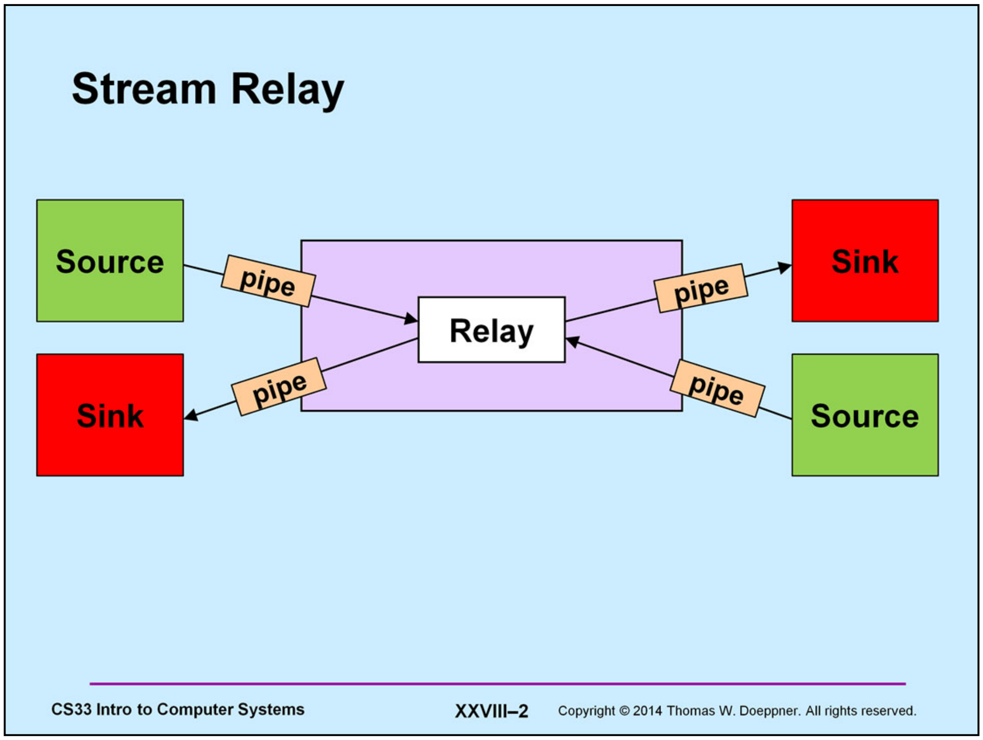
CS 33

More Network Programming

The source code used in this lecture is on the course web page.



We start by looking at what's known as *event-based programming*: we write code that responds to events coming from a number of sources. As a simple example, before we use the approach in a networking example, we exam a simple relay: we want to write a program that takes data, via a pipe, from the left source and sends it, via a pipe, to the right sink. At the same time it takes data from the right source and sends it to the left sink.

Solution?

```
while(…) {
   size = read(left, buf, sizeof(buf));
   write(right, buf, size);
   size = read(right, buf, sizeof(buf));
   write(left, buf, size);
}
```


This solution is probably not what we'd want, since it strictly alternates between processing the data stream in one direction and then the other.

Select System Call

```
int select(
  int nfds,          // size of fd_sets
  fd_set *readfds,   // descriptors of interest
                     // for reading
  fd_set *writefds,  // descriptors of interest
                     // for writing
  fd_set *excpfds,   // descriptors of interest
                     // for exceptional events
  struct timeval *timeout
                     // max time to wait
);
```


Relay Sketch

```
void relay(int left, int right) {
   fd_set rd, wr;
   int maxFD = max(left, right) + 1;
   FD_ZERO(&rd); FD_SET(left, &rd); FD_SET(right, &rd);
   FD_ZERO(&wr); FD_SET(left, &wr); FD_SET(right, &wr);
   while (1) {
      select(maxFD, &rd, &wr, 0, 0);
      if (FD_ISSET(left, &rd))
         read(left, bufLR, BSIZE);
      if (FD_ISSET(right, &rd))
         read(right, bufRL, BSIZE);
      if (FD_ISSET(right, &wr))
         write(right, bufLR, BSIZE);
      if (FD_ISSET(left, &rd))
         write(left, bufRL, BSIZE);
   }
}
```


