The ISO/OSI Security Architecture

As well as defining their seven-layered model, the ISO/OSI group also defined a range of terminologies forming their

ISO/OSI Security Architecture. It includes the requirements (also listed in Lecture 10):

- **data confidentiality** - protects data as it traverses the network from being disclosed to incorrect parties. Even the presence of particular communication sequences between parties should not be identified.

- **data integrity** - protects the data from modification or removal while in the network,

- **data origin authentication** - validates the sender of the data,

- **data receiver authentication** - validates the receiver of the data,

- **peer-entity authentication** - validates all network components, such as hardware routers and peer software components through which a data stream must travel, and

- **non-repudiation** - creates and verifies evidence that the claimed sender sent the data, that the intended receiver did receive it, and that neither can deny that this occurred.

**NOTE:** the core TCP/IP internetworking suite meets **none** of the requirements of the ISO/OSI Security Architecture. Support for additional services is evolving, primarily at the Application Layer, but changes cannot be easily made to lower layers.
Digital certificates

Digital certificates have been loosely described as *the driver's license for the Internet*.

A digital certificate provides a *binding* between an entity's public key, and one or more attributes to its identity.

- An *entity* may be a person, a executing piece of software, or a device such as a router or a smart-card.
- A *certification authority* (CA) attests to the authenticity of the entity's public key by digitally signing a message with its own private key.
- The 'quality' of the certificate depends on the detail of information provided to the CA (more later).
- Either, public *and* private keys may be issued by the CA, or the CA may *challenge* the entity's public key.

The successful use of digital certificates appears within a large community - little is gained by issuing one's own.
Digital certificate encoding

Today, certificates are defined by the ISO X.509 protocol and appears as an application/x-x509-user-cert MIME type.

The data is encoded using Abstract Syntax Notation (ASN.1), encoding and transmitted in ASCII using base64 encoding.

(18bit data -> 24bit representation).

Early debate centred on whether the certificate itself needed to be encrypted (now not).
Browser support for digital certificates

Digital certificates are managed by all common browsers: Firefox, Safari, Opera, Netscape Navigator, Microsoft Internet Explorer ...

If visiting a site with the secure Hypertext Transport Protocol, as with https://secure.csse.uwa.edu.au/ we can view digital certificate information via the 'padlocked' icon.

Unfortunately, there are often few CAs from Australia in most common browsers.
Browser support for digital certificates

The browser will display the digital certificate from the current page - here showing:

- The subject of the certificate,
- The issuer (CA) of the certificate,
- The serial number of the certificate,
- The period of validity of the certificate, and
- The message digest of certificate.

If the issuer of a site's digital certificate is already known by the browser (either 'hard-wired' or manually added), the issuer's certificate may be viewed and verified.

Version 3 of X.509 introduced extension fields - the association of additional information with a certificate. Each extension has:

- an extension type providing semantics and typing of the extension (e.g. a string),
- an extension value", such as an email or IP address, and
- a criticality indicator indicating if the whole certificate should be ignored if an extension is not recognized.

Standard extensions (?) now describe the 'strength' and purpose of the certificate - digital signature, non-repudiation, key encipherment, data encipherment, certificate signing, etc.
Certificate Path validation

CAs are organized in hierarchies - each parent CA signs a certificate vouching for a subordinate CA's public key.

When validating a chain of certificates, the certificate path, the path is followed until the top of the chain is reached (when?).

There is no automated way of verifying the top of a certificate chain other than verifying that it is one of a list of directly known (and implicitly trusted) certificates (such as in a browser).

Several companies, such as VeriSign, Thwaite, Baltimore, AT&T, and a growing band of government departments have positioned themselves 'at the top'.

Networks and Security (CITS3002), Lecture 11, p6, 12th May 2015.
Certificate Revocation Lists

A certificate revocation list (CRL) allows clients and servers to check whether the entity they are dealing with has a valid certificate.

Trust breaks down, and CRLs are required, when:

- a subject's private key is exposed,
- a CA's private key is exposed, and
- the relationship between the subject and CA changes (e.g. the subject is no longer employed by the CA, or stops paying money to the CA).

