Concurrent Computing (Computer Networks)

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
⟨Daniel.Page@bristol.ac.uk⟩

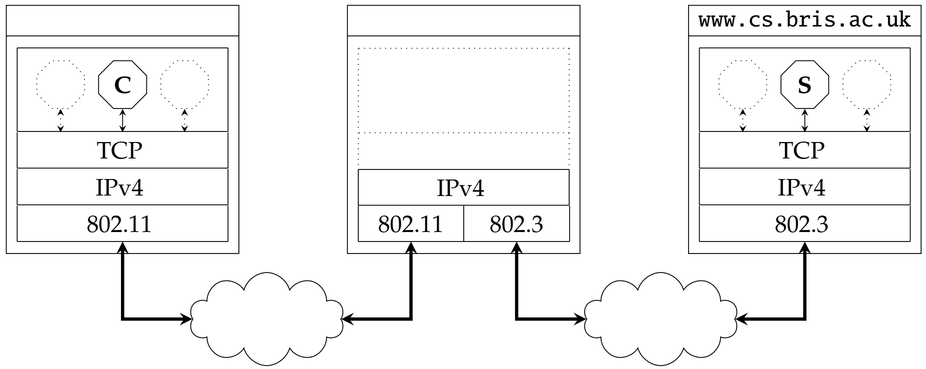
March 14, 2016

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
   - the associated notes page may be pre-populated with extra, written explaination of material covered in lecture(s), plus
   - anything with a "grey'ed out" header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:

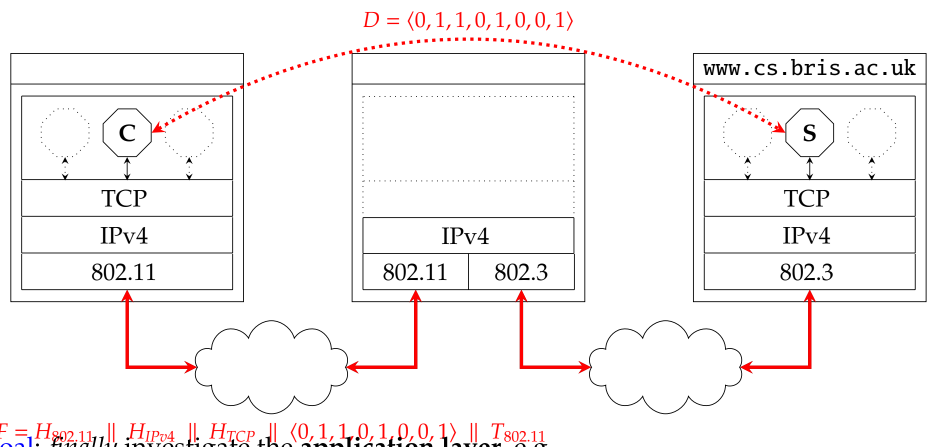


- Goal: *finally* investigate the **application layer**, e.g.,
  - the (mainly OS-based) network stack implementation,
  - the interface between application and network stack, i.e.,
    1. a **raw socket**, or
    2. the **POSIX sockets** API,

    and
  - examples of how *you* can use all this!

Notes:

- Using a raw socket is basically a way to bypass layers of the network stack, and use lower layers directly instead. `traceroute` may need to directly manipulate IP header fields, for example, rather than have the IPv4 layer act for it.
- The POSIX sockets API is effectively a standard version of the original TCP/IP implementation, namely **Berkeley sockets** (or **BSD sockets**) which first appeared in 1983. As a result, there's a tendency to think it as being "UNIX only". In reality, however, almost every OS uses roughly the same interface; **Winsock**, the Windows socket API, roughly does so for example.



- Goal: *finally* investigate the **application layer**, e.g.,
  - the (mainly OS-based) network stack implementation,
  - the interface between application and network stack, i.e.,
    1. a **raw socket**, or
    2. the **POSIX sockets** API,

    and
  - examples of how *you* can use all this!

Notes:

- Using a raw socket is basically a way to bypass layers of the network stack, and use lower layers directly instead. `traceroute` may need to directly manipulate IP header fields, for example, rather than have the IPv4 layer act for it.
- The POSIX sockets API is effectively a standard version of the original TCP/IP implementation, namely **Berkeley sockets** (or **BSD sockets**) which first appeared in 1983. As a result, there's a tendency to think it as being "UNIX only". In reality, however, almost every OS uses roughly the same interface; **Winsock**, the Windows socket API, roughly does so for example.