Here a simplified version of a program to handle the relay problem using *select*. An *fd_set* is a data type that represents a set of file descriptors. FD_ZERO, FD_SET, and FD_ISSET are macros for working with fd_sets; the first makes such a set represent the null set, the second sets a particular file descriptor to included in the set, the last checks to see if a particular file descriptor is included in the set.

Relay (1)

```
void relay(int left, int right) {
  fd_set rd, wr;
  int left_read = 1, right_write = 0;
  int right_read = 1, left_write = 0;
  int sizeLR, sizeRL, wret;
  char bufLR[BSIZE], bufRL[BSIZE];
  char *bufpR, *bufpL;
  int maxFD = max(left, right) + 1;
```


This and the next three slides give a more complete version of the relay program.

Initially our program is prepared to read from either the left or the right side, but it's not prepared to write, since it doesn't have anything to write.

Relay (2)

```
while(1) {
  FD_ZERO(&rd);
  FD_ZERO(&wr);
  if (left_read)
    FD_SET(left, &rd);
  if (right_read)
    FD_SET(right, &rd);
  if (left_write)
    FD_SET(left, &wr);
  if (right_write)
    FD_SET(right, &wr);

  select(maxFD, &rd, &wr, 0, 0);
```


We set up the fd_sets rd and wr to indicate but we are interested in reading from and writing to (initially we have no interest in writing, but are interested in reading from either side).

Relay (3)

```
if (FD_ISSET(left, &rd)) {
  sizeLR = read(left, bufLR, BSIZE);
  left_read = 0;
  right_write = 1;
  bufpR = bufLR;
}
if (FD_ISSET(right, &rd)) {
  sizeRL = read(right, bufRL, BSIZE);
  right_read = 0;
  left_write = 1;
  bufpL = bufRL;
}
```


If there is something to read from the left side, we read it. Having read it, we're temporarily not interested in reading anything further from the left side, but now want to write to the right side.

In a similar fashion, if there is something to read from the right side, we read it.

Relay (4)

```
    if (FD_ISSET(right, &wr)) {
      if ((wret = write(right, bufpR, sizeLR)) == sizeLR) {
        left_read = 1; right_write = 0;
      } else {
        sizeLR -= wret; bufpR += wret;
      }
    }
    if (FD_ISSET(left, &wr)) {
      if ((wret = write(left, bufpL, sizeRL)) == sizeRL) {
        right_read = 1; left_write = 0;
      } else {
        sizeRL -= wret; bufpL += wret;
      }
    }
  }
  return 0;
}
```


Writing is a bit more complicated, since the outgoing pipe might not have room for everything we have to write, but just some of it. Thus we must pay attention to what write returns. If everything has been written, then we can go back to reading from the other side, but if not, we continue trying to write.

