IERG 4180
Network Software Design and Programming

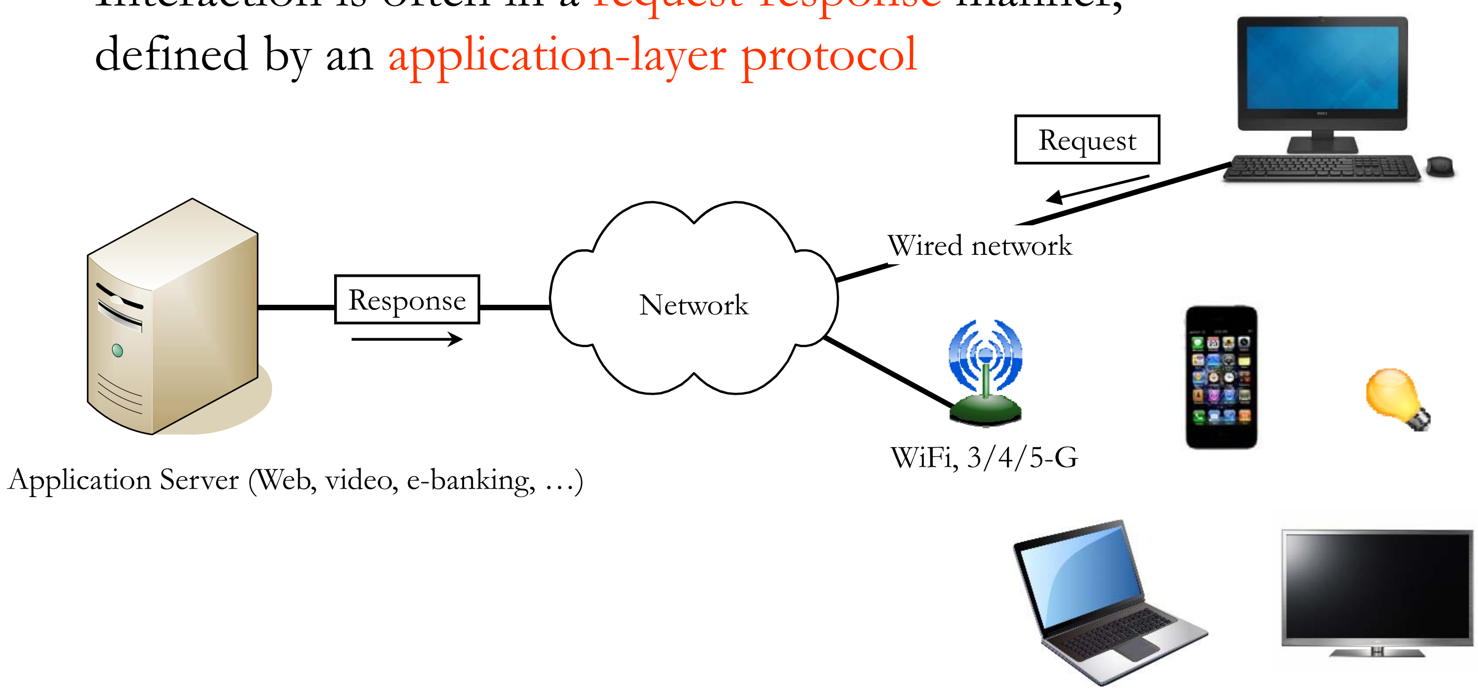
Client-Server Programming
Part I – Fundamentals

Rev. 1.1 – 28 Sep 2016

Contents

1. Client-Server Model

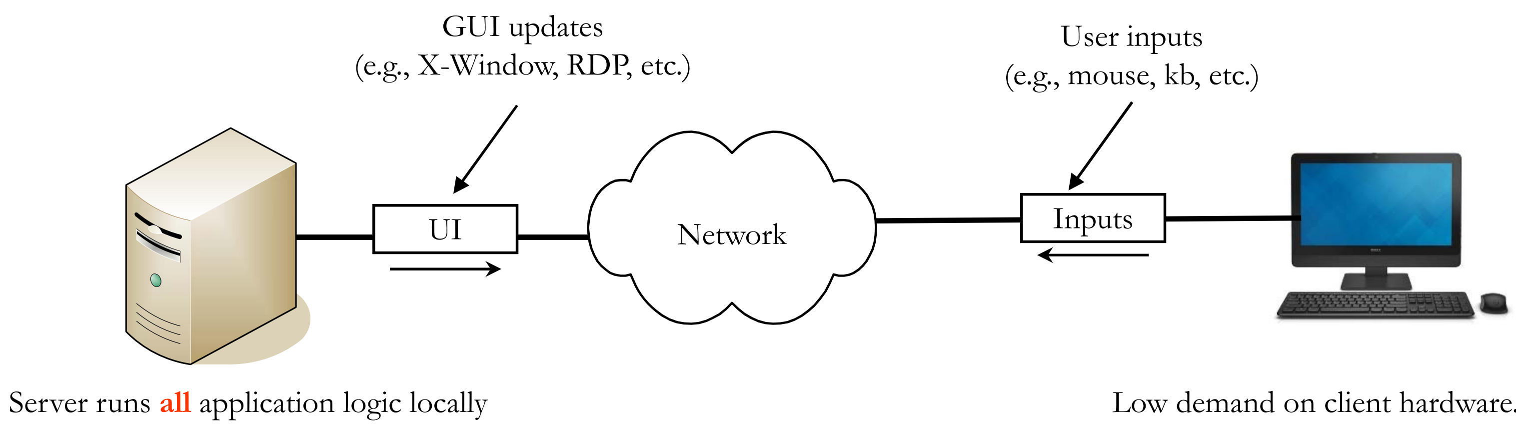
- Architecture
  - Server hardware and operating system platform choices are more flexible
  - Interaction is often in a request-response manner, defined by an application-layer protocol



