CHAPTER 3

A WEB-ENABLED INTERFACE FOR REMOTE APPLICATION ACCESS

This chapter describes a client-server system that enables users to access remote software applications from a Java-enabled Web browser. The targeted browser is Netscape Communicator™ 4.5. Section 3.1 provides an overview of the client-server system developed for providing Web-enabled application access. Section 3.2 describes the "NetCAD" server system, which consists of a set of sub-servers for network accessible applications. Section 3.3 describes the "NetCAD" clients, which are signed applets that are downloaded over the Internet and that serve as clients for the NetCAD system. These applets require a plug-in "patch" for the Netscape Communicator 4.5 browser, which is discussed in Section 3.4. Section 3.5 describes how all these pieces fit together to form a generic network-centric application access system.

3.1 NetCAD: A JAVA BASED CLIENT-SERVER SYSTEM FOR APPLICATION ACCESS

One approach to devising a network-centric interface for remote application access is the client-server model. Such a system has been developed in support of this thesis, and it is called "NetCAD."

In NetCAD, the clients are instances of the NetCAD applet, a signed applet that can be downloaded with a web page from the NetCAD website. These NetCAD clients communicate with the NetCAD server through TCP/IP socket connections. Indeed, in keeping with the "sandbox" restrictions generally imposed on an applets (Section 2.6.3),

the NetCAD applet only opens network connections with the host machine from which it was downloaded. Therefore, the server runs on the same host machine, even though the NetCAD applet is a signed applet that, in principle, can request privileges to open network connections with arbitrary hosts. This design decision was made to improve security from the user's point of view. It was felt that the user needed to be aware of the host serving a request, and thus would be suspicious of arbitrary network connections. The basic design philosophy is that the NetCAD applet deviate as little as possible from the secure "sandbox" model and always work within a well defined domain that is controlled by the user.

The NetCAD server consists of a number of sub-servers. Each sub-server listens for connections on a predetermined network port. A port is an abstraction in the computer's memory for managing and keeping track of the network connections, and each computer with an IP address has several thousand logical ports. This way, the NetCAD server can allocate a specific port for a particular application access; thus offering access to a number of applications, each on a different port and handled by a different sub-server. This is depicted in Figure 3.1.

The NetCAD applet has the knowledge of (1) the network ports on which the sub-servers offer their services, and (2) the NetCAD server's public key. To simplify key management, the private key of a single RSA key-pair is shared by all the sub-servers. Thus, the applet can communicate securely with any of the sub-servers. Once the applet has been downloaded, it communicates with a NetCAD sub-server as follows (Figure 1.2):

(1) The client contacts a particular sub-server on its port, generates a session key, encrypts the session key with the server's public key, and sends the encrypted key to the sub-server.

(2) The sub-server receives the encrypted session key and decrypts it using its private key.

(3) The client sends the input file to be processed, encrypted with the session key, to the sub-server.

(4) The sub-server receives the encrypted file and decrypts it with its copy of session key.

(5) The sub-server executes the application program using the decrypted file as the input.

(6) The sub-server encrypts the output with the session key and sends it back to the client.

(7) The client decrypts the output using the session key, and saves it to a user-specified location.

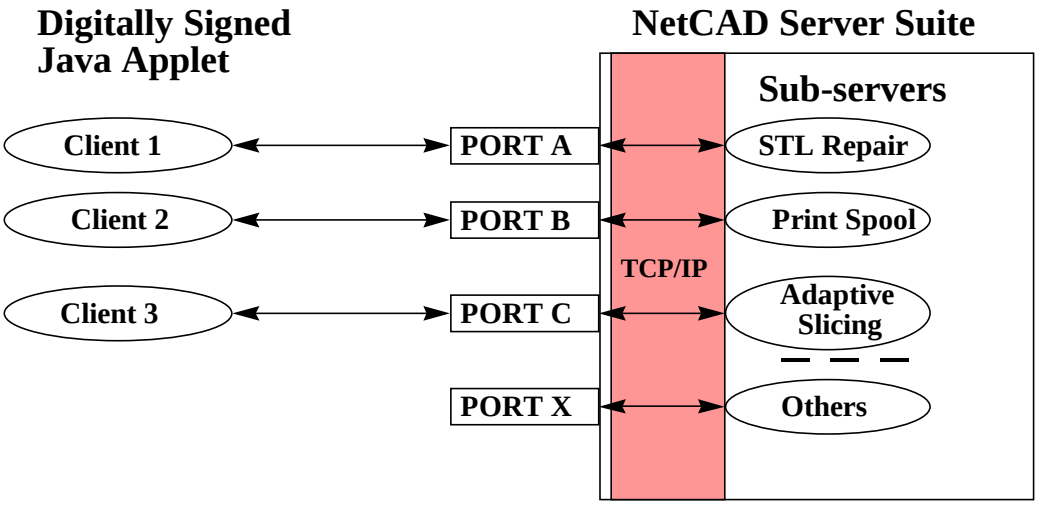


Figure 3.1 NetCAD client-server system.

In this protocol, the bulk of the data transmitted is encrypted with the session key, which is based on the DES algorithm. The exchange of the session key itself is secured

using public-key cryptography based on a RSA key-pair. Thus, the NetCAD system uses hybrid cryptography for securing the data exchange. The relative speed of the DES algorithm is utilized for the bulk of data encryption and decryption, while the slower RSA algorithm is used for session-key exchange. The client, which is an instance of the NetCAD applet, is digitally signed using public-key certificate. This signature can be used to verified the developer of the applet, and hence the client's trustworthiness, depending upon the trust that the user places on the developer.

For cryptographic operations, both the server and the applets use Cryptix™ version 3.0.3, a JCA-compliant cryptographic library in Java from Systemics, Limited. To facilitate the applet's use of the Cryptix library, a modification to the Netscape's Java security system was made. This will be discussed in detail in Sections 3.4.

3.2 NetCAD SERVER SYSTEM

The NetCAD server system consists of a set of sub-servers, each of which listen to a separate port dedicated to a specific application service. When a sub-server receives a request across its port, it creates a client handler for handling the request. The sub-server, then, returns to listening for new requests, while the client handler serves the request.

Each client handler runs on a different *thread of control* or *thread*. A thread is a section of code that is executed independently of other threads of control within a single program. Thus, one can have a single program execute a number of different tasks, each on a different thread executing independently of other threads. Such a program is called a *multi-threaded* program. This feature has been incorporated into the NetCAD sub-server's design, making it a multi-threaded server. Thus, for each request, the sub-server

creates the right type of client handler and returns to listening for new connections. Each client handler runs on a new thread, completely oblivious of other threads. When a client handler has handled its request, its thread dies and the client handler becomes free to handle a new request on a new thread. This is illustrated in Figure 3.2.