POSIX sockets API (1) – The Interface

	Function	Description	Blocking?
	`socket`	Form the data structure used to describe communication end-point	×
	`bind`	Associate socket data structure with (local) address	×
	`close`	Close socket and stop using it	×
	`shutdown`	Close socket and stop using it, with control over how	×
	`getsockopt` `setsockopt`	Get or set options for a socket, i.e., control how it functions	×
UDP	`sendto`	Transmit a datagram to (remote) address	✓
UDP	`recvfrom`	Receive a datagram from (remote) address	✓
TCP	`listen`	Mark socket as passive, i.e., for incoming connections	×
TCP	`accept`	Wait for a connection to be established	✓
TCP	`connect`	Actively establish a connection with (remote) address	✓
TCP	`send`	Transmit a segment via connection	✓
TCP	`recv`	Receive a segment via connection	✓
	`select` `poll`	Wait for activity that would allow non-blocking access	✓

Notes:

- The Linux implementation of the socket API is well documented: see, for example,

```
man -s 2 bind
```

  and so on. Note that some of the functions listed aren't strictly part of the (standard) API: rather, they represent "helper" functions which make using it easier.
- You may see `gethostbyname` or `gethostbyaddr` used in example code; these are deprecated analogues of `getnameinfo` and `getaddrinfo`, which are now preferred.
- The fact *some* functions are blocking versus non-blocking means it is important not to misunderstand their name: `recvfrom`, for example, should be read as "register to receive data some time in the future" rather than "receive data *now*".
- Strictly speaking use of `connect` within the context of UDP can make sense: rather than establish a connection, it simply specifies the default address for subsequent `sendto` and `recvfrom` invocations.
- You can think about the socket API via analogy with making telephone calls:
  - `socket` is like obtaining a telephone to use,
  - `bind` is like making your telephone number available st. incoming callers know it,
  - `listen` is like plugging in the telephone, and turning on ringer so you know when an incoming call occurs,
  - `connect` is like placing a call using the destinations number,
  - `accept` is like picking up the receiver to answer incoming call, with caller ID st. the destination knows the source number,
  - `close` is like putting down receiver to terminate an active call.

  In the same context, you could think about DNS as being analogous to the telephone book: it allows one to look a number based on a description.

POSIX sockets API (1) – The Interface

Function	Description
`getnameinfo`	Convert an internal, machine-readable data structure into a host name
`getaddrinfo`	Convert a host name into an internal, machine-readable data structure
`inet_aton`	Convert a dotted-decimal address into a binary, machine-readable address
`inet_ntoa`	Convert a binary, machine-readable address into a dotted-decimal address
`htons/htons`	Convert a 16/32-bit host order integer into network order
`ntohs/ntohs`	Convert a 16/32-bit network order integer into host order

Notes:

- The Linux implementation of the socket API is well documented: see, for example,

```
man -s 2 bind
```

  and so on. Note that some of the functions listed aren't strictly part of the (standard) API: rather, they represent "helper" functions which make using it easier.
- You may see `gethostbyname` or `gethostbyaddr` used in example code; these are deprecated analogues of `getnameinfo` and `getaddrinfo`, which are now preferred.
- The fact *some* functions are blocking versus non-blocking means it is important not to misunderstand their name: `recvfrom`, for example, should be read as "register to receive data some time in the future" rather than "receive data *now*".
- Strictly speaking use of `connect` within the context of UDP can make sense: rather than establish a connection, it simply specifies the default address for subsequent `sendto` and `recvfrom` invocations.
- You can think about the socket API via analogy with making telephone calls:
  - `socket` is like obtaining a telephone to use,
  - `bind` is like making your telephone number available st. incoming callers know it,
  - `listen` is like plugging in the telephone, and turning on ringer so you know when an incoming call occurs,
  - `connect` is like placing a call using the destinations number,
  - `accept` is like picking up the receiver to answer incoming call, with caller ID st. the destination knows the source number,
  - `close` is like putting down receiver to terminate an active call.

  In the same context, you could think about DNS as being analogous to the telephone book: it allows one to look a number based on a description.