Clients may run in a wide variety of platforms and connect via different kinds of networks.

- Application Server
  - Able to process requests from multiple clients simultaneously
  - Automatically runs upon operating system startup
  - Operates continuously as a long-running process
  - Listen for incoming connections/requests using a well-known transport address (e.g., http://www.cuhk.edu.hk)
  - Must fail in a predictable manner and can be restarted into a consistent state
  - Public servers are open to network intruders and attacks
  - A plan for capacity scalability cannot be an after-thought
- Application Client
  - Initiates requests to servers to request data/service
  - May perform local processing independent of server
  - May not require constant connection to server to execute

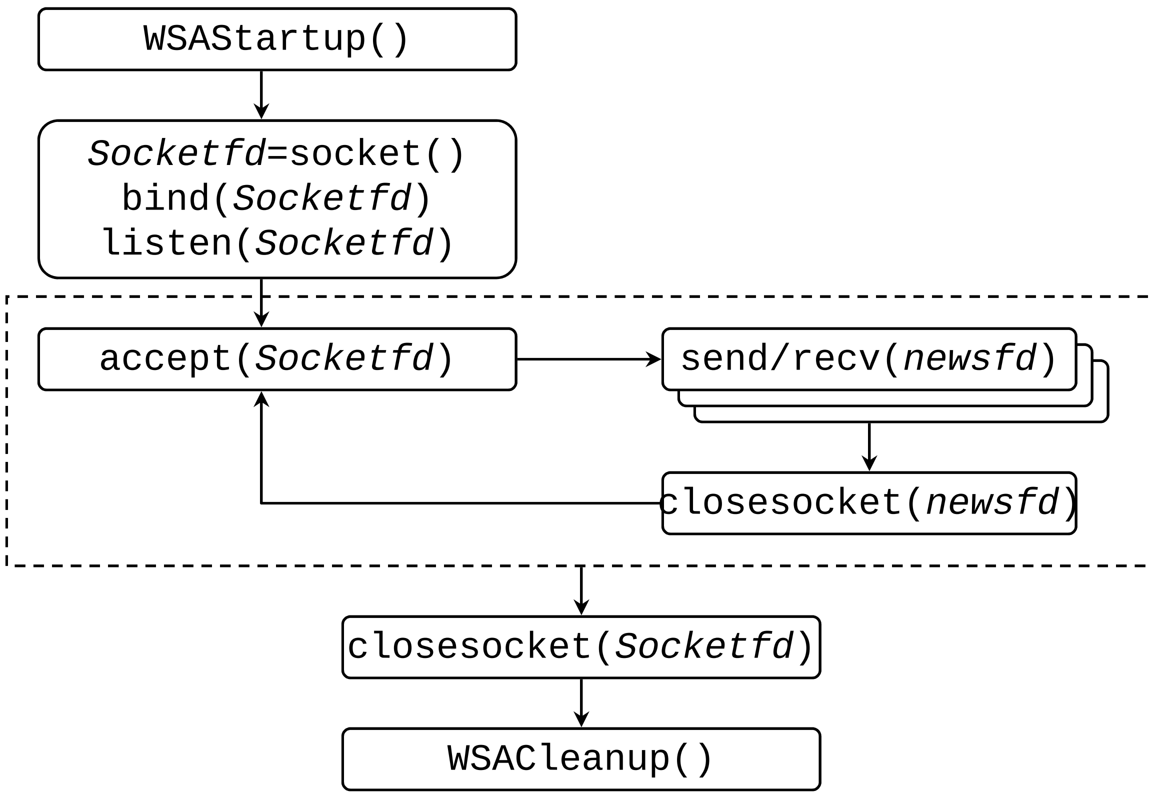
- Contrast to Thin-Client (or Terminal) Model
  - Client responsible for user interface rendering and interaction only
  - No transfer of actual data to client for processing



