• Socket Options
• Posix name/address conversion
• Out-of-Band Data

It's important to know about some of these topics, although it might not be apparent how and when to use them.

Details are in the book - we are just trying to get some idea of what can be done.
Socket Options
Socket Options

- Various attributes that are used to determine the behavior of sockets.

- Setting options tells the OS/Protocol Stack the behavior we want.

- Support for generic options (apply to all sockets) and protocol specific options.
Option types

- Many socket options are Boolean flags indicating whether some feature is enabled (1) or disabled (0).

- Other options are associated with more complex types including `int`, `timeval`, `in_addr`, `sockaddr`, etc.
Read-Only Socket Options

- Some options are readable only (we can’t set the value).
Setting and Getting option values

`getsockopt()` gets the current value of a socket option.

`setsockopt()` is used to set the value of a socket option.

```
#include <sys/socket.h>
```
getsockopt()

```c
int getsockopt( int sockfd,
               int level,
               int optname,
               void *opval,
               socklen_t *optlen);
```

**level** specifies whether the option is a general option or a protocol specific option (what level of code should interpret the option).
# Socket and IP-layer socket options

<table>
<thead>
<tr>
<th>Level</th>
<th>Optname</th>
<th>Get</th>
<th>Set</th>
<th>Flag</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOL_SOCKET</td>
<td>SO_ERROR</td>
<td>Y</td>
<td>N</td>
<td></td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>SO_LINGER</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>SO_KEEPALIVE</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>linger</td>
</tr>
<tr>
<td>IPPROTO_IP</td>
<td>IP_HDRINCL</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Int</td>
</tr>
<tr>
<td></td>
<td>IP_TOS</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>int</td>
</tr>
<tr>
<td>IPPROTO_TCP</td>
<td>TCP_MAXSEG</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>TCP_NODELAY</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>int</td>
</tr>
</tbody>
</table>
setsockopt()

int setsockopt( int sockfd,
    int level,
    int optname,
    const void *opval,
    socklen_t optlen);
Example: SO_LINGER

- Specifies how the `close` function operates for a connection-oriented protocol.

```c
#include <unistd.h>
int close(int socketfd);
```

- Decrease the reference count for the descriptor
- If the reference count is 0:
  - send any data that is already queued to be sent to the other end
  - Normal TCP Connection termination sequence
SO_LINGER

Value is of type:

```c
struct linger {
    int l_onoff;  /* 0 = off */
    int l_linger;  /* time in seconds */
};
```

- Used to control whether and how long a call to close will wait for pending ACKS.
- connection-oriented sockets only.
SO_LINGER usage

- By default, calling `close()` on a TCP socket will return immediately.
- The closing process has no way of knowing whether or not the peer received all data.
- Setting SO_LINGER means the closing process can determine that the peer machine has received the data (but not that the data has been `read()`!).
SO_LINGER

- \( l\text{\_onoff} = 1 & l\text{\_linger} = 0 \)
  - TCP aborts the connections when it is closed

- \( l\text{\_onoff} = 1 & l\text{\_linger} \neq 0 \)
  - close return if either:
    - all the data is sent and acked
    - the linger time has expired.

- Check an example
shutdown

- Starts TCP’s normal connection termination sequence, regardless of the reference count

```c
#include <sys/socket.h>
int shutdown(int sockfd, int howto);
```

- **howto**
  - SHUT_RD: the read half of the connection is closed
  - SHUT_WR: the write half of the connection is closed
  - SHUT_RDWR: the read half and the write half of the connection are both closed
shutdown() vs SO_LINGER

**Summary**

- `close` returns immediately without waiting at all
- `close` lingers until the ACK of our FIN is received
- `shutdown` followed by a `read` waits until we receive the peer’s FIN
General Options

- Protocol independent options.
- Handled by the generic socket system code.
- Some general options are supported only by specific types of sockets (SOCK_DGRAM, SOCK_STREAM).
Some Generic Options

SO_BROADCAST
SO_DONTROUTE
SO_ERROR
SO_KEEPALIVE
SO_LINGER
SO_RCVBUF, SO_SNDBUF
SO_REUSEADDR
SO_BROADCAST

- Boolean option: enables/disables sending of broadcast messages.
- Underlying DL layer must support broadcasting!
- Applies only to SOCK_DGRAM sockets.
- Prevents applications from inadvertently sending broadcasts (OS looks for this flag when broadcast address is specified).
SO_DONTROUTE

- Boolean option: enables bypassing of normal routing.