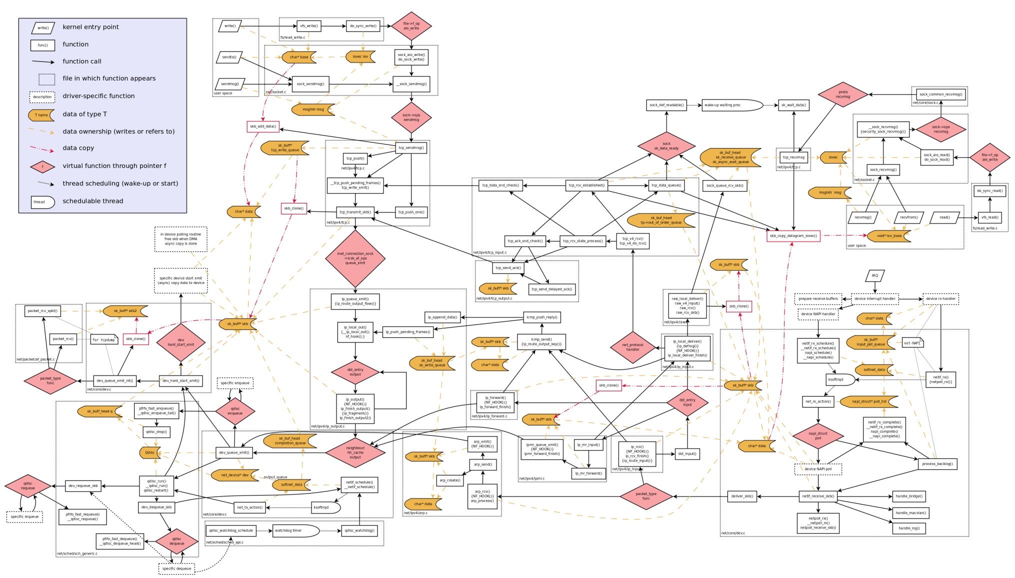
POSIX sockets API (2) – An Implementation (in Linux)

- Some (rough) design goals might include
  1. offer POSIX-compliant interface,
  2. offer RFC-compliant implementation,
  3. maximise efficiency (e.g., low-latency, effective use of bandwidth),
  4. maximise flexibility (e.g., general- not special-purpose),
  5. allow configurability,
  6. ...

  which lead to some underlying golden rules, e.g.,
  - make use of all possible hardware support,
  - make use of effective data structures,
  - minimise copying,
  - optimise for common-case,
  - ensure correctness for corner-cases,
  - ...

Notes:

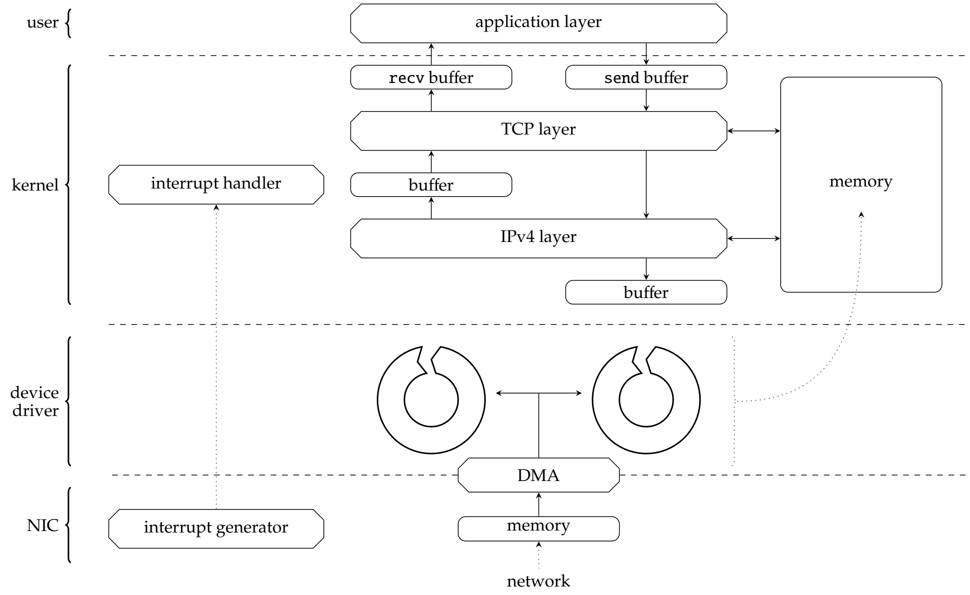
POSIX sockets API (3) – An Implementation (in Linux)



http://www.linuxfoundation.org/collaborate/workgroups/networking/kernel_flow

Notes:

POSIX sockets API (4) – An Implementation (in Linux)



Notes:

- A challenge wrt. describing "the" Linux implementation is that it is continually evolving! On one hand, this is positive: it continues to improve so as to meet new demands and use-case. On the other hand, the moving target makes it harder to offer an accurate, general overview.
  Various documents try to do so, however, at various levels of detail. A good example is

  http://datatag.web.cern.ch/datatag/papers/tr-datatag-2004-1.pdf

  which stems from an EU-funded research project.