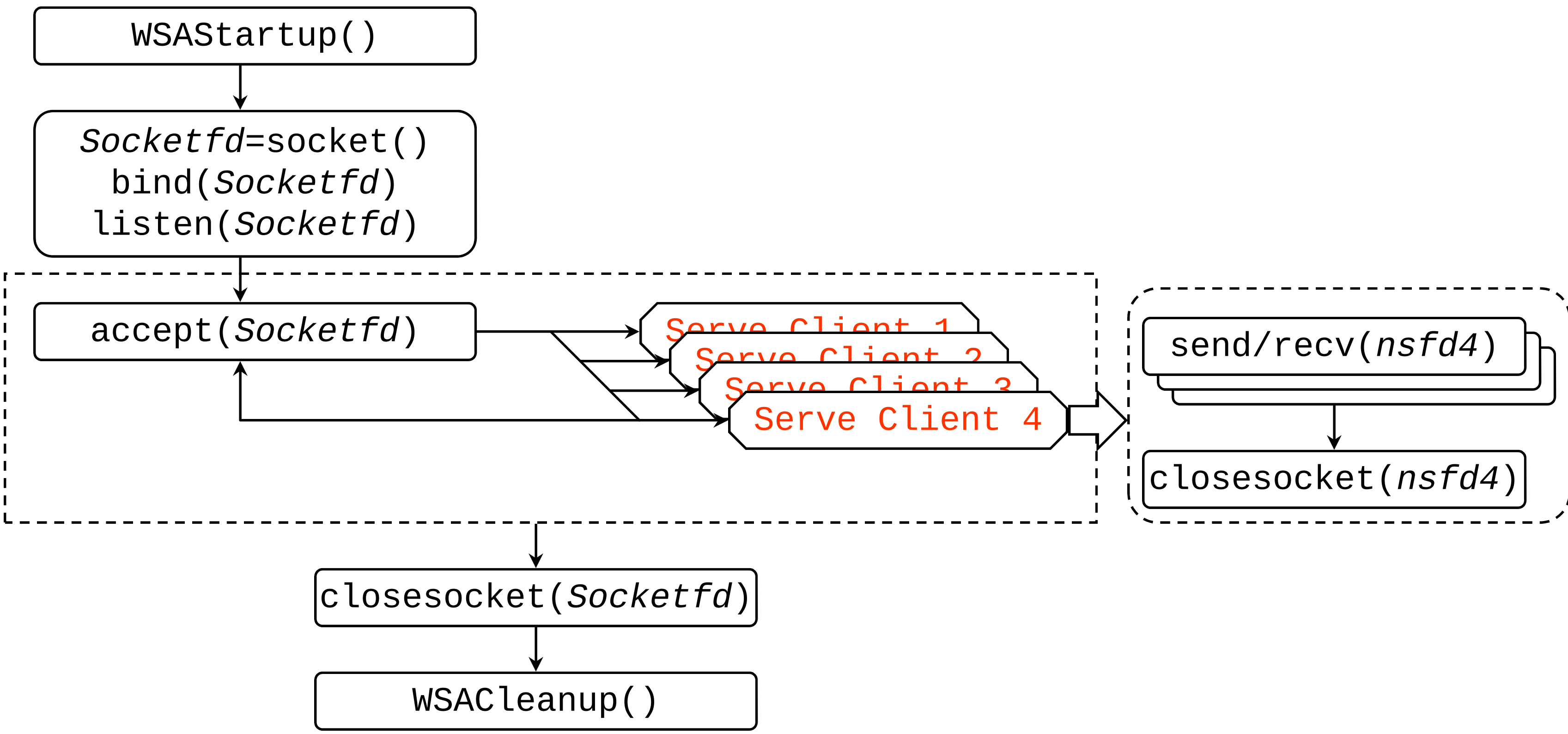
Examples:
True Client-Server: Playing a multiplayer car racing game using PC connected to a game server.
Terminal Model: Playing games via platforms such as OnLive (acquired by Sony in 2015).

2. Iterative Server

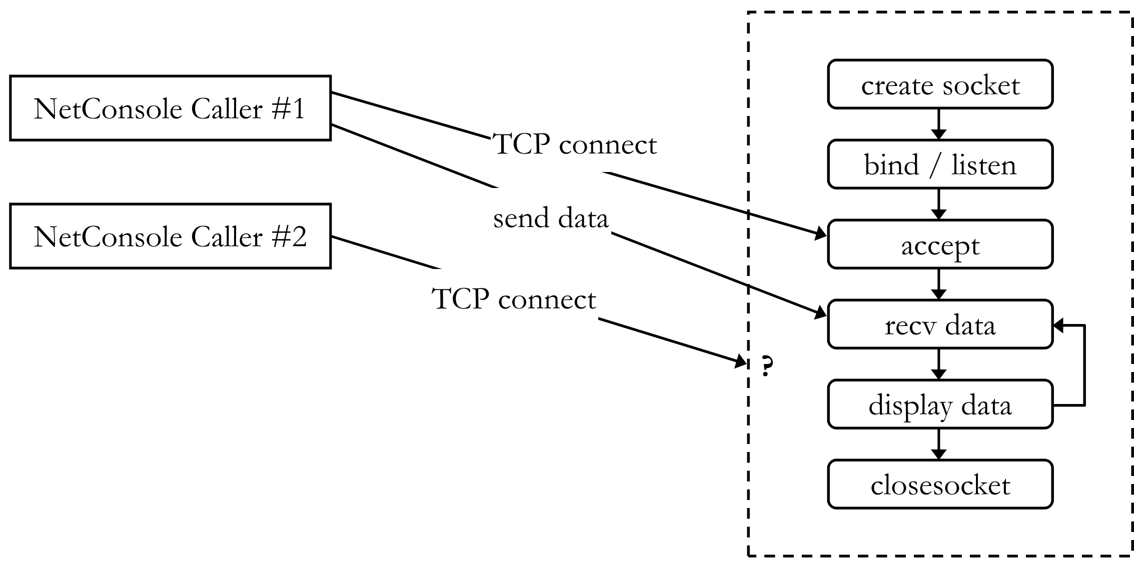
- Serves one client at a time (e.g., NetConsole Listener).
  - Subsequent request will need to wait until previous one finishes (how?)