Certificate revocation plays a crucial part in the authentication process:

- Obtain the subject's digital certificate and verify its validity.
- Extract the serial number of the certificate.
- Fetch the current CRL from the CA.
- Verify the CRL's digital signature, and record its publication time and when the next CRL is to be published.
- Examine the CRL to determine if the intended certificate been revoked or suspended (based on the certificate serial number).
- Alert the user if the certificate is revoked.

Limitations of Certificate Revocation

In a large public key infrastructure community, CRLs are both large and must be downloaded frequently.

Applications can be significantly slowed by the need to retrieve the latest CRL from a heavily taxed directory server (or other distribution point).

There exists a compromise between always being up-to-date, versus the risk of false certificate acceptance.
The SSL/TLS Protocol

The Secure Sockets Layer (SSL) is a communications protocol implementing 3 cryptographic assurances - authentication, confidentiality, and message integrity. Its primary purpose is to make e-Commerce users comfortable with exchanging financial information over the Internet.

A very brief history:

- Netscape developed SSL protocol in its browsers in 1994, and then v2.0 in 1995.
- Microsoft quickly developed and released its Private Communicating Technology (PCT) in 1995, which briefly competed with SSL. Media attention to SSL and PCT bugs in 1995 resulted in SSL v3.0, and PCT quietly disappeared.
- In 1996 the Internet Engineering Task Force (IETF) formed its Transport Layer Security (TLS) group to write the SSL standard. In Jan 1999 TLS v1.0 (almost identical to SSL v3.01) was released, as RFC-2246, and provided within major web browsers and by some standard programming language libraries, notably openssl (see www.openssl.org) and JDK 1.2 onwards.

In popular web-browsers, SSL is employed when the browser displays a locked padlock or key icon, when connections are made with the secure hyper-text transport protocol (https:), which first establishes an SSL connection between client (browser) and (web) server and then transmits standard HTTP data within the payload of SSL packets.

Some material for these notes gratefully copied from the document SSL: Foundation for Web Security.
SSL and its Relationship with TCP/IP

The SSL protocol runs above TCP/IP and below higher-level protocols such as HTTP, FTP, or IMAP (the Internet Message Access protocol, for email).

SSL uses TCP/IP on behalf of the higher-level protocols, and in the process allows:

- an SSL-enabled server to authenticate itself to an SSL-enabled client. An SSL-enabled client can use public-key cryptography to check that a server's certificate and public ID are valid and have been issued by a certificate authority (CA) listed in the client's list of trusted CAs.

- (optionally) allows the client to authenticate itself to the server. Using the same techniques as those used for server authentication, an SSL-enabled server can check that a client's certificate and public ID are valid and have been issued by a certificate authority (CA) listed in the server's list of trusted CAs.

- allows both machines to establish an encrypted connection. All information sent between client and a server is encrypted. In addition, all data sent over an encrypted SSL connection is protected against *man-in-the-middle attacks* with a mechanism for detecting tampering.
Establishing an SSL/TCP connection to a server

```c
#include <openssl/ssl.h>
#include ....

typedef struct {
    int socket;
    SSL *sslHandle;
    SSL_CTX *sslContext;
} mySSLConnection;

// Establish a connection to a server using SSL
mySSLConnection *sslConnect(const char *hostname, const char *port) {
    mySSLConnection *c = calloc(1, sizeof(mySSLConnection));
    if(c == NULL)
        return NULL;

    // Establish a standard TCP connection with the server
    c->socket = connect_to_server_using_TCP(hostname, port);
    if(c->socket >= 0) {
        // Initialize the SSL library
        SSL_load_error_strings();
        SSL_library_init();

        // Create an SSL context, specifying required protocols
        c->sslContext = SSL_CTX_new( SSLv23_client_method() );
        if(c->sslContext == NULL)
            goto bad;

        c->sslHandle = SSL_new(c->sslContext);
        if(c->sslHandle == NULL)
            goto bad;

        // Associate the standard TCP connection with the SSL context
        if(SSL_set_fd(c->sslHandle, c->socket))
            goto bad;

        // Perform the SSL handshaking between client and server
        if(SSL_connect(c->sslHandle) != 1)
            goto bad;
        return c;
    }

    bad:
        ERR_print_errors_fp(stderr);
        error("sslConnect failed");
        ssldisconnect(c);
        return NULL;
}
```

// Perform communication with:
```c
int bytes_written = SSL_write(c->sslHandle, buffer, sizeof(buffer) );
int bytes_read = SSL_read(c->sslHandle, buffer, nbytes );
```