POSIX sockets API (5) – An Implementation (in Linux)

- Fact: ports are basically buffers within network stack.
- Implication #1:
  - packets and segments might be received out-of-order, *but*
  - buffering enforces in-order delivery to the application.

Notes:

- The "some sort of time-out timer" might seem a little vague, but we've already encountered a concrete example: Nagle's algorithm can be seen as triggering transmission in this way.

POSIX sockets API (5) – An Implementation (in Linux)

- Fact: ports are basically buffers within network stack.
- Implication #2: `send` and transmission are decoupled ...
- ... transmission *could* occur
  1. when a complete segment is accumulated, or
  2. when transmission is forced, e.g., via
     - use of the PSH flag, or
     - some sort of time-out timer

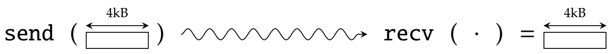
  so basically needs to realise a trade-off:
  - less efficient wrt. latency (wait more time) but more efficient wrt. bandwidth (transmit complete segments more often), or
  - more efficient wrt. latency (wait less time) but less efficient wrt. bandwidth (transmit complete segments less often).

Notes:

- The "some sort of time-out timer" might seem a little vague, but we've already encountered a concrete example: Nagle's algorithm can be seen as triggering transmission in this way.

POSIX sockets API (5) – An Implementation (in Linux)

- Fact: ports are basically buffers within network stack.
- Implication #3: `send` and `recv` are decoupled ...
- ... any one of



is possible.

Notes:

- The "some sort of time-out timer" might seem a little vague, but we've already encountered a concrete example: Nagle's algorithm can be seen as triggering transmission in this way.

Daniel Page (daniel.page@bristol.ac.uk)
Concurrent Computing (Computer Networks) git # 3627080 @ 2016-03-11 University of BRISTOL

POSIX sockets API (5) – An Implementation (in Linux)

- Fact: ports are basically buffers within network stack.
- Implication #3: `send` and `recv` are decoupled ...
- ... any one of

`send` (4kB) ⇝ `recv` (·) = 2kB
`recv` (·) = 2kB

is possible.

Notes:

- The "some sort of time-out timer" might seem a little vague, but we've already encountered a concrete example: Nagle's algorithm can be seen as triggering transmission in this way.

POSIX sockets API (5) – An Implementation (in Linux)

- Fact: ports are basically buffers within network stack.
- Implication #3: `send` and `recv` are decoupled ...
- ... any one of

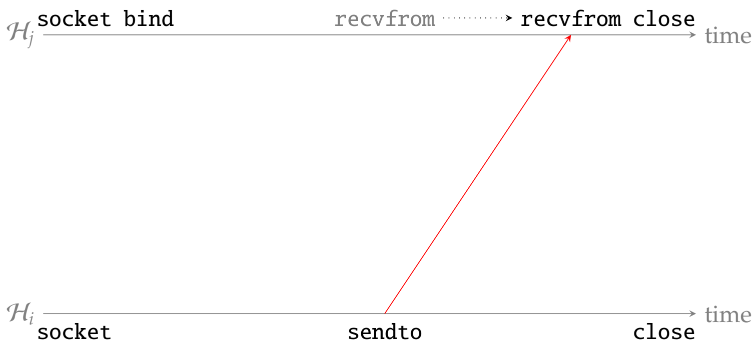
`send` (4kB) ⇝ `recv` (·) = 1kB
`recv` (·) = 1kB
`recv` (·) = 1kB
`recv` (·) = 1kB

is possible.

Notes:

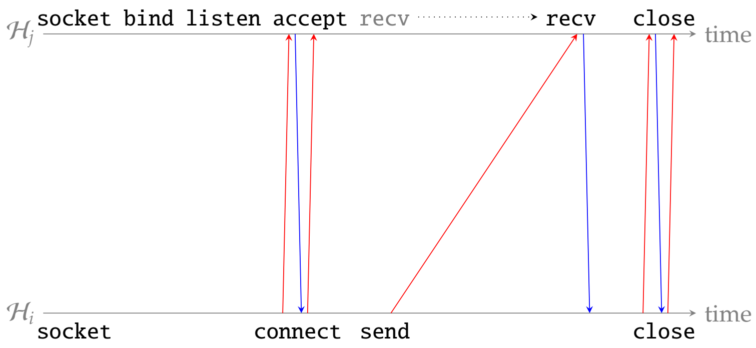
- The "some sort of time-out timer" might seem a little vague, but we've already encountered a concrete example: Nagle's algorithm can be seen as triggering transmission in this way.