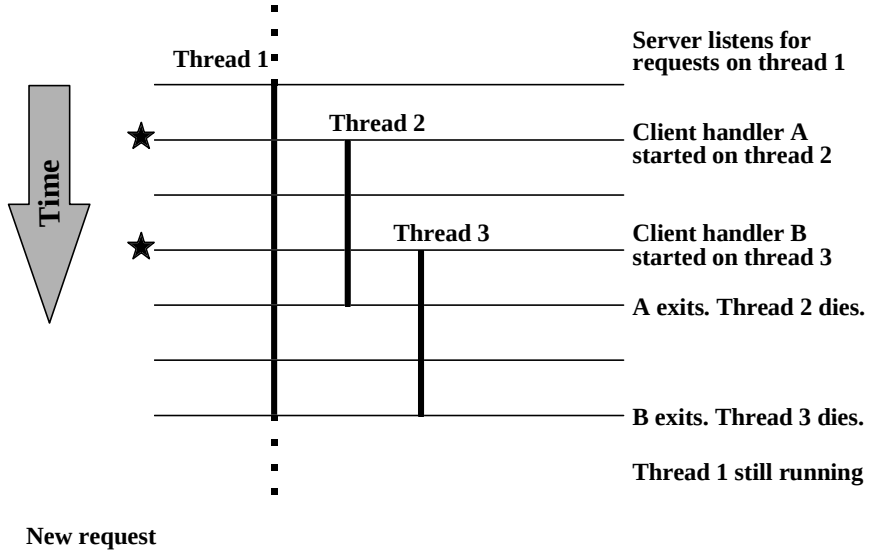


Figure 3.2 Threads in a NetCAD sub-server. The sub-server runs on the main thread, always listening for new requests. Client handlers run on new threads created by the server thread.

The NetCAD server system has been implemented as a Java application. In this code, the generic sub-server is represented by the *NetCADServer* class, which is subclassed to form the *ThreadedServer* class. The *ThreadedServer* class adds multi-threaded capability to its base class. The client handler, represented by the *ClientHandler* class, is an abstract class for handling requests from NetCAD applets. Figure 3.3 illustrates the responsibilities the *NetCADServer*, the *ThreadedServer* and the *ClientHandler* classes, and their relationship to each other.

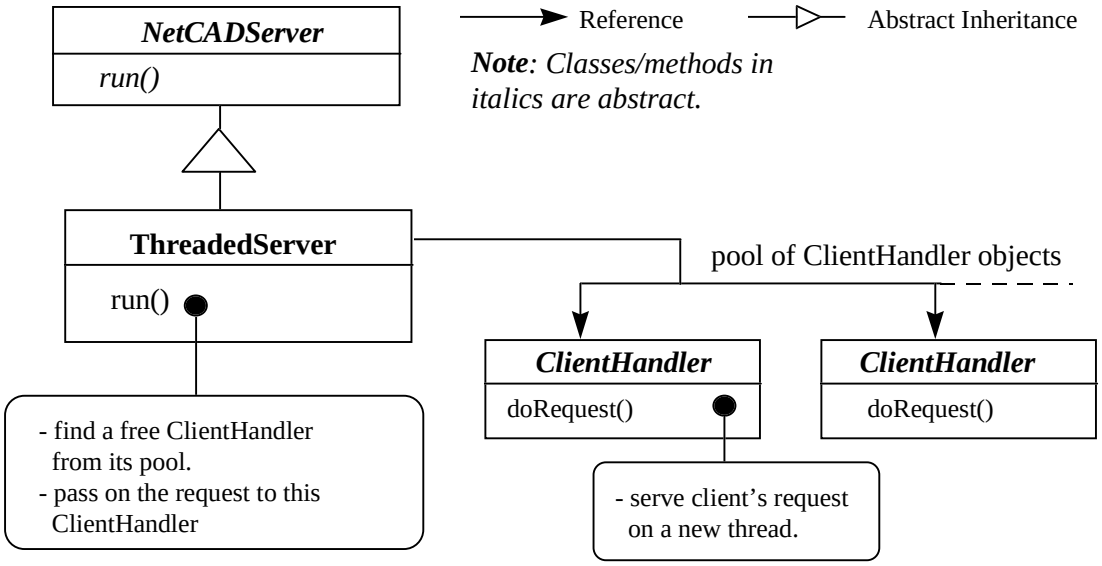


Figure 3.3 Class diagram of *NetCADServer*, *ThreadedServer* and *ClientHandler* showing their main responsibilities and their relationship to each other.

If there were no restrictions on the number of client handlers created by a sub-server, it could easily overload the operating system with too many eligible threads or too much network traffic and file I/O. Thus, each sub-server has an upper limit on the number of requests it may handle simultaneously. This limit, l_i, is set at start-up for each sub-server i. Thus, the sub-server i will create up to l_i client handlers, each running on a separate thread. Once l_i requests are being served, any additional request is queued and must wait for a client handler to become available. Each sub-server manages its own pool of client handlers by recycling existing client handlers and creating, within limits, l_i, new client handlers on demand. The algorithm for managing the client handler pool is illustrated in Figure 3.4.