// Disconnect from remote SSL server and free mySSLconnection
void ssldisconnect(mySSLconnection *c) {
    if(c != NULL) {
        if(c->sslHandle) {
            SSL_shutdown(c->sslHandle);
            SSL_free(c->sslHandle);
        }
        if(c->socket >= 0)
            close(c->socket);
        if(c->sslContext)
            SSL_CTX_free(c->sslContext);
        free(c);
    }
}
```
Hello and Negotiation Parameters

The first activity undertaken by SSL/TLS is the negotiation of cryptographic parameters (the building blocks) between the two end-points that probably don't know each others' abilities nor capacities.

Using the common names of Alice and Bob as the server and client software, respectively, Bob sends a plaintext hello message to Alice, and suggests parameters for the communication session. Bob may also provide alternatives:

**client Bob sends**: (not the actual encoding)

```plaintext
Version: TLSv1, SSLv3
Key Exchange: RSA, Diffie-Hellman
Secret Key Cipher Method: TripleDES, DES
Message Digest: SHA-1, MD5
Data Compression: PKZip, gzip
Random Number: 195,633,929
```

**server Alice may respond with**:

```plaintext
Version: TLSv1
Key Exchange: RSA
Secret Key Cipher Method: DES
Message Digest: SHA-1
Data Compression: PKZip
Random Number: 329,893,820
```

In combination, all of these agreements are termed the **cipher suite**.

After responding, Alice (server) also sends her digital certificate signed by a certification authority (CA); Bob (client) uses his trusted copy of the CA's public key to verify Alice's certificate. Only if Alice sends her certificate, does SSL/TLS permit her to request Bob's certificate (and Bob may choose to not provide it).
Cipher Suites provided by SSL

SSL implementations support a variety of different cryptographic algorithms, or ciphers, for use in operations such as authenticating the server and client to each other, transmitting certificates, and establishing session keys.

The choice of cipher suite depends on factors such as the SSL version, company policies regarding acceptable encryption strength, and government restrictions on export of SSL-enabled software.

As public-key encryption is typically 100-1000x slower than symmetric key encryption, SSL employs public-key encryption only for the exchange of the agreed secret key; this is then used for symmetric encryption of the data-stream.

Example cipher suites supported by the SSL protocol that use the RSA key-exchange algorithm include:

- **strong** (typically only permitted within the US)
  Triple DES (for symmetric encryption), SHA-1 message digest, 128-bit RC4 and MD5, 128-bit RC2 and MD5, DES and SHA-1.

- **exportable** (from the US)
  40-bit RC4 and MD5, 40-bit RC2 and MD5.

- **weak** (authentication and tamper detection only)
  No symmetric-key encryption and MD5.

Of note, the new Rijndael encryption algorithms, forming the new Advanced Encryption Standard (AES) to replace DES (documented in [RFC-3268](https://tools.ietf.org/html/rfc3268)) is slowly being supported.
**Key Agreement (Exchange)**

Bob (the client) next generates a 48-byte random value, termed the *pre-master secret*, encrypts it with Alice's (server's) public RSA key, and sends it to her.

Alice then decrypts the pre-master secret with her private key.

So far, there is only a single shared secret, but now both Alice and Bob generate 6 new secret keys.

Alice first generates three secret keys for Alice->Bob messages:

- a DES secret key for stream encryption,
- a key for the chosen message-integrity algorithm, and
- a key providing an initialization vector for the cipher.

Alice also generates three additional keys for Bob->Alice communication. Similarly, Bob must generate (but not transmit) 6 *identical* keys for the reverse communication.

Bob sends Alice a message encrypted with their (assumed) shared secret keys - the *finished handshake* message. This is the first message encrypted and sent with the generated keys.

Alice responds to Bob with her own encrypted finished handshake message. Bob is now assured that he is communicating with Alice, as only Alice could have decrypted the initial pre-master secret with her private key.
The SSL Record Protocol Layer

The following sequence of steps is finally used to transmit messages:

- Alice compresses a plaintext message using the agreed upon (optional) compression algorithm,
- Alice hashes the compressed data, appends her secret HMAC key, and generates a digital signature of the data,
- Alice encrypts the compressed data and its signature with her Alice→Bob DES key, and transmits the message.

- Bob decrypts the compressed data and its signature,
- Bob authenticates the message by recalculating the digest,
- and (optionally) finally decompresses the plaintext message.
Man-in-the-middle Attacks

Although public-keys need not be hidden (and hence do not need encrypting when embedded in digital certificates), they do need to be carefully stored and transmitted.

If the first connection and host key exchange between a client and a particular host is intercepted, the \textit{man-in-the-middle} attack fools both the client and server into thinking that they are communicating directly with one another when, in fact, an attacker is actually intercepting all traffic between the two.

Using the common names of Alice and Bob as 'the good guys', and Eve as the 'eavesdropper':

1. The client (Bob) initiates a connection with the server (Alice). Unknown to both Bob and Alice, an attacker (Eve) is waiting to intercept their connection negotiation.

2. Bob requests Alice's public key, confident that only Alice will have its corresponding private key. Eve sees the request, passes it through to Alice, but intercepts Alice's (genuine) reply.

3. Eve sends to Bob Eve's public key (authenticating herself as Alice). Bob uses this public key (thinking it is Alice's) to encrypt messages for Alice.

4. Eve intercepts all encrypted messages from Bob to Alice, decrypts them with Eve's private key, reads them, encrypts them with Alice's public key, and forwards them all to Alice. Alice is none the wiser.

5. Similarly, if Alice asks Bob for Bob's public key, Eve can also pass to Alice Eve's public key (in places of Bob’s genuine reply), and also decrypt-read-reencrypt all messages from Alice to Bob.
Using SSL from other socket-based software

Ideally, as the SSL protocol sits above the TCP/IP layer, it should 'simply' provide the same set of APIs as, say, the Berkeley sockets interface and all traffic would be 'automatically' encrypted.

Despite providing functions named `ssl_connect()` and `ssl_accept()`, using OpenSSL is not quite that simple.

Access to the SSL protocol is best provided through a well-trusted library such as OpenSSL. The library provides (rather complex and poorly documented) access to the SSL protocol, and many of the required cryptographic 'building blocks', such as hash functions, random-number generators, symmetric and asymmetric cryptography, and digital certificate verification.

For some tutorial-style OpenSSL examples see our Resources page.

Another approach is not to code-in SSL facilities to existing (closed-source?) programs, but to use `protocol tunneling` to provide authentication and encryption, such as `stunnel`, a free tunneling package requiring OpenSSL.

Consider the requirement to SSL-enable an IMAP server that is already running on a server. The server's command:

```
stunnel -d 130.95.1.88:imap2 -r 127.0.0.1:imap2
```

will receive incoming SSL connections from the wider-Internet and redirect (copy) all communication to the local IMAP server. We then need to convince each client machine that it provides its own mail server, and then run:

```
stunnel -c -r 130.95.1.88:imap2 -A /etc/ca_certs
```
What Are Client/Server Software Architectures?

Client/server computing is the logical extension of modular programming.

Modular programming has as its fundamental assumption that separation of a large piece of software into its constituent parts ("modules") creates the possibility for easier development and better maintainability.

Client/server computing takes this a step farther by recognizing that those modules need not all be executed within the same memory space.

With this architecture, the calling module becomes the client (that which requests a service), and the called module becomes the server (that which provides the service).

The logical extension of this is to have clients and servers running on the appropriate hardware and software platforms for their functions.

For example, database management system servers running on platforms specially designed and configured to perform queries, or file servers running on platforms with special elements for managing files.

ssh@vnet.ibm.com

For a long time it was widely-held myth that client/server computing had something to do with PCs or Unix machines. Cloud-computing, and mobile-computing, are contemporary examples demonstrating that the choice of hardware and operating system platforms has become quite irrelevant, and that interoperability through standards is all-important for success.
What Does A Client Process Do?

The client is a process that sends a message to a server process, requesting that the server perform a service.

Client programs usually manage the user-interface portion of the application, validate data entered by the user, dispatch requests to server programs, and sometimes execute business logic. The client-based process is the front-end of the application that the user sees and interacts with.

The client process often manages the local resources that the user interacts with such as the monitor, keyboard, and peripherals.

One of the key elements of a client workstation is the graphical user interface (GUI). Normally a part of operating system, i.e. the window manager detects user actions, manages the windows on the display, and displays the data in the windows.

What Does A Server Process Do?

Server programs generally receive requests from client programs, execute database retrieval and updates, manage data integrity and dispatch responses to client requests. The server-based process may run on another machine on the network. This server could be the host operating system or network file server, providing file system services and application services.

The server process often manages shared resources such as databases, printers, communication links, or high powered-processors. The server process performs the back-end tasks that are common to similar applications.

kalakota@uhura.cc.rochester.edu
What is a Two-Tier Architecture?

A two-tier architecture is one where a client talks directly to a server, with no intervening server. It is typically used in small environments (fewer than 50 simultaneous clients).

A common error in client/server development is to prototype an application in a small, two-tier environment, and then attempt to scale up by simply adding more client connections to the server.

This approach will usually result in an ineffective system, as the server becomes overwhelmed. To properly scale to hundreds or thousands of users, it is usually necessary to move to a three-tier architecture.

What is a Three-Tier Architecture?

A three-tier architecture introduces (another) server (or an agent) between the client(s) and the traditional server.

The role of the agent is manyfold. It can provide translation services (as in adapting a legacy application on a mainframe to a client/server environment), metering services (as in acting as a transaction monitor to limit the number of simultaneous requests to a given server), or intelligent agent services (as in mapping a request to a number of different servers, collating the results, and returning a single response to the client.

Lloyd.Taylor@jhuapl.edu
What is an 'Intranet'?

The explosion of the World Wide Web is due to the world-wide acceptance of a common transport (TCP/IP), server standard (HTTP), and markup language (HTML). Many corporations have discovered that these same technologies can be used for internal client/server applications with the same ease that they are used on the Internet.

Thus was born the concept of the *intranet* - the use of Internet technologies for implementing internal client/server applications.