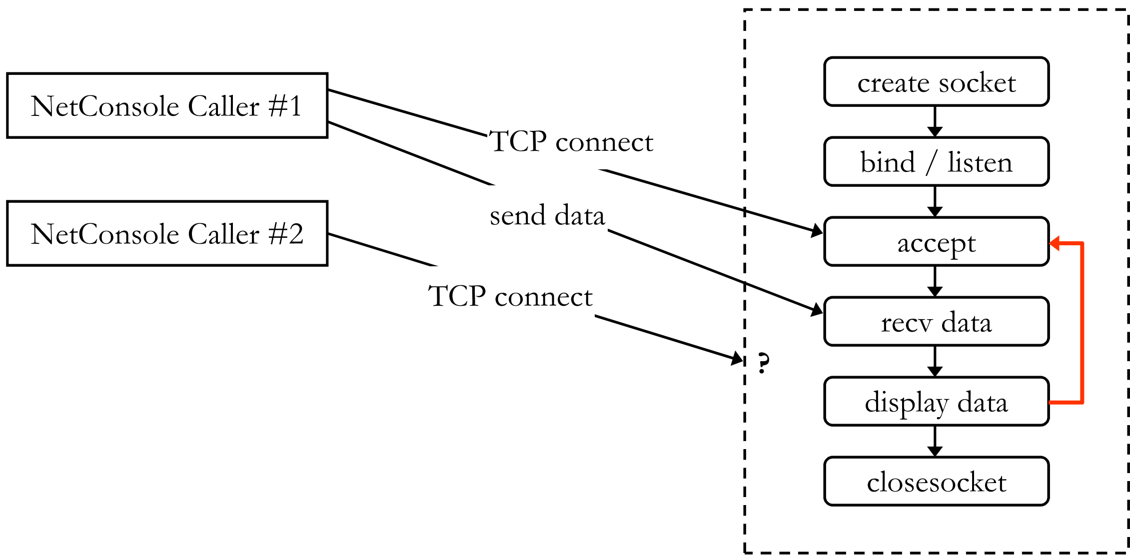
- Able to serve multiple clients concurrently.
  - Server resources (e.g., bandwidth, memory) are shared across concurrent client sessions



- Synchronous programming and blocking I/O model.
  - Accepting a second connection:



- Synchronous programming and blocking I/O model.
  - Accepting a second connection:

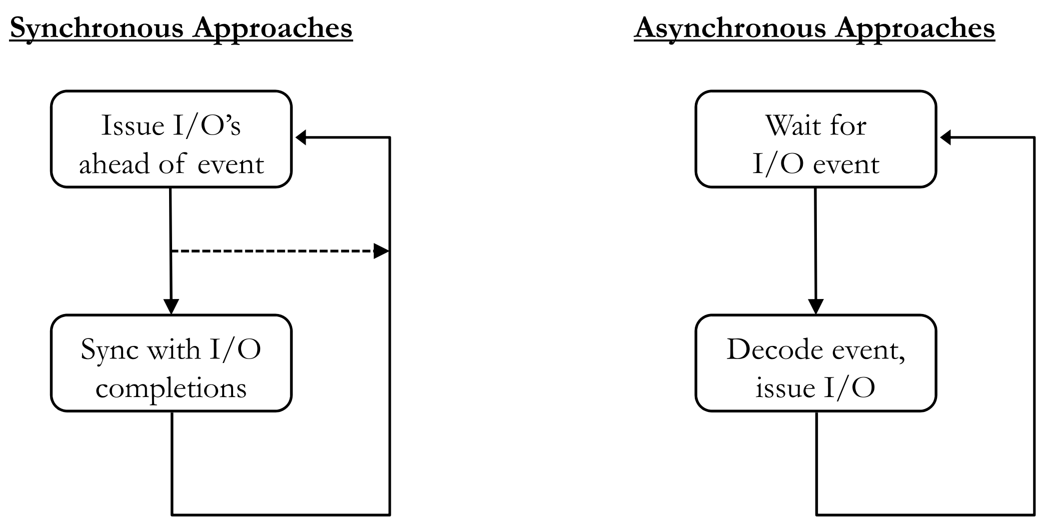


Question: What is the fundamental problem here?