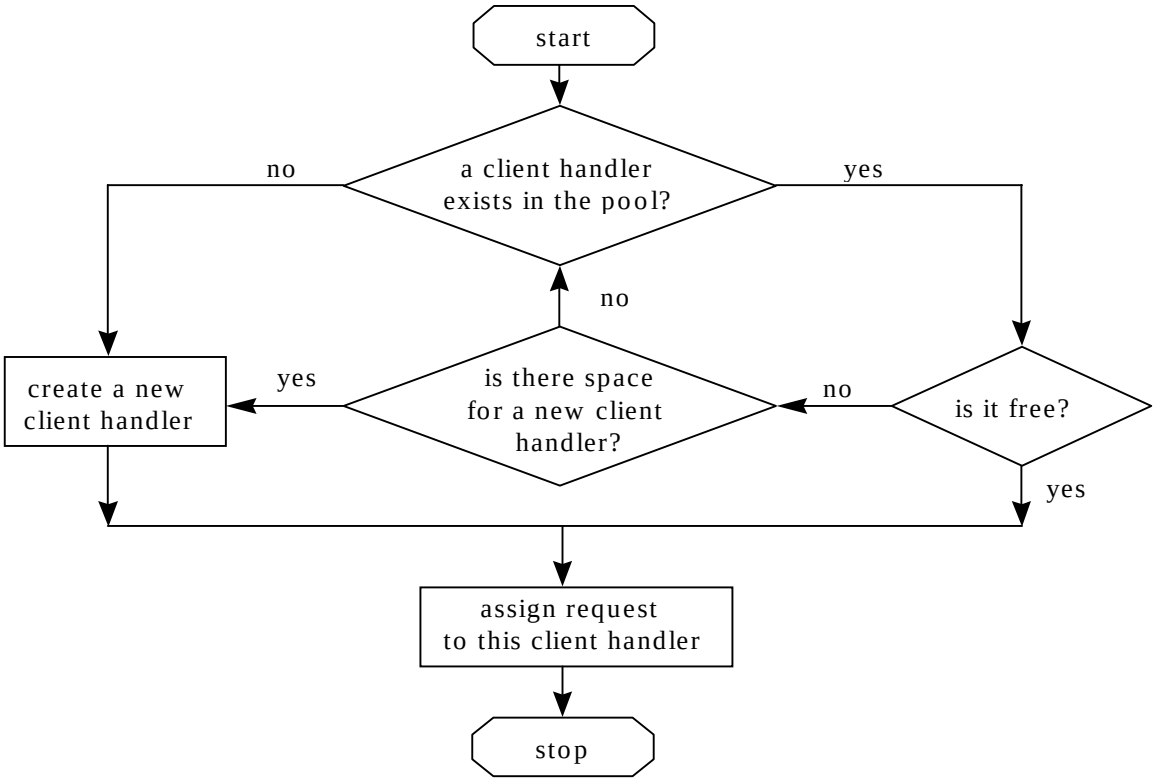


Figure 3.4 The algorithm used by NetCAD sub-servers to manage a pool of client handlers.

It is important to note that a sub-server (a *ThreadedServer* object) just listens to the network socket for incoming connections. When it receives a request, it delegates the request to a client handler (a *ClientHandler* object). However, one generic client handler cannot process efficiently all the different types of potential services. It therefore needs a way to further delegate the request to a specific client handler. This is achieved by making the *ClientHandler* an abstract class that defines the interface for handling requests and by employing the factory pattern.

The factory pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate [Gamma94]. In other words, the factory method lets a class defer instantiation to its subclasses. Therefore, a sub-server (*ThreadedServer*

object) does not create a client handler directly, but uses a *ClientHandlerFactory* interface to create a custom client handler.

The *ClientHandlerFactory* is an abstract class that must be subclassed to define a specific type of *ClientHandlerFactory* that creates the right type of *ClientHandler*. For instance, two *ClientHandlerFactory* sub-classes have been implemented as a part of this thesis research; *STLRepairFactory* and *RPSpoolFactory*, together with their respective *ClientHandler* sub-classes, *STLRepairHandler* and *RPSpoolHandler*. The first of these two sets support a .STL file repair service. Its sub-server calls the *STLRepairFactory* (*ClientHandlerFactory*) object to create a pool of *STLRepairHandler* (*ClientHandler*) objects, and delegates the service requests to the *STLRepairHandler* objects in its pool. The sub-server communicates these actions through the *ClientHandlerFactory* and *ClientHandler* interfaces. Therefore, from the sub-server's point of view, it is just managing a pool of a *ClientHandler* created by its *ClientHandlerFactory*, though it is actually managing a particular type of *ClientHandler*s determined by the type of *ClientHandlerFactory* used. A general sub-server can therefore function as a sub-server dedicated to a particular service by varying the *ClientHandlerFactory* object associated with it. This paradigm is illustrated in Figure 3.5.

The actual processing of the input file received from a client is done by the `handleRequest()` method of the concrete implementation of the *ClientHandler* class. This method calls the appropriate system-dependent pre-compiled executable to process the file. Any program that takes an input file, processes it in some way, and produces an output file, can be incorporated into the NetCAD server system.

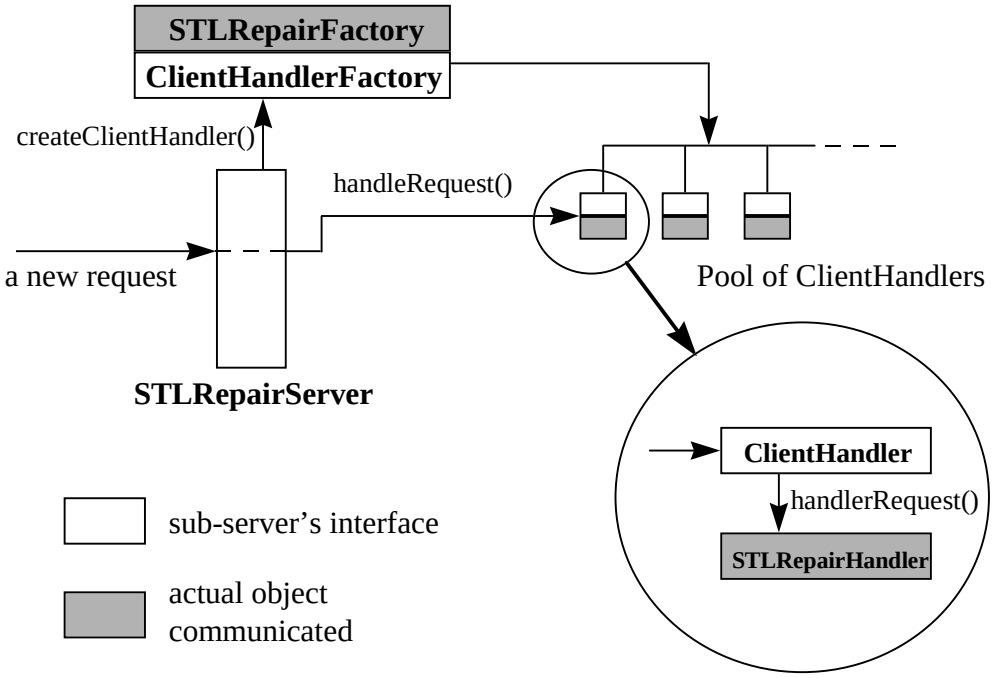


Figure 3.5 The working of the *STLRepairServer*, which is a sub-server that provides .STL file repair service. The *STLRepairServer* is a subclass of the general *ThreadedServer* class.

The sub-server is also responsible for security during data transmission. This includes (1) decrypting the session key received from the client; (2) using this session key to decrypt encrypted files received from the client; (3) checking the integrity of these files; and (4) encrypting the output file and returning it to the client. The *ClientHandler* employs a *SecurityHandler* object to perform these tasks. The *SecurityHandler* class uses methods provided by the *CryptoHandler* class to abstract the above tasks into convenient methods. The *CryptoHandler* class implements the methods defined in an interface named *CryptoInterface*. *CryptoHandler* is built upon Cryptix version 3.0.3 cryptographic library.