A Really Simple Protocol

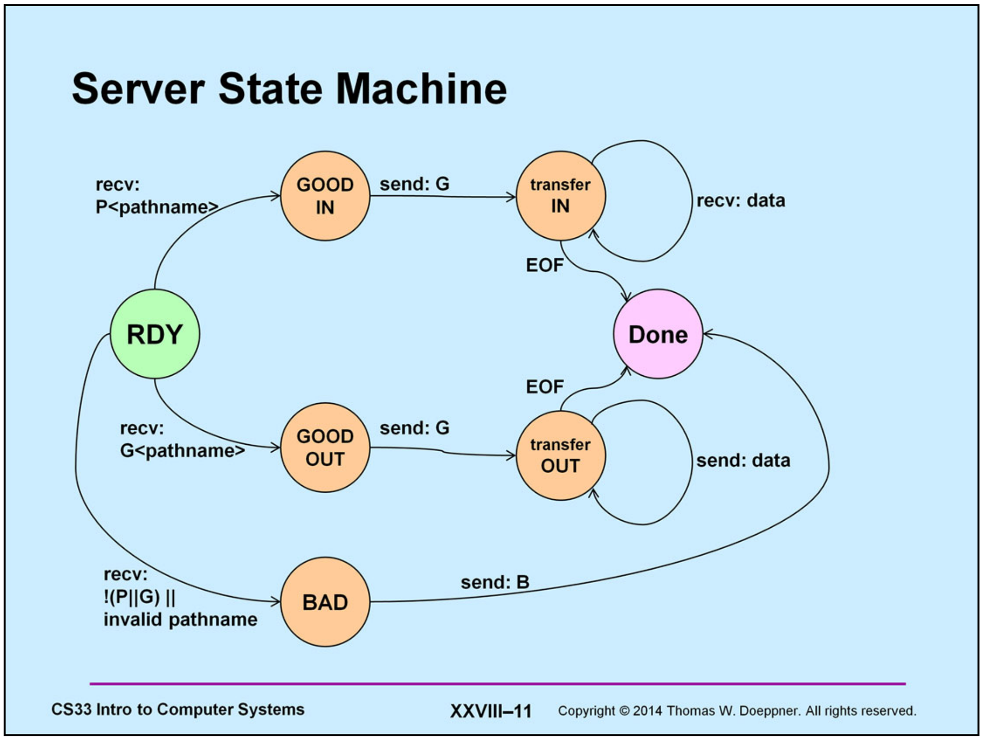
- **Transfer a file**
  - layered on top of TCP
    - » reliable
    - » indicates if connection is closed
- **To send a file**

  P<null-terminated pathname><contents of file>
- **To retrieve a file**

  G<null-terminated pathname><contents of file>

Now we use the event paradigm to implement a simple file-transfer program.

A complete version of the program shown here is on the course web page.



We design the protocol in terms of a simple state machine.

Keeping Track of State

```
typedef struct client {
  int fd;     // file descriptor of local file being transferred
  int size;   // size of out-going data in buffer
  char buf[BSIZE];
  enum state {RDY, BAD, GOOD, TRANSFER} state;
  /*
     states:
        RDY: ready to receive client's command (P or G)
        BAD: client's command was bad, sending B response + error msg
        GOOD: client's command was good, sending G response
        TRANSFER: transferring data
    */
  enum dir {IN, OUT} dir;
  /*
     IN: client has issued P command
     OUT: client has issued G command
   */
} client_t;
```


Note the use of the *enum* data type. Variables of this type have a finite set of possible values, as given in the declaration.

Keeping Track of Clients

```
client_t clients[MAX_CLIENTS];
for (i=0; i < MAX_CLIENTS; i++)
  clients[i].fd = -1; // illegal value
```


Each client of our server is represented by a separate client_t structure. We allocate an array of them and will refer to client's client_t structure by the file descriptor of the socket used to communicate with it. Thus if we're using a socket whose file desriptor is sfd to communicate with a client, then the client's state is in clients[sfd].

Main Server Loop

```
while(1) {
  select(maxfd, &trd, &twr, 0, 0);
  if (FD_ISSET(lsock, &trd)) {
    // a new connection
    new_client(lsock);
  }
  for (i=lsock+1; i<maxfd; i++) {
    if (FD_ISSET(i, &trd)) {
      // ready to read
      read_event(i);
    }
    if (FD_ISSET(i, &twr)) {
      // ready to write
      write_event(i);
    }
  }
  trd = rd; twr = wr;
}
```


lsock is the file descriptor for the "listening-mode" socket on which the server is waiting for connections. Our server may be handling multiple clients; each will be communicating with the server via a separate connected socket. These sockets have file descriptors greater than *lsock*. Note that *trd*, *twr*, *rd* and *wr* are all of type *fd_set*. *rd* and *wr* are initialized so that *rd* contains just the file descriptor for the listening socket and *wr* is empty. *trd* and *twr* are copied from *rd* and *wr* respectively before the loop is entered.