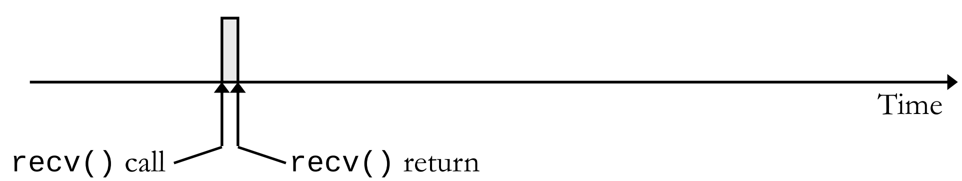
4. I/O Multiplexing

- The Problem
  - Program execution is driven by process/thread
  - A process/thread blocked by an I/O call will become suspended, and unable to serve other pending I/O calls
- Synchronous Programming Approaches*
  - Non-blocking I/O with polling (Unix, Windows)
  - The `select()` system call (Unix, Windows, also Java)
  - Multi-Process with blocking I/O (Unix)
  - Multi-threading with blocking I/O (Unix, Windows, Java)
  - Overlapped I/O (Windows only)
  - Combinations of the above

- Asynchronous Programming Approaches
  - Message-driven I/O (Windows only)
  - I/O Completion Port, Registered I/O (Windows only)
  - Unix Signals SIGIO (Unix only)
  - Linux `poll` and `epoll`
  - Combination with multi-process / multi-threading



- Turns a socket into non-blocking mode
  - On a per-socket basis, default is blocking
  - Operations on non-blocking sockets are guaranteed to not block
  - Three possible outcomes
    - Completed successfully
      - e.g., `recv()` returns > 0
    - Failed
      - e.g., `recv()` returns `SOCKET_ERROR` with `WSAGetLastError() != WSAEWOULDBLOCK`
    - Cannot complete w/o blocking
      - e.g., `recv()` returns `SOCKET_ERROR` with `WSAGetLastError() == WSAEWOULDBLOCK`



5. Non-Blocking I/O with Polling

- Turns a socket into non-blocking mode

```
void ConcurrentListenerUsingNonBlockingIOandPolling(char *address, char *port)
{
   /// Step 1: Prepare address structures. ///
   sockaddr_in *TCP_Addr  = new sockaddr_in;
   memset (TCP_Addr, 0, sizeof(struct sockaddr_in));
   TCP_Addr->sin_family = AF_INET;
   TCP_Addr->sin_port   = htons(atoi(port));
   TCP_Addr->sin_addr.s_addr   = INADDR_ANY;

   /// Step 2: Create a socket for incoming connections. ///
   SOCKET Sockfd = socket(AF_INET, SOCK_STREAM, 0);
   bind(Sockfd, (struct sockaddr *)TCP_Addr, sizeof(struct sockaddr_in));
   listen(Sockfd, 5);
   unsigned long ul_val = 1; // 1 means enable
   if (ioctlsocket(Sockfd, FIONBIO, &ul_val) == SOCKET_ERROR) {
      printf("\nioctlsocket() failed. Error code: %i\n", WSAGetLastError());
      return;
   }

   // ...
```

Note: For Linux use `ioctl()` instead of `ioctlsocket()`.

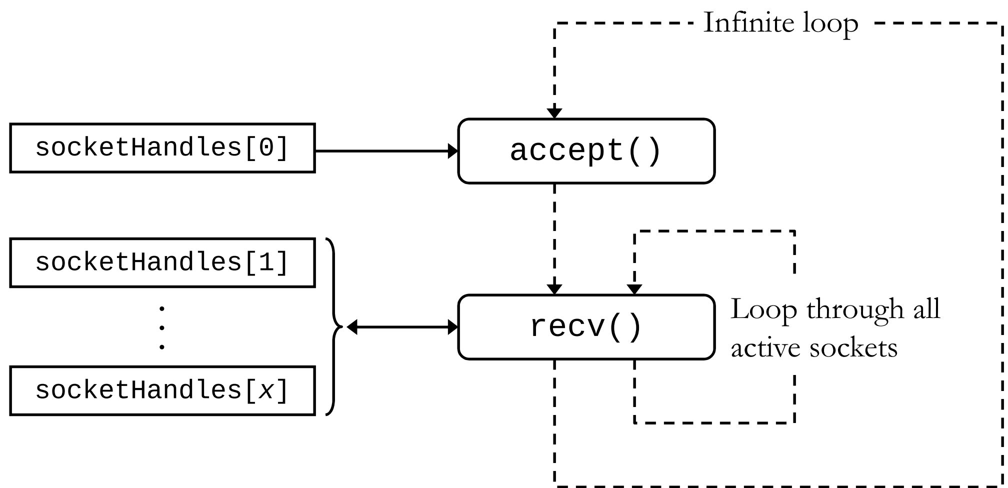
- `accept()` will become non-blocking:

```
/// Step 3: Setup the data structures for multiple connections. ///
const int maxSockets = 10;
SOCKET socketHandles[maxSockets]; // Array for the socket handles
bool   socketValid[maxSockets];   // Bitmask to manage the array
int    numActiveSockets = 1;
for (int i=1; i<maxSockets; i++) socketValid[i] = false;
socketHandles[0] = Sockfd;  socketValid[0] = true;

/// Step 4: Poll all active sockets for data/accept. ///
while (1) {
   // Check for incoming connection first.
   if (socketValid[0]) {
      SOCKET newsfd = accept(Sockfd, 0, 0);
      if (newsfd != SOCKET_ERROR) { // accept() succeeded
         // Append the new entry to the socketHandles[] array//
         socketHandles[numActiveSockets] = newsfd;
         socketValid[numActiveSockets] = true;
         ++numActiveSockets;
         if (numActiveSockets == maxSockets) { // Suspend accept
            socketValid[0] = false;
         }
      } else if (WSAGetLastError() != WSAEWOULDBLOCK) {
         printf("\naccept() failed. Error code: %i\n", WSAGetLastError());
         return;
      } // Note the case for WSAEWOULDBLOCK is ignored.
   }
```