Within the *CryptoHandler* class, the `encryptStreamWithDES()` and the `decryptStreamWithDES()` methods work together to provide transparent integrity

check on the data they encrypt and decrypt, respectively. Their algorithms are outlined in Figures 3.6 (a) and 3.6 (b), respectively. These two methods must be used together. The data encrypted and written to an output stream by `encryptStreamWithDES()` must be processed by `decryptStreamWithDES()`. Therefore, both the sub-servers and the NetCAD applet need to have the knowledge of the *CryptoHandler* class.

(1) Create a byte array, *b*, of *L* + 20 bytes, where *L* is the length of file to be encrypted.

(2) Create a message digest of the file using Secure Hash Algorithm (SHA) algorithm. The SHA algorithm hashes any pre-image to hash value of 20 bytes.

(3) Fill the first 20 bytes of *b* with the hash value generated in step 2.

(4) Fill the next *L* bytes of *b* with the contents of the file to be encrypted.

(5) Encrypt and send the byte array *b* using the session key, in blocks of 1024 bytes, using the Cipher Feed-back (CFB) mode.

Figure 3.6 (a) Algorithm for `encryptStreamWithDES()` method.

(1) Create a byte array, *b*, of *L* + 20 bytes, where *L* is the length of file to be decrypted.

(2) Read and decrypt the incoming bytes from an input stream into the byte array, *b*, using the CFB mode.

(3) Calculate the hash value of the last *L* bytes from the byte array, *b*, using SHA.

(4) Compare this hash value with the value stored in the first 20 bytes of *b*. If the hash values match, the integrity of the received data is assumed to be verified; otherwise the data has been corrupted during transmission.

Figure 3.6 (b) Algorithm for `decryptStreamWithDES()` method.

The class diagram of the NetCAD server system and that of the different components of a sub-server (*STLRepairServer*) are illustrated in Figures 3.7 (a) and 3.7 (b), respectively. To create a new sub-server, a new subclass of *ClientHandler* and its corresponding *ClientHandlerFactory* subclass need to be created while keeping the rest of the structure unchanged. Thus, the different sub-servers differ only in the type of *ClientHandlerFactory* objects that they employ, which facilitates the easy implementation of several sub-servers within the NetCAD server system.

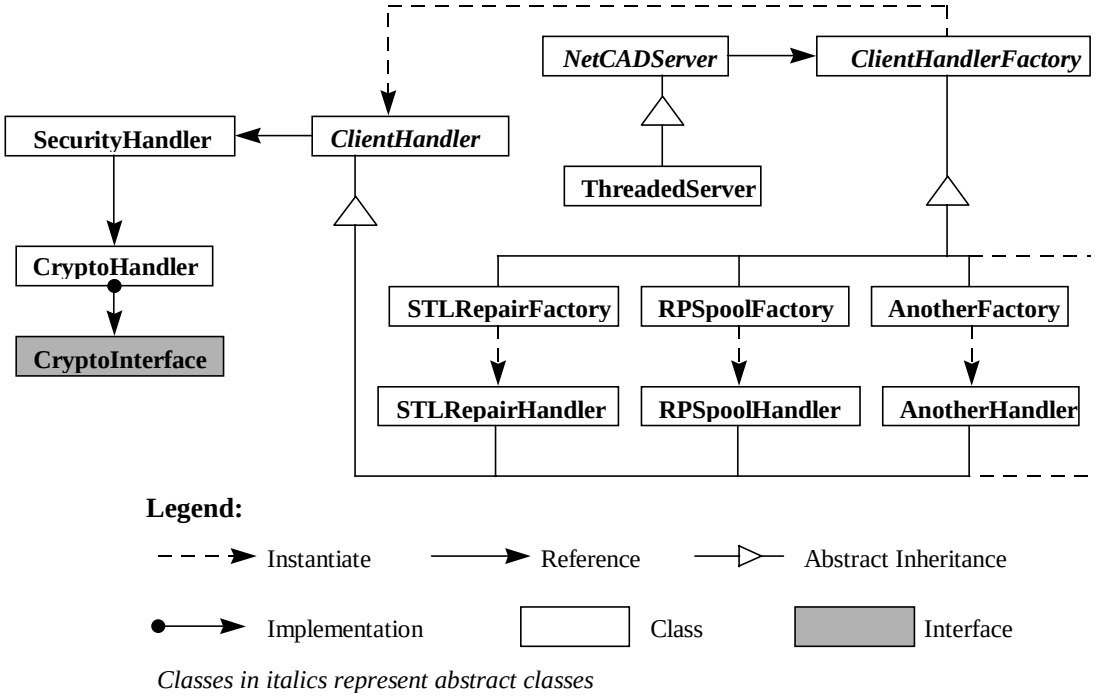


Figure 3.7 (a) Class diagram of NetCAD server structure.

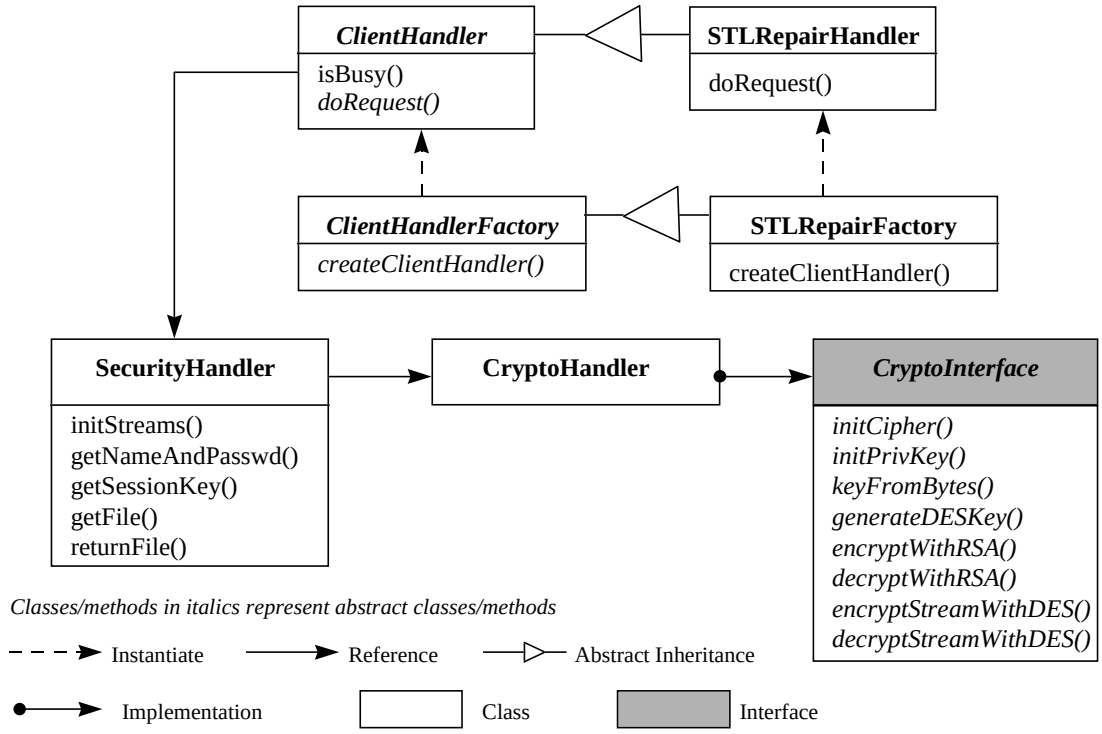


Figure 3.7 (b) Class diagram of various components of *STLRepairServer*.

