Hall, Inc. 1995, pages 335-364.

The SSL library establishes a sockets connection with an application running on a remote computer and then performs a security handshake. Once the security handshake is complete, the SSL library then encrypts and decrypts all data sent and received from a remote host computer through the socket connection. The SSL library is used with a reliable transport protocol. For the UNIX Windows environments, this is commonly provided by TCP/IP. The sockets API can also be used to provide access to Xerox XNS, Novell SPX/IPX and the OSI protocols as well.

The sockets API, typically serves as an interface used by applications to communicate with the TCP/IP protocol stack. Generally, the client and server programs each invoke operating system functions that set up an association between them referred to as a sockets connection. The client and server applications then invoke operating system functions to send and receive information between them over a network, such as the Internet, in a similar manner to calling functions to perform ordinary input/output. The information, for example, may include graphs, data, instructions and even computer programs. The sockets connection between the two programs uses data structures which allow access to TCP/IP services. The sockets API, ordinarily provides a series of system calls that application programs can invoke to request connection services.

More specifically, the typical approach to using sockets is that a server application creates an open socket ready to accept connections at a given IP address and (well known) port. Once such a socket has been created, buffer space can be allocated to store incoming connection requests. The server socket ordinarily behaves as a passive endpoint waiting for a connection request to arrive from a client. In order for a client to send a connection request to a passive socket, the socket must have a name. Therefore, names are bound to sockets but the reverse is not true; a name can address the named socket. To initiate a connection to a remote socket a client application ordinarily requests a connection and specifies a local (client) socket and a remote name as parameters. Multiple clients can use a single server socket. Client requests can be queued in a buffer associated with the server socket. Typically when a request is accepted from the queue, a new socket is created and the new socket is used as the server side connection endpoint with a client. In this manner, a single well known port number can be used to establish many simultaneous socket connections. A shutdown (or close) request usually is employed to terminate a socket connection.

Thus, in a present embodiment of the invention, an application layer program makes sockets calls to its SSL library which sets up a sockets connection and also ensures security during data transmission over the network. There are important advantages in binding a connection interface (sockets API) and a security protocol (described below) in a layer between an application program layer and a transport layer in accordance with the invention. For example, since the sockets API is so widely used by application programs and operating systems, the SSL's sockets API can be adopted for use in a broad range of applications environments. Moreover, since the SSL library of the presently preferred embodiment of the invention, is disposed between the application program layer and the transport layer, security can be provided to multiple different types of applications without significant modification to the applications themselves. Furthermore, changes in security requirements or procedures often can be more readily implemented by changing security protocols disposed above the transport layer, which is typically implemented as part of the operating system, than by changing the operating system itself. As explained above, the SSL library is layered between the application layer programs and the operating system transport control protocol stack. As a result, SSL security can be modified and upgraded without changing the operating system.

D. SSL Library's Security: Key Exchange, Authentication and Integrity Checks

This section explains the apparatus and methods whereby a client application and a server application ensure adequate security during an information exchange between them. Subsequent sections explain the details of a novel "Handshake Protocol Specification" used during SSL communications between client and server and the details of a novel "Handshake Protocol" used to establish a secure "sockets" layer (SSL) communication channel between client and server. The disclosure in this section references public key algorithms, bulk ciphers, authentication processes and integrity checks in general. It will be appreciated that the principles discussed in this section can be applied to numerous different specific instances of each of these security and integrity check mechanisms. The section below describing the handshake protocol provides many examples of different combinations of public key algorithms, bulk ciphers, authentication processes and integrity checks that can be employed consistent with the invention.

RSA Key Exchange Assuming No "Session-Identifier"

Referring to FIG. 4, there is shown the message flow during handshake protocol negotiation where RSA key exchange is employed and no "session-identifier" is stored in server cache. Note that at the stage at which the message exchange in FIG. 4 begins, the server and client have already established a "sockets" connection between them, and the server has determined that the connection is to be a secure connection and that it will employ the novel SSL processes and program control mechanisms described herein. As explained above, the sockets API is well known. Moreover, in accordance with the invention, the sockets connection is initiated...