- Poll the rest of the active sockets:

```
// ...
for (int i=1; i<numActiveSockets; i++) {
   if (socketValid[i]) {
      // Receive data and display to stdout. //
      char buf[256]; int len=255;
      int numread = recv(socketHandles[i], buf, len, 0);
      if (numread == SOCKET_ERROR) {
         if (WSAGetLastError()!=WSAEWOULDBLOCK) {
            printf("\nrecv() failed. Error: %i\n", WSAGetLastError());
            return;
         }
         continue;
      } else if (numread == 0) { // connection closed
         // Pack the socket array
         closesocket(socketHandles[i]);
         --numActiveSockets;
         socketHandles[i] = socketHandles[numActiveSockets];
         socketValid[numActiveSockets] = false;
         if (numActiveSockets == (maxSockets-1))
            socketValid[0] = true;
      } else {
         printf("[%i]: ", i);
         for (int x=0; x<numread; x++) putchar(buf[x]);
      }
   }
} // end-for
```

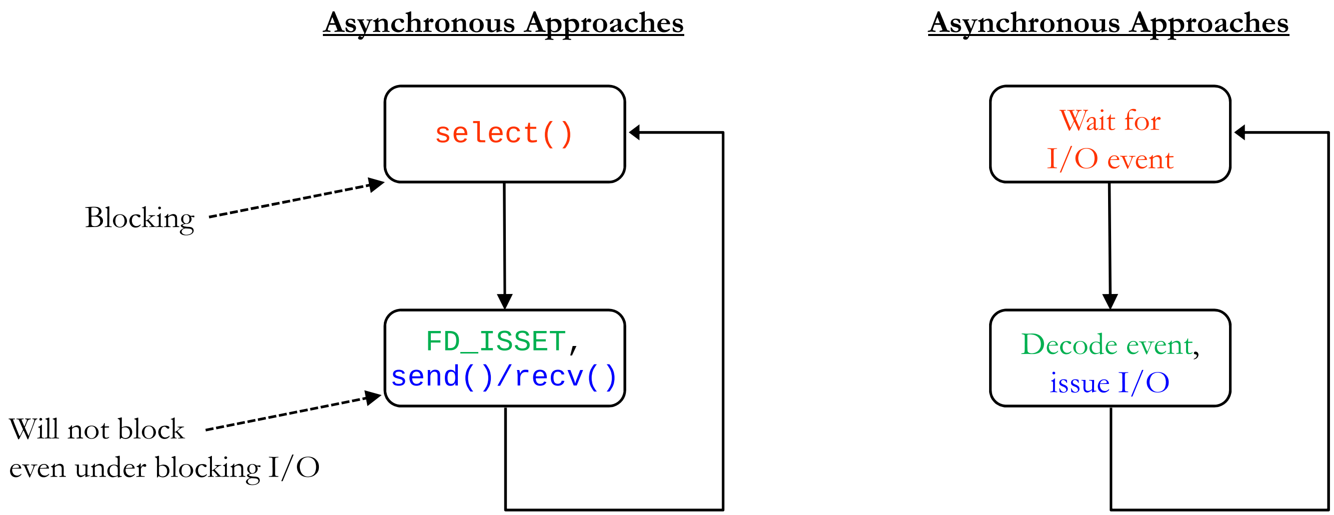
- Program Flow



Question: What is the tradeoff of polling non-blocking sockets?

6. I/O Multiplexing Using `select()`

- Asynchronous Approach
  - Blocking I/O, single thread




```
int select( int nfds,           // Max fd+1 in Unix/Linux, ignored in windows.
            fd_set* readfds,    // Set of sockets for read events.
            fd_set* writefds,   // Set of sockets for write events.
            fd_set* exceptfds, // Set of sockets for exceptions/errors.
            const struct timeval* timeout // Max time to block.
          );
```

- The `select()` API function:

```
int select( int nfds,          // Max fd+1 in Unix/Linux, ignored in windows.
            fd_set* readfds,   // Set of sockets for read events.
            fd_set* writefds,  // Set of sockets for write events.
            fd_set* exceptfds, // Set of sockets for exceptions/errors.
            const struct timeval* timeout // Max time to block.
          );
```

- Data Structures:

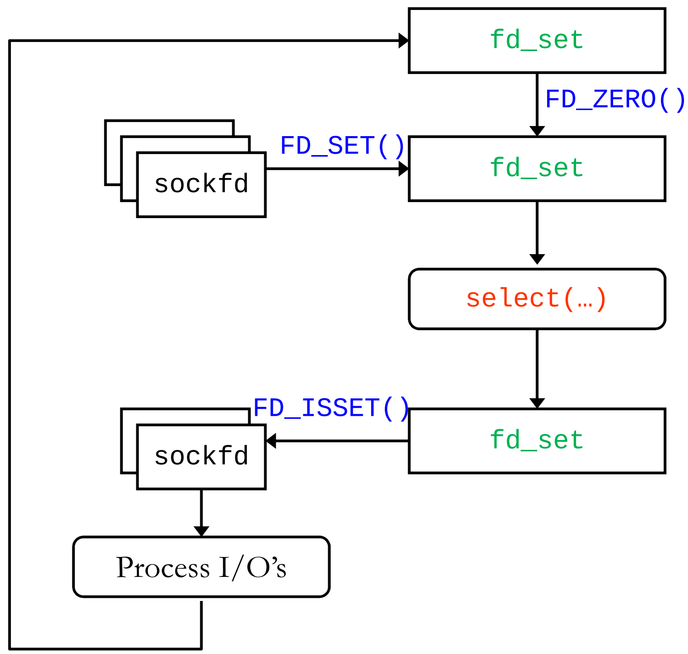
```
typedef struct fd_set {
   u_int fd_count;              // Number of sockets in the set.
   SOCKET fd_array[FD_SETSIZE]; // Array of sockets that are in the set.
} fd_set;

typedef struct timeval {
   long tv_sec;   // seconds
   long tv_usec;  // microseconds
} timeval;
```

- Macros:

```
FD_CLR  (s, *set);  // Removes the socket handle s from set.
FD_ISSET(s, *set);  // Nonzero if s is a member of the set. Otherwise, zero.
FD_SET  (s, *set);  // Adds socket handle s to set.
FD_ZERO (*set);     // Initializes the set to the null set.
```

6. I/O Multiplexing Using `select()`

- Program Flow:



`fd_set` contains the list of sockets to monitor.

`select()` blocks until registered events occur.

`fd_set` is updated to contain only sockets with events.

Process the I/O events for *each* returned sockets.

```
int select( int nfds,           // Max fd+1 in Unix/Linux, ignored in windows.
            fd_set* readfds,    // Set of sockets for read events.
            fd_set* writefds,   // Set of sockets for write events.
            fd_set* exceptfds,  // Set of sockets for exceptions/errors.
            const struct timeval* timeout // Max time to block.
          );
```

7. Concurrent NetConsole

- Implements Listener using `select()`:

```
void ConcurrentListenerUsingSelect (char *address, char *port)
{
   /// Step 1: Prepare address structures. ///
   sockaddr_in *TCP_Addr  = new sockaddr_in;
   memset (TCP_Addr, 0, sizeof(struct sockaddr_in));
   TCP_Addr->sin_family = AF_INET;
   TCP_Addr->sin_port   = htons(atoi(port));
   TCP_Addr->sin_addr.s_addr   = INADDR_ANY;

   /// Step 2: Create a socket for incoming connections. ///
   SOCKET Sockfd = socket(AF_INET, SOCK_STREAM, 0);
   bind(Sockfd, (struct sockaddr *)TCP_Addr, sizeof(struct sockaddr_in));
   listen(Sockfd, 5);

   /// Step 3: Setup the data structures for multiple connections. ///
   const int maxSockets = 10;        // At most 10 concurrent clients
   SOCKET socketHandles[maxSockets]; // Array for the socket handles
   bool   socketValid[maxSockets];   // Bitmask to manage the array
   int    numActiveSockets = 1;
   for (int i=1; i<maxSockets; i++) socketValid[i] = false;

   socketHandles[0] = Sockfd;
   socketValid[0] = true;
```

continue …

- Implements Listener using `select()`:

```
/// Step 4: Setup the select() function call for I/O multiplexing. ///

fd_set fdReadSet;
while (1) {

   // Setup the fd_set //
   int topActiveSocket = 0;
   FD_ZERO(&fdReadSet);
   for (int i=0; i<maxSockets; i++) {
      if (socketValid[i] == true) {
         FD_SET(socketHandles[i], &fdReadSet);
         if (socketHandles[i] > topActiveSocket)
            topActiveSocket = socketHandles[i]; // for Linux compatibility
      }
   }

   // Block on select() //

   int ret;
   if ((ret = select(topActiveSocket+1, &fdReadSet,
                     NULL, NULL, NULL)) == SOCKET_ERROR) {
      printf("\nselect() failed. Error code: %i\n", WSAGetLastError());
      return;
   }
```