3.3 NetCAD CLIENT

In the NetCAD system, the clients are instances of the NetCAD Java applet. The client opens a TCP/IP socket connection with a NetCAD sub-server running on the host machine and sends the user-specified input file to the sub-server for processing. When the file has been processed, the sub-server returns the processed file to the client, and the client saves it at a user-specified location. The client may also communicate some other information related to the service as required by the server. All this communication is secured using hybrid cryptography.

This scope of operation goes beyond the conventional “sandbox” model that applets are intended to operate within. It is therefore necessary for the user to grant the applet

privileges to access certain local resources outside the “sandbox”. One approach is to apply the basic signed-applet model, which opens access to all local resources once the user trusts the applet’s source and permits it to run. This differs from the access-control mechanism used in NetCAD applets, which is based on Netscape’s Capabilities model. In this model, a trusted, or, signed applet requests from the user only those local access privileges it needs, using the “Capabilities classes” which is a freeware Java class library that is available from Netscape Communications Corporation [NCC97].

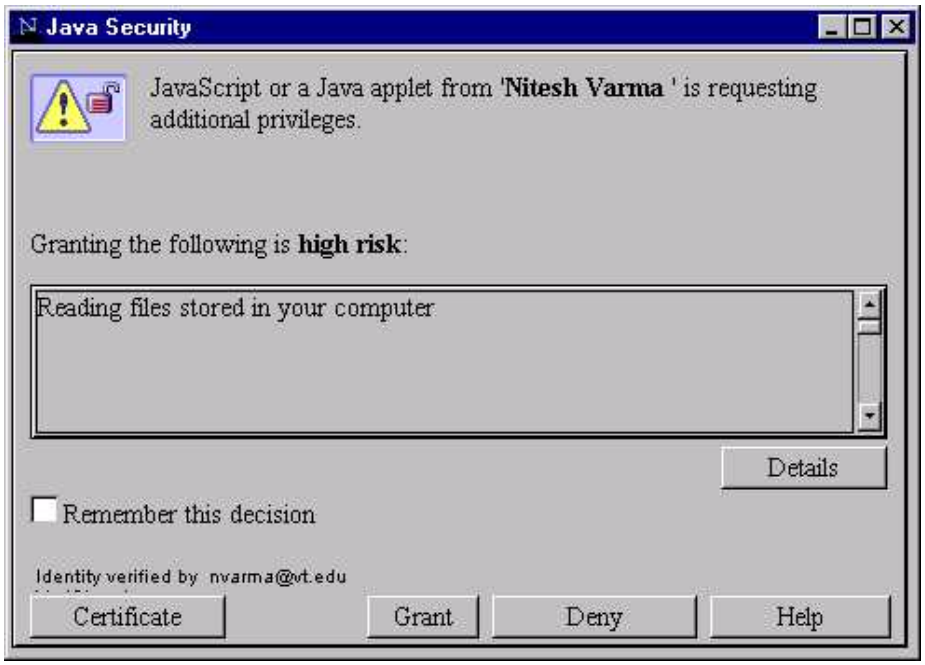
In the Capabilities model, a *principal* is the entity requesting access to a particular local resource or *target* and authorization or denial to access the requested resource represents the principal’s *privilege*. A principal is an instance of the *Principal* class and typically represents a signing certificate. A target is an instance of the *Target* class and typically represents one or more system resources, such as reading files stored on a local disk or writing to a local disk. Finally, fine-grained access control is provided by an instance of the *Privilege* class, which controls which principal may access what target and for what duration.

The applet’s code comes bundled in a digitally signed JAR file. A signed JAR file is an archive file in the standard cross-platform ZIP format that contains one or more files that are digitally signed using a signing certificate. The digital signatures of the files in the JAR is contained in three files; *manifest.mf*, *zigbert.mf*, and *zigbert.rsa*. These are archived along with the other files in the JAR and placed in a directory called META-INF. The whole JAR file is downloaded as a part of the applet. Therefore, as each of the applet’s class files is loaded by the browser’s JVM, its digital signature can be verified against the information in META-INF directory. If the applet’s code passes the digital

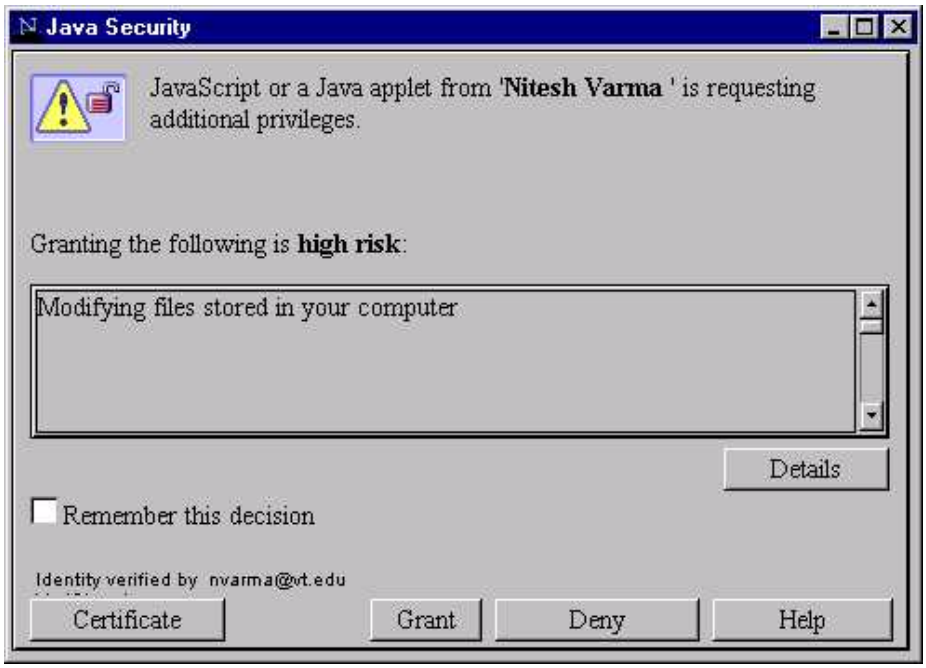
signature verification, then the user is presented with dialog boxes (Figure 3.8) that request extended privileges for the applet to access specific targets. At this point, the user can verify the principal's identity by reviewing the developer's certificate by clicking on the "Certificate" button. A sample developer's certificate is shown in Figure 3.9.