- Used by routing daemons.
SO_ERROR

- Integer value option.
- The value is an error indicator value (similar to `errno`).
- Readable (get’able) only!
- Reading (by calling `getsockopt()`) clears any pending error.
SO_KEEPALIVE

- Boolean option: enabled means that STREAM sockets should send a *probe* to peer if no data flow for a “long time”.
- Used by TCP - allows a process to determine whether peer process/host has crashed.
- Consider what would happen to an open telnet connection without keepalive.
- Detect *half-open connections* and terminate them.
SO_RCVBUF
SO_SNDBUF

- Integer values options - change the receive and send buffer sizes.
- Can be used with STREAM and DGRAM sockets.

- With TCP, When should this option be set?
  - this option effects the window size used for flow control - must be established before connection is made.
SO_REUSEADDR

- Boolean option: enables binding to an address (port) that is already in use.

- By default, `bind` fails when the listening server is trying to bind a port that is part of an existing connection.

- How?
SO_REUSEADDR

- A listening server is started.

- A connection request arrives and a child process is spawned to handle that client.

- The listening server terminates, but the child continues to service the client on the existing connections.

- The listening server is restarted.
SO_REUSEADDR

- Used by servers that are transient - allows binding a passive socket to a port currently in use (with active sockets) by other processes.

- Can be used to establish separate servers for the same service on different interfaces (or different IP addresses on the same interface).
IP Options (IPv4): IPPROTO_IP

- IP_HDRINCL: used on raw IP sockets when we want to build the IP header ourselves.
- IP_TOS: allows us to set the “Type-of-service” field in an IP header.
- IP_TTL: allows us to set the “Time-to-live” field in an IP header.
TCP socket options
(IPPROTO_TCP)

- TCP_MAXSEG: set the maximum segment size sent by a TCP socket.
another TCP socket option

TCP_NODELAY: can disable TCP’s Nagle algorithm that delays sending small packets if there is unACK’d data pending.

TCP_NODELAY also disables delayed ACKS (TCP ACKs are cumulative).
Socket Options Summary

- This was just an overview
  - there are many details associated with the options described.
  - There are many options that haven’t been described.
  - Our text is one of the best sources of information about socket options.

- Let’s see an example:

```c
getsockopt(fd, IPPROTO_TCP, TCP_MAXSEG, &val, &len);
```
Posix name/address conversion
Posix Name/Adress Conversion

- We've seen `gethostbyname` and `gethostbyaddr` - these are protocol dependent.
  - Not part of sockets library.
- Posix includes protocol independent functions:
  - `getaddrinfo()`  `getnameinfo()`
gethostbyname

struct hostent *gethostbyname(
    const char *hostname);

struct hostent is defined in netdb.h:

#include <netdb.h>
`struct hostent`
hostent picture

- h_name
- h_aliases
- h_addrtype
- h_length
- h_addr_list

Official Name
- alias 1
- alias 2
- IP address 1
- IP address 2

null
getaddrinfo, getnameinfo

- These functions provide name/address conversions as part of the sockets library.

- In the future it will be important to write code that can run on many protocols (IPV4, IPV6).
Why `getaddrinfo()`?