Using POSIX sockets (1) – UDP



Notes:

Using POSIX sockets (2) – TCP



Notes:

Using POSIX sockets (3) – An “echo uppercase” TCP client/server

Listing (C)

```c
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>

void handle( int cs ) {
  char t[ 1024 ];

  while( true ) {
    // terminal -> t
    fgets( t, 1024, stdin );
    // server   <- t
    send( cs, t, strlen( t ), 0 );
    // server   -> t'
    t[ recv( cs, t, 1023, 0 ) ] = '\0';
    // terminal <- t'
    fputs( t, stdout );
  }

  // close connection
  close( cs );
}
```

Listing (C)

```c
int main( int argc, char* argv[] ) {
  struct sockaddr_in sa; socklen_t sl = sizeof( sa );
  struct sockaddr_in ca; socklen_t cl = sizeof( ca );

  memset( &sa, 0, sl );

  sa.sin_family      = AF_INET;
  sa.sin_addr.s_addr = inet_addr( argv[ 1 ] );
  sa.sin_port        = htons( atoi( argv[ 2 ] ) );

  // open   socket
  int cs = socket( AF_INET, SOCK_STREAM, 0 );
  // open   connection
  connect( cs, ( struct sockaddr* )( &sa ), sl );
  // handle connection
  handle( cs );

  return 0;
}
```

Notes:

- This code has a number of caveats and limitations; some compromises have been made to fit it on the slide(s), so try to treat it as an example rather than a reference.
  - There is no error checking *at all*, which is clearly not ideal! More or less *every* function, `bind` for example, returns an error code that *should* be checked carefully; the global variable `errno` captures more detailed information about any error that occurs.
  - Both the client and server loop indefinitely, i.e., neither allow a way for the user to terminate the connection without forcibly terminating the associated process: this might not be a great design strategy in general.
  - It should *really* use a type cast around `&ca`, i.e., use `( struct sockaddr* )( &ca )`, when calling `accept`.
    This highlights a subtle design choice, stemming in part from how C structures work. If you look at the definitions of `sockaddr` and `sockaddr_in`, it's clear you can cast the latter into the former: they share the same first field, so in a sense `sockaddr` is just a more general version of `sockaddr_in`.
  - An important issue with the code relates to `send`: it doesn't in fact *guarantee* to transmit all the data you give to it as input. As a result, it is common to use an auxiliary function of the form

```c
void sendall( int s, uint8_t* x, int n ) {
  int l = n;

  while( n > 0 ) {
    if( ( n -= send( s, x + l - n, n, 0 ) ) < 1 ) {
      break;
    }
  }
}
```

    to give a more useful outcome.
  - The IP address and port are specified via the command-line, but there are alternatives. For example, the following

IP address	Port	Semantics
hard-coded, `INADDR_ANY`	hard-coded, zero	kernel chooses IP address, kernel chooses port
hard-coded, `INADDR_ANY`	user-supplied, non-zero	kernel chooses IP address, user specifies port
user-supplied	hard-coded zero	user specifies IP address, kernel chooses port
user-supplied	user-supplied, non-zero	user specifies IP address, user specifies port

    shows one can defer to the kernel to some extent. Even then, however, IP addresses are demanded over domain names: it is of course possible to perform DNS resolution via the (local) resolver with appropriate additions.

Using POSIX sockets (3) – An “echo uppercase” TCP client/server

Listing (C)

```c
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>

void handle( int cs ) {
  char t[ 1024 ];



  while( true ) {
    // client -> t
    t[ recv( cs, t, 1023, 0 ) ] = '\0';
    // t' = toupper( t )
    for( int i = 0; i < strlen( t ); i++ ) {
      t[ i ] = toupper( t[ i ] );
    }
    // client <- t'
    send( cs, t, strlen( t ), 0 );
  }

  // close connection
  close( cs );


}
```