The NetCAD applet requests the following special privileges from the user: (a) the "UniversalPropertyRead" privilege, which is required to read Cryptix properties; (b) the "UniversalFileRead" privilege which is required to read the file to be processed; (c) the "UniversalFileWrite" privilege which is required to write the processed file; (d) the "AddSecurityProvider" privilege which is required to add Cryptix as the security provider for cryptographic operations; and (e) the "GetSecurityProviders" privilege, which is used to check if Cryptix is already installed as the security provider before an attempt is made to install it. The first three privileges reference Netscape Capabilities classes, while the last two reference the Cryptix library.

The applet's JAR file also contains the server's RSA public key. However, unlike the applet's class files which are verified at run-time before being loaded, the public key is not automatically verified. Therefore, to protect the public key from a man-in-the-middle attack [Schneier96], its SHA-1 message digest is hard-coded into the applet's code. Since the applet is verified when loaded, a man-in-the-middle attack on the applet is prevented (Figure 3.10), and the applet can safely compute the SHA-1 message digest of the key in its JAR file and compare the resultant hash with the one that is hard-coded. If the hashes match, it can be reasonably assumed that the public key has not been tampered with. This procedure, therefore, ensures the integrity of both the applet's code and the RSA public key while the applet is downloaded over the Internet.



(a)



(b)

Figure 3.8 Applet requesting access to local resources: (a) UniversalFileRead privilege; (b) UniversalFileWrite privilege.

Figure 3.9 The developer's signing certificate as viewed from Netscape Communicator's dialog box.

In case of a man-in-the-middle attack on the applet itself, where a malicious party tampers with the applet code, the digital signature on the applet will be broken. The applet will no longer pass the signature verification and will produce error messages on the browser's Java console window (Figure 3.10).

Once the applet's code and the server's public RSA key have been securely downloaded, the applet is responsible for: (1) generating a session key using the DES algorithm and exchanging it with the sub-server using the server's RSA public key; (2) encrypting the input file that it sends to the sub-server using the session key it generated; and (3) decrypting the output file it receives from the sub-server and verifying its integrity. The applet employs the Cryptix library to perform these tasks. As with the

NetCAD server class structure (Section 3.2), the *CryptoHandler* class is used with the only modification being that it is now stripped down to keep the size of the applet's code to a minimum while still meeting the applet's requirements. This follows the programming practice in which the size of the applet code is kept to a minimum to minimize the associated download time.

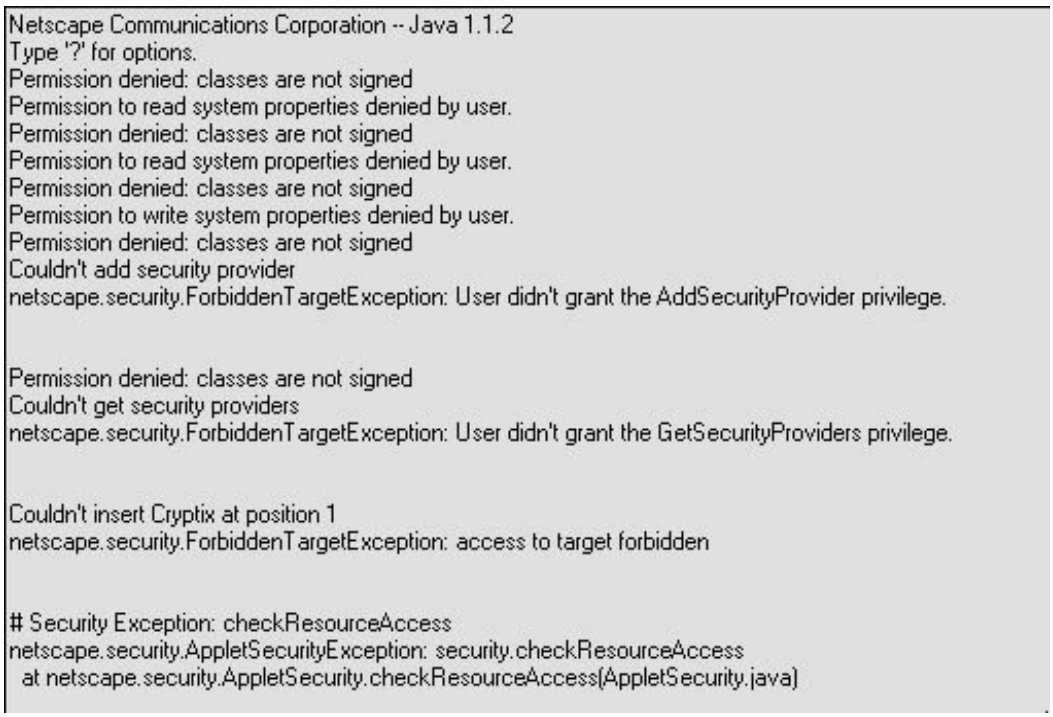


Figure 3.10 An example of the error generated in the Netscape Communicator's Java console if the browser encounters a signed applet with a broken signature.

The NetCAD applet consists of the main class called *Client* which is responsible for creating the user interface for the applet (Figure 3.11). The *Client* employs a *RequestHandler* object to manage the service request. The *RequestHandler* is an abstract class that must be subclassed for each type of service that the *Client* makes available. An example of this is the *STLRequestHandler* class which opens a TCP/IP socket connection to the *STLRepairServer* sub-server at a pre-determined port and carries out the client-side

handling of .STL file repair requests. Thus, the *RequestHandler* class provides an interface for dynamically dispatching requests to the right kind of request handler objects. The *Client* class also employs an *InfoHandler* object to record the service-specific information, such as the location of the file on the local disk and user's user-name and password. The *Client* object gathers this information through its graphical user interface and from a dialog box produced by the *PropertyDialog* class, and stores the information in its *InfoHandler* object. The *InfoHandler* object is then passed to the appropriate *RequestHandler* which subsequently employs a *CryptoHandler* object to carry out the cryptographic operations. The *CryptoHandler* class implements all the methods defined in its interface called *CryptoInterface*. Thus, the *Client* class is only concerned with creating a graphical user interface, recording the information gathered, and delegating the service request to the appropriate *RequestHandler*; while the *RequestHandler* object is responsible for communicating with its sub-server and sending the service request. Figure 3.12 illustrates the class structure of the NetCAD applet.