New Client

```
// Accept a new connection on listening socket
// fd. Return the connected file descriptor

int new_client(int fd) {
  int cfd = accept(fd, 0, 0);
  clients[cfd].state = RDY;
  FD_SET(cfd, &rd);
  return cfd;
}
```


Read Event (1)

```
// File descriptor fd is ready to be read. Read it, then handle
// the input
void read_event(int fd) {
  client_t *c = &clients[fd];
  int ret = read(fd, c->buf, BSIZE);
  switch (c->state) {
  case RDY:
    if (c->buf[0] == 'G') {
      // GET request (to fetch a file)
      c->dir = OUT;
      if ((c->fd = open(&c->buf[1], O_RDONLY)) == -1) {
        // open failed; send negative response and error message
        c->state = BAD;
        c->buf[0] = 'B';
        strncpy(&c->buf[1], strerror(errno), BSIZE-2);
        c->buf[BSIZE-1] = 0;
        c->size = strlen(c->buf)+1;
      }
```


Read Event (2)

```
    else {
      // open succeeded; send positive response
      c->state = GOOD;
      c->size = 1;
      c->buf[0] = 'G';
    }
    // prepare to send response to client
    FD_SET(fd, &wr);
    FD_CLR(fd, &rd);
    break;
  }
```


Read Event (3)

```
  if (c->buf[0] == 'P') {
    // PUT request (to create a file)
    c->dir = IN;
    if ((c->fd = open(&c->buf[1],
        O_RDWR|O_CREAT|O_TRUNC, 0666)) == -1) {
        // open failed; send negative response and error message
        ...
  } else {
      // open succeeded; send positive response
      ...
  }
  // prepare to send response to client
  FD_SET(fd, &wr);
  FD_CLR(fd, &rd);
  break;
}
```

Read Event (4)

```
case TRANSFER:
  // should be in midst of receiving file contents from client
  if (ret == 0) {
    // eof: all done
    close(c->fd);
    close(fd);
    FD_CLR(fd, &rd);
    break;
  }
  if (write(c->fd, c->buf, ret) == -1) {
    // write to file failed: terminate connection to client
    ...
    break;
  }
  // continue to read more data from client
  break;
}
```

Write Event (1)

```
// File descriptor fd is ready to be written to. Write to it, then,
// depending on current state, prepare for the next action.
void write_event(int fd) {
  client_t *c = &clients[fd];
  int ret = write(fd, c->buf, c->size);
  if (ret == -1) {
    // couldn't write to client; terminate connection
    close(c->fd);
    close(fd);
    FD_CLR(fd, &wr);
    c->fd = -1;
    perror("write to client");
    return;
  }
  switch (c->state) {
```


Write Event (2)

```
case BAD:
  // finished sending error message; now terminate client connection
  close(c->fd);
  close(fd);
  FD_CLR(fd, &wr);
  c->fd = -1;
  break;
```

Write Event (3)

```
case GOOD:
  c->state = TRANSFER;
  if (c->dir == IN) {
    // finished response to PUT request
    FD_SET(fd, &rd);
    FD_CLR(fd, &wr);
    break;
  }
  // otherwise finished response to GET request, so proceed
```


Write Event (4)

```
 case TRANSFER:
   // should be in midst of transferring file contents to client
   if ((c->size = read(c->fd, c->buf, BSIZE)) == -1) {
     ...
     break;
   } else if (c->size == 0) {
     // no more file to transfer; terminate client connection
     close(c->fd);
     close(fd);
     FD_CLR(fd, &wr);
     c->fd = -1;
     break;
   }
   // continue to write more data to client
   break;
 }
}
```


Problems

- Works fine as long as protocol is followed correctly
  - can client (malicious or incompetent) cause server to misbehave?
- How can the server limit the number of clients?
- How does server limit file access?