- Puts protocol dependence in library (where it belongs).
  - Same code can be used for many protocols (IPV4, IPV6)
  - Re-entrant function - `gethostbyname` is not!
    - Important to threaded applications.
getaddrinfo()

int getaddrinfo(
    const char *hostname,
    const char *service,
    const struct addrinfo* hints,
    struct addrinfo **result);

getaddrinfo() replaces both
gethostbyname() and getservbyname()
getaddrinfo() parameters

hostname is a hostname or an address string (dotted decimal string for IP).

service is a service name or a decimal port number string.
struct addrinfo

struct addrinfo {
    int       ai_flags;
    int       ai_family;
    int       ai_socktype;
    int       ai_protocol;
    size_t    ai_addrlen;
    char      *canonname;
    struct sockaddr  *ai_addr;
    struct addrinfo  *ai_next;
};
getaddrinfo() hints

hints is an addrinfo *(can be NULL) that can contain:

- `ai_flags` (AI_PASSIVE, AI_CANONNAME)
- `ai_family` (AF_XXX)
- `ai_socktype` (SOCK_XXX)
- `ai_protocol` (IPPROTO_TCP, etc.)
getaddrinfo() result

result is returned with the address of a pointer to an addrinfo structure that is the head of a linked list.

It is possible to get multiple structures:

- multiple addresses associated with the hostname.
- The service is provided for multiple socket types.
**addrinfo usage**

- **ai_flags**
- **ai_family**
- **ai_socktype**
- **ai_protocol**
- **ai_addrlen**
- **ai_canonname**
- **ai_addr**
- **ai_next**

Used in call to **socket()**

```
int socket(int family, int type, int proto);
```

Used in call to **bind()**, **connect()**, **or sendto()**

```
int bind(int sockfd, const struct sockaddr *myaddr, int addrlen);
```

```
getnameinfo()

int getnameinfo(
    const struct sockaddr *sockaddr,
    socklen_t addrlen,
    char *host,
    size_t hostlen,
    char *serv,
    size_t servlen,
    int flags);

getnameinfo() looks up a hostname and a service name given a sockaddr
Out-of-Band Data
Out-of-Band *Data*

- TCP (and other transport layers) provide a mechanism for delivery of "high priority" data ahead of "normal data".
- We can almost think of this as 2 streams:

![Diagram showing two TCP ports with normal and special data flows](image)
TCP OOB Data

- TCP supports something like OOB data using URGENT MODE (a bit is set in a TCP segment header).

- A TCP segment header field contains an indication of the location of the urgent data in the stream (the byte number).
TCP Segment Format

- source port number
- destination port number
- sequence number
- acknowledgment number
- window size
- TCP checksum
- urgent pointer
- option (if any)
- data (if any)
Sending OOB Data

```c
send(sd,buff,1,MSG_OOB);
```

Use `send()` to put a single byte of urgent data in a TCP stream.

The TCP layer adds some segment header info to let the other end know there is some OOB data.
Receiving OOB Data

- The TCP layer generates a `SIGURG` signal and sends to the owner process of the socket.

- `select()` will tell you an exception condition is present.
Reading URG data
(a.k.a. re-urg-e-dataing)

- Depending on how things are set up:
  - the data can be read using `recv()` with a `MSG_OOB` flag set.
  - The data can be read *inline* and the receiving process can monitor the *out-of-band-mark* for the connection (using `sockatmark()`)
Questions

- Is there a difference between the signal function call
  
  ```c
  send(fd, "ab", 2, MSG_OOB);
  ```

- and the two function calls
  
  ```c
  send(fd, "a", 1, MSG_OOB);
  send(fd, "b", 1, MSG_OOB);
  ```
sockatmark

- Return whether the current byte is the out-of-band byte

- `read()` always stops at the out-of-band mark.
So what?

- OOB Data might be used:
  - a heartbeat between the client and server to detect early failure (example in the book).
  - A way to communicate an exceptional condition to a peer even when flow control has stopped the sender.
Rlogin: Server to Client Commands

- Need to mark command bytes because only one TCP connection is used
- Use TCP’s urgent mode to mark command bytes
- To send a command to client, server enters urgent mode and makes command byte the last byte of urgent data
- Four command bytes
  - 0x02: client flushes output
  - 0x10: client stops performing flow control
  - 0x20: client resumes flow control
  - 0x80: client sends current window size to server immediately and notifies server if changing window size
- Out-of-band signaling