3.4 PATCH FOR NETSCAPE COMMUNICATOR 4.5 BROWSER

The NetCAD applets are designed to perform cryptographic operations using the Cryptix library. However, to achieve this, the following obstacles needed to be overcome: (1) Java applets cannot load over the network any Java class file which belongs to a package whose name starts with "java"; and (2) The Netscape Communicator 4.5 browser does not include *java.security* package in its JVM. This is a problem because the NetCAD applet needs the Cryptix library and *java.security* package. They cannot be included in

the applet's JAR file because Cryptix library includes classes that belong to *java.security*, *java.security.interfaces* and *java.lang* packages.

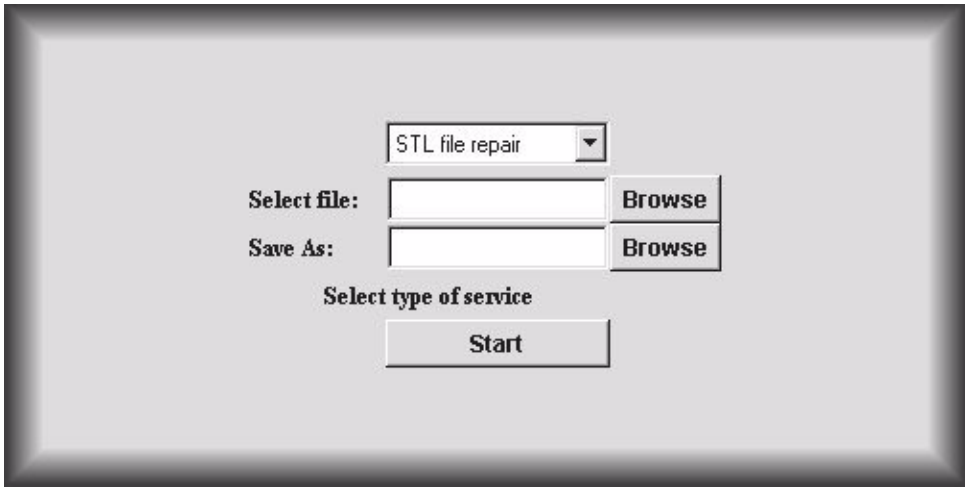


Figure 3.11 NetCAD applet's graphical user interface.

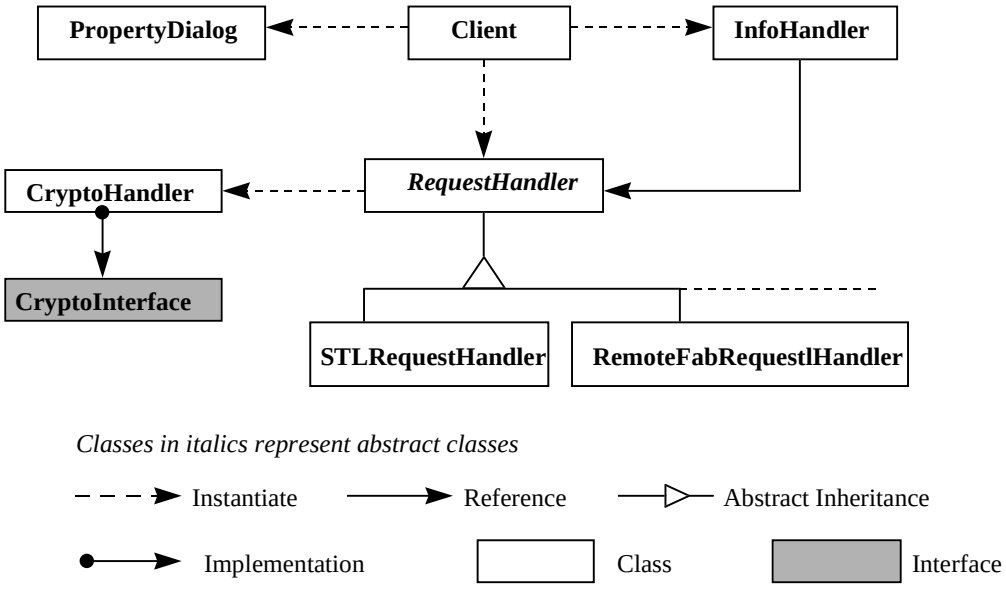


Figure 3.12 NetCAD applet's class structure.

To overcome this problem, one can install a patch for the browser in the form of a digitally signed JAR file named *Crypto_Support.jar* prior to downloading the NetCAD applet. The Crypto_Support.jar file contains all the Java packages and properties files that make up Cryptix library and the *java.security* package with the exception of a slightly modified *Security* class in *java.security* package. The content of this JAR file is digitally signed using the same signing certificate that is used to sign the applet's JAR file. It is important that these two digital signatures be the same to prevent the installation of malicious implementations of the Cryptix library or the *java.security* package. The modified *Security* class, which is located in the Crypto_Support.jar file, re-implements certain security checks using the Cryptix's *IJCE_SecuritySupport* class. This is necessary because the implementation of `checkSecurityAccess()` method of *SecurityManager* class in Netscape Communicator 4.5 always results in a security exception. Java 1.1 does not specify that such exception must always be generated, but the developers of the JVM for Netscape Communicator 4.5, and its prior releases, have chosen to do so. The modified *Security* class checks that "AddSecurityProvider" and "GetSecurityProvider" privileges are granted to the intended principal and not to a malicious one. The *Security* class compares the principal to whom these privileges were granted to its own principal. If these match, the *Security* class is satisfied that a legitimate applet is instructing it to add cryptographic support of the Cryptix library from Crypto_Support.jar file.

Digitally signing the files in Crypto_Support.jar provides security in two ways:

1. The *Security* class is able to check the principal of the applet against its own.

2. Once the principal of the applet is verified, the applet checks the principal of the Cryptix classes, at run-time, before using them. If the principal of any of the used classes is different from the principal of the applet, then a subversion of the classes in the Crypto_Support.jar file is detected and the applet warns the user about the subversion and terminates.

Thus, under this scheme, the signed applet requests essential privileges in private methods and verifies the signature (principal) of all the classes in the Crypto_Support.jar file that it uses. If one of the classes used is replaced by an adversary, then this is detected by the applet. The applet itself is verified to be from a trusted entity both by the user and the modified *Security* class in the Crypto_Support.jar file.

To install the Crypto_Support.jar file, the user needs to download it from NetCAD service website and place it in the appropriate directory location:

Windows 95/NT: [Netscape_Home]\Navigator\Program\Java\classes directory

UNIX: [Netscape_Home]/java/classes directory

Then, when the browser's JVM starts, it recognizes all Java class files under the [Netscape_Home]/.../java/classes directory, including those inside the Crypto_Support.jar file.