One key advantage of Web-based intranets is that the problem of managing code on the client is greatly reduced. Assuming a standard browser on the desktop, all changes to user interface and functionality can be done by changing code on the HTTP server. Compare this with the cost of updating client code on 2,000 desktops.

A second advantage is that if the corporation is already using the Internet, no additional code needs to be licensed or installed on client desktops. To the user, the internal and external information servers appear integrated.

A rapidly-disappearing disadvantage is that there is limited ability to provide custom coding on the client. In the early days of the Web, there were limited ways of interacting with the client. The Web was essentially "read-only", with protocols such as *gopher* and *WAIS*. With the release of code tools such as Java and JavaScript, this limitation is no longer a major issue.

*Lloyd.Taylor@jhuapl.edu*
Characteristics Of Client/Server Architectures

1. A combination of a client or front-end portion that interacts with the user, and a server or back-end portion that interacts with the shared resource.

   The client process contains solution-specific logic and provides the interface between the user and the rest of the application system. The server process acts as a software engine that manages shared resources such as databases, printers, modems, or high powered processors.

2. The front-end task and back-end task have fundamentally different requirements for computing resources such as processor speeds, memory, disk speeds and capacities, and input/output devices.

3. The environment is typically heterogeneous and multivendor. The hardware platform and operating system of client and server are not usually the same. Client and server processes communicate through a well-defined set of standard application program interfaces (APIs), RPCs, and RMIs.

4. An important characteristic of client-server systems is scalability. Horizontal scaling means adding or removing client workstations with only a slight performance impact.

   Vertical scaling means migrating to a larger and faster server machine, or to multiservers.

   kalakota@uhura.cc.rochester.edu
Partitioning Client/Server Responsibilities

In moving a single, monolithic application to a separated client/server configuration, we must address a number of issues:

1. Is there a *functional* partition at all?
   
   Are there separate responsibilities that *can* be performed by separate tasks?  
   *Should* they be separated?

2. Is there a *data-driven* partition?
   
   Can different sections of the data be centralized, or split between multiple tasks?  
   Can these multiple tasks execute on separate hardware, with separate address-spaces?  
   *Should* distinct data partitions be replicated?

3. Is there an extensive use of global variables?
   
   Is there significant state information that controls the execution of the application?  
   *Is it possible, and desirable, to centralize this information anyway?*

4. Are there any hidden intra-application communication mechanisms (such as variables, exceptions, or signals)?
   
   Is there unusual, possibly asynchronous, control flow in the application (e.g. the use of global goto's, or asynchronous signals)?

---

Networks and Security (CITS3002), Lecture 11, p22, 12th May 2015.
Concurrency (and hence speed) in Servers

The primary motivation for providing concurrency in servers is, of course, speed.

Concurrency is derived from using a 'non-queuing' model of execution, either by using a new (copy of the) server to support each client, or to provide faster, 'time-sliced' response to each client.

If no concurrency is available in the server, pending requests from new and existing clients are either blocked or refused (c.f. the listen() socket API call).

In general, clients leave their concurrency to the operating system, unless the application is sufficiently large, or time-critical, that it is the only process on a CPU and it performs its own internal scheduling.

Increased concurrency, and hence speed, is required (demanded) when:

- forming responses requires significant I/O.
- processing time is proportional to the type of request.
- application-specific, high-performance, hardware is available.

Definitions:

- **iterative servers** - single request at a time.
- **concurrent servers** - multiple 'simultaneous' requests.
Iterative Servers - managing a single connection

Firstly, consider a *iterative server* - each connection, or *session*, with a distinct client is handled until its completion.

Our server process will fully service each client until its completion, and continue to service subsequent clients until told to terminate.

Case 1 - makes subsequent clients wait:

```c
// Open a socket, listen, bind an address
....
keep_going = true;

while(keep_going) {
    new_client = accept(my_socket, ....);
    if(new_client == -1) {
        perror("accept");
        keep_going = false;
    } else {
        extern bool service(int sd);
        keep_going = service(new_client);
        shutdown(new_client, SHUT_RDWR);
        close(new_client);
    }
}

shutdown(my_socket, SHUT_RDWR);
close(my_socket);
```

If the time to service each client is lengthy, or of indeterminable duration, new clients may be kept waiting for their initial connection - perhaps being terminated with an error of **ECONNREFUSED**.
Iterative Servers - managing one process per client

In this second model, a separate operating system process is spawned to handle each new client. There is thus no waiting required if any clients are slow or long-running, but (significant?) additional load is added to an operating system.

Case 2 - will service subsequent clients quicker, but cannot scale indefinitely:

```c
// Open a socket, listen, bind an address
....
bool keep_going = true;

while(keep_going) {
  new_client = accept(mysocket, ...);

  if(new_client == -1) {
    perror("accept");
    keep_going = false;
  } else {
    switch ( fork() ) {
      case -1:
        keep_going = false;
        perror("fork");
        break;

      case  0: {
        extern bool service(int sd);

        close(mysocket);
        (void)service(new_client);

        shutdown(new_client, SHUT_RDWR);
        close(new_client);
        exit(EXIT_SUCCESS);
        break;
      }
      default:
        shutdown(new_client, SHUT_RDWR);
        close(new_client);
        break;
    }
  }
}
shutdown(mysocket, SHUT_RDWR);
close(mysocket);
```

Note that while this example will work, that it is not complete. In particular, the parent process will need to (eventually) wait for the completion of its child processes, else many “zombie” processes will persist.
Concurrent Servers Using `select()`

With a concurrent server, a single server handles many clients within the same process. This obviates the need for interprocess communication between multiple servers (such as file locking).

We use a new network supporting system call, `select()`, to inform our process which descriptors are ready for I/O. The descriptors may be open to files, devices, sockets or pipes.

`select` deals with sets of descriptors (implemented as an array or bitmap in C or C++) and provides functions for their manipulation.

Here we examine descriptors open for reading, timing out each 10 seconds:

```c
#include <sys/select.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

    bool keep_going = true;
    ....
    while(keep_going) {
        fd_set readset;
        FD_CLR(&readset);
        FD_SET(sd, &readset);

        foreach other descriptor open for reading {
            FD_SET(desc, &readset);
        }

        struct timeval timeout;
        timeout.tv_sec = 10;   // wait up to 10 seconds
        timeout.tv_usec = 0;

        if(select(MAXDESC, &readset, NULL, NULL, &timeout) >= 0) {
            if(FD_ISSET(sd, &readset)) {
                client = accept(sd, ...); // yet another new client
                ....
            }
            foreach other descriptor open for reading {
                if(FD_ISSET(desc, &readset)) {
                    service(desc);
                    ....
                }
            }
        }
    }
```

Note that this example is typical in employing only one set of file descriptors, `readset`. A service much more concerned about I/O speeds, particularly disk blocking, would employ another set of file descriptors, `writeset`, or perform asynchronous file I/O.