continue …

- Implements Listener using `select()`:

```
// Process the active sockets //
for (i=0; i<maxSockets; i++) {
   if (!socketValid[i]) continue; // Only check for valid sockets.
   if (FD_ISSET(socketHandles[i], &fdReadSet)) { // Is socket i active?
      if (i==0) { // the socket for accept()
         // accept new connection //
         SOCKET newsfd = accept(Sockfd, 0, 0);
         // Find a free entry in the socketHandles[] //
         int j = 1;
         for ( ; j<maxSockets; j++) {
            if (socketValid[j] == false) {
               socketValid[j] = true;
               socketHandles[j] = newsfd;
               ++numActiveSockets;
               if (numActiveSockets == maxSockets) {
                  // Ignore new accept()
                  socketValid[0] = false;
               }
            break;
            }
         }
      }
```

continue …

- Implements Listener using `select()`:

```
          else { // sockets for recv()
             // Receive data and display to stdout. //
             char buf[256]; int len=255;
             int numread = recv(socketHandles[i], buf, len, 0);
             if (numread == SOCKET_ERROR) {
                printf("\nrecv() failed. Error code: %i\n",
                       WSAGetLastError());
                return;
             } else if (numread == 0) { // connection closed
                // Update the socket array
                socketValid[i] = false;
                --numActiveSockets;
                if (numActiveSockets == (maxSockets-1))
                   socketValid[0] = true;
                closesocket(socketHandles[i]);
             } else {
                printf("[%i]: ", i);
                for (int x=0; x<numread; x++) putchar(buf[x]);
             }
          }
          if (--ret == 0) break; // All active sockets processed.
       }
    }
 }
```

8. Final Remarks on `select()`

- Compatibility
  - The `select()`-based I/O model originates from BSD sockets.
  - Does not require multiple processes or multi-threading
  - The Winsock implementation is slightly different but compatible.
  - Winsock also implemented another variant of `select()`
    - `WSAAsyncSelect()`
- Limitations
  - The maximum number of sockets that can be handled by each `select()` is implementation dependent.
  - Linux can handle up to 1024 handles (minus stdin, stdout, and stderr).
  - Winsock varies from 1024 to very large number (e.g., 500K+ in Windows 10 Enterprise)
  - Overheads increase with number of sockets in fd_set.

8. Final Remarks on `select()`

- Setting up the `fd_set` in every iteration:

```
/// Step 4: Setup the select() function call for I/O multiplexing. ///
fd_set fdReadSet;
while (1) {
   FD_ZERO(&fdReadSet); int topActiveSocket = 0;
   for (int i=0; i<maxSockets; i++) {
      if (socketValid[i] == true) {
         FD_SET(socketHandles[i], &fdReadSet);
         if (socketHandles[i] > topActiveSocket)
            topActiveSocket = socketHandles[i]; // for Linux
      }
   }
```

- Finding out which socket has event for processing:

```
   // Process the active sockets //
   for (i=0; i<maxSockets; i++) { // Scan through all valid sockets
      if (!socketValid[i]) continue; // Only check for valid sockets.
      if (FD_ISSET(socketHandles[i], &fdReadSet)) { // Is socket i active?
         if (i==0) { // the socket for accept()
            // accept new connection //
            // ...(omitted)
         }
```

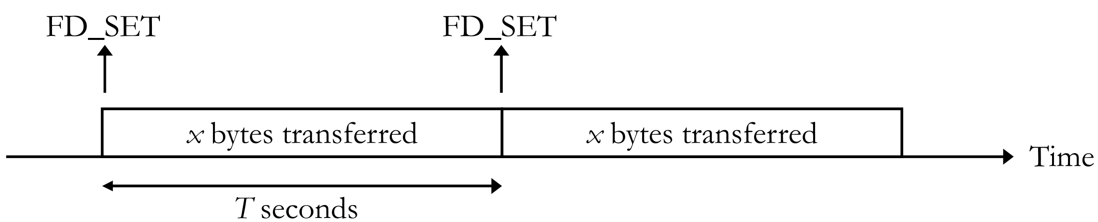
8. Final Remarks on `select()`

- Benchmarking (NetConsole)

```
// Setup the fd_set //
int t=0;
timer.Start();
FD_ZERO(&fdReadSet);
t = timer.ElapseduSec();
printf("\nFD_ZERO: %i micro-seconds.\n", t);

timer.Start();
for (int i=0; i<maxSockets; i++) {
   if (socketValid[i] == true) {
      FD_SET(socketHandles[i], &fdReadSet);
   }
}
t = timer.ElapseduSec();
printf("\nFD_SET: %i micro-seconds.\n", t);

// ...
   timer.Start();
   if (FD_ISSET(socketHandles[i], &fdReadSet)) { // Is socket i active?
   t = timer.ElapseduSec();
   printf("\nFD_ISSET (Active): %i micro-seconds.\n", t);
```

8. Final Remarks on `select()`

- Benchmarking (NetConsole)
  - Results (time in micro-seconds)
    - FD_SET
      - 64 sockets ~ 9 micro-secs
      - 128 sockets ~ 40 micro-secs
      - 256 sockets ~ 130 micro-secs
      - 512 sockets ~ 370 micro-secs
      - 1024 sockets ~1400 micro-secs



Transfer Rate = x/T
Example: 512 sockets, x=64 bytes, T=370μs, then transfer rate = ~1.4Mbps.

References

- Winsock2 Reference, MSDN, http://msdn2.microsoft.com/en-us/library/ms741416.aspx