3.5 THE NetCAD SYSTEM

Sections 3.2, 3.3 and 3.4 described the client, the server, and the client's cryptographic support, respectively, in isolation. The following describes their mutual interaction to deliver secure services to remote clients (Figure 3.13):

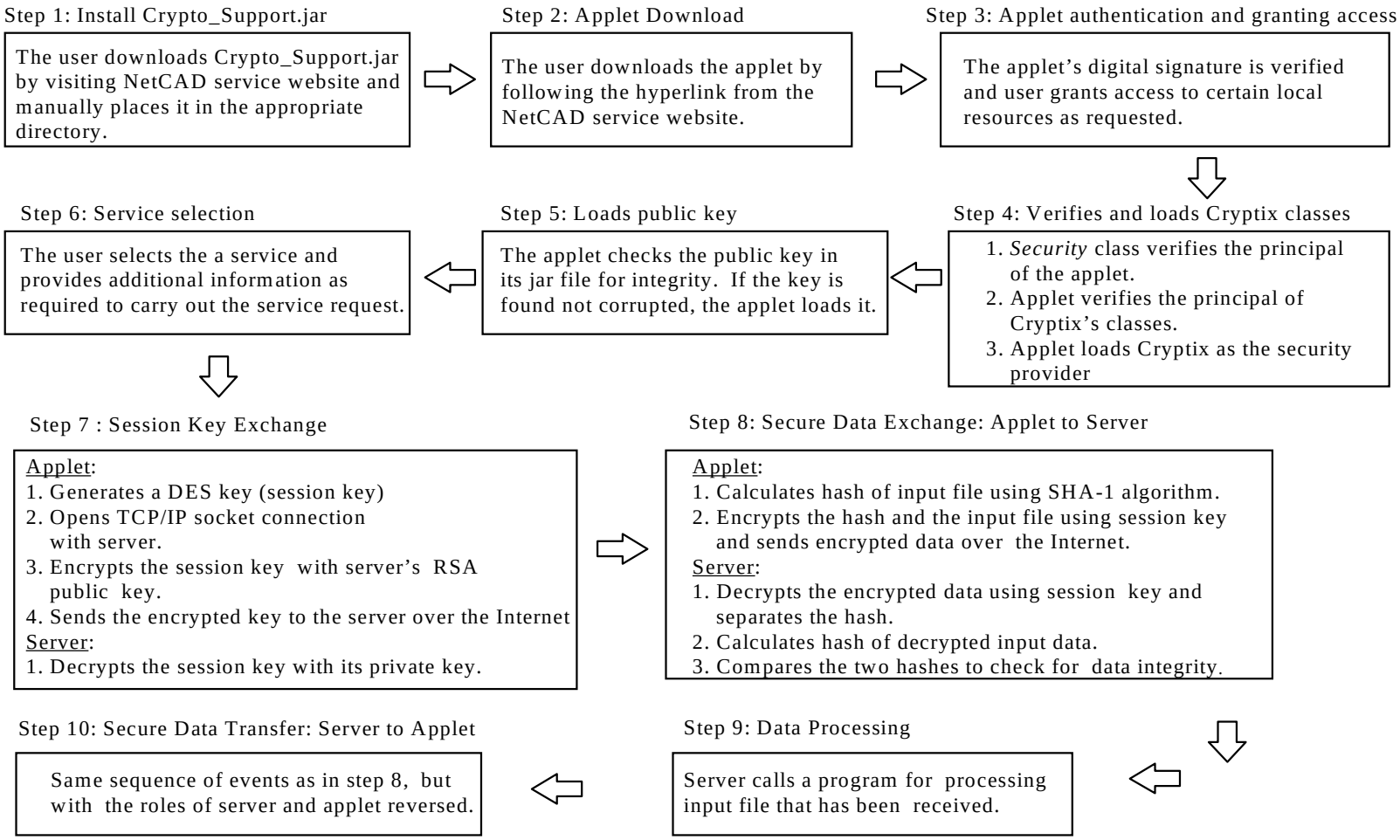


Figure 3.13 Sequence of events of a client-server interaction in the NetCAD system.

(1) The user downloads the NetCAD services web page by pointing the browser to the appropriate Universal Resource Locator (URL). From this page, the user downloads the Crypto_Support.jar file and manually installs it as described in Section 3.4.

(2) The user then follows a hyperlink to the NetCAD applet's page. Visiting this page will automatically download the NetCAD applet's JAR file.

(3) The browser's JVM loads the applet, the browser asks the users for five specific "out-of-the-sandbox" privileges for the applet on the behalf of its code-signer. These are the "UniversalFileRead," "UniversalFileWrite," "UniversalPropertyRead," "AddSecurityProvider," and "GetSecurityProviders" privileges.

(4) The user must grant all the privileges for the applet to work. Once these privileges have been granted, the applet makes Cryptix the security provider. However, before the Cryptix library, which is located in Crypto_Support.jar file, is activated, the *Security* class in the Crypto_Support.jar file compares the principal of the applet with its own principal. If the *Security* class inside the Crypto_Support.jar archive is tampered with, then its signature will be broken. This will result in an invalid principal and the signature verification failure, making the Cryptix cryptographic support unavailable. Before the applet loads the Cryptix classes, it matches their principal against its own principal to detect any subversion of one or more of the Cryptix's classes.

(5) The applet reads the server's RSA public key which is stored in its JAR file and checks its integrity. If the key passes this test, the applet loads it; otherwise the applet informs the user that the key has been corrupted and terminates.

(6) The user selects from a pull-down menu the kind of service desired, and selects the locations of the local file to be processed and where the output is to be saved. Some services will likely require additional information, for which the applet presents the user with a dialog box.

(7) The applet creates a DES key, opens a TCP/IP socket connection with the appropriate sub-server, encrypts the DES key with the server's public RSA key, and sends the encrypted key to the sub-server over the network. At the other end, the sub-server decrypts the DES key with its private key. Hence, the applet and the sub-server have securely exchanged the session key.

(8) The applet reads the specified file, encrypts it on the fly with the session key, and sends it to the sub-server. The applet also includes a message digest of the file along with the file itself. The sub-server decrypts the received data on the fly using its copy of session key, and checks the integrity of the file by computing its message digest and comparing it with the message digest that was included with the file.

(9) If the file is found to be uncorrupted, then the sub-server invokes a pre-compiled executable to processes the file and write the output to a specific location.

(10) The sub-server encrypts the output file using the session key and sends it to the applet. The applet decrypts this file and writes it to the user-specified location on the local disk. Finally, the data integrity of the output file is verified using its message digest.