Listing (C)

```c
int main( int argc, char* argv[] ) {
  struct sockaddr_in sa; socklen_t sl = sizeof( sa );
  struct sockaddr_in ca; socklen_t cl = sizeof( ca );

  memset( &sa, 0, sl );

  sa.sin_family      = AF_INET;
  sa.sin_addr.s_addr = inet_addr( argv[ 1 ] );
  sa.sin_port        = htons( atoi( argv[ 2 ] ) );

  // open   socket
  int ss = socket( AF_INET, SOCK_STREAM, IPPROTO_IP );
  // bind   socket
  bind( ss, ( struct sockaddr* )( &sa ), sl );
  // listen for connections
  listen( ss, 10 );

  while( true ) {


    // open   connection
    int cs = accept( ss, &ca, &cl );
    // handle connection
    handle( cs );
  }

  // close socket
  close( ss );

  return 0;
}
```

Notes:

- This code has a number of caveats and limitations; some compromises have been made to fit it on the slide(s), so try to treat it as an example rather than a reference.
  - There is no error checking *at all*, which is clearly not ideal! More or less *every* function, `bind` for example, returns an error code that *should* be checked carefully; the global variable `errno` captures more detailed information about any error that occurs.
  - Both the client and server loop indefinitely, i.e., neither allow a way for the user to terminate the connection without forcibly terminating the associated process: this might not be a great design strategy in general.
  - It should *really* use a type cast around `&ca`, i.e., use `( struct sockaddr* )( &ca )`, when calling `accept`.
    This highlights a subtle design choice, stemming in part from how C structures work. If you look at the definitions of `sockaddr` and `sockaddr_in`, it's clear you can cast the latter into the former: they share the same first field, so in a sense `sockaddr` is just a more general version of `sockaddr_in`.
  - An important issue with the code relates to `send`: it doesn't in fact *guarantee* to transmit all the data you give to it as input. As a result, it is common to use an auxiliary function of the form

```c
void sendall( int s, uint8_t* x, int n ) {
  int l = n;

  while( n > 0 ) {
    if( ( n -= send( s, x + l - n, n, 0 ) ) < 1 ) {
      break;
    }
  }
}
```

    to give a more useful outcome.
  - The IP address and port are specified via the command-line, but there are alternatives. For example, the following

IP address	Port	Semantics
hard-coded, `INADDR_ANY`	hard-coded, zero	kernel chooses IP address, kernel chooses port
hard-coded, `INADDR_ANY`	user-supplied, non-zero	kernel chooses IP address, user specifies port
user-supplied	hard-coded zero	user specifies IP address, kernel chooses port
user-supplied	user-supplied, non-zero	user specifies IP address, user specifies port

    shows one can defer to the kernel to some extent. Even then, however, IP addresses are demanded over domain names: it is of course possible to perform DNS resolution via the (local) resolver with appropriate additions.

Using POSIX sockets (3) – An "echo uppercase" TCP client/server

Listing (C)

```c
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>

void* handle( void* __cs ) {
  char t[ 1024 ];

  int cs = *( int* )( __cs );

  while( true ) {
    // client -> t
    t[ recv( cs, t, 1023, 0 ) ] = '\0';
    // t' = toupper( t )
    for( int i = 0; i < strlen( t ); i++ ) {
      t[ i ] = toupper( t[ i ] );
    }
    // client <- t'
    send( cs, t, strlen( t ), 0 );
  }

  // close connection
  close( cs );

  return NULL;
}
```

Listing (C)

```c
int main( int argc, char* argv[] ) {
  struct sockaddr_in sa; socklen_t sl = sizeof( sa );
  struct sockaddr_in ca; socklen_t cl = sizeof( ca );

  memset( &sa, 0, sl );

  sa.sin_family      = AF_INET;
  sa.sin_addr.s_addr = inet_addr( argv[ 1 ] );
  sa.sin_port        = htons( atoi( argv[ 2 ] ) );

  // open   socket
  int ss = socket( AF_INET, SOCK_STREAM, IPPROTO_IP );
  // bind   socket
  bind( ss, ( struct sockaddr* )( &sa ), sl );
  // listen for connections
  listen( ss, 10 );

  while( true ) {
    pthread_t id;

    // open   connection
    int cs = accept( ss, &ca, &cl );
    // handle connection
    pthread_create( &id, NULL, &handle, &cs );
  }

  // close socket
  close( ss );

  return 0;
}
```

Notes:

- This code has a number of caveats and limitations; some compromises have been made to fit it on the slide(s), so try to treat it as an example rather than a reference.
  - There is no error checking *at all*, which is clearly not ideal! More or less *every* function, `bind` for example, returns an error code that *should* be checked carefully; the global variable `errno` captures more detailed information about any error that occurs.
  - Both the client and server loop indefinitely, i.e., neither allow a way for the user to terminate the connection without forcibly terminating the associated process: this might not be a great design strategy in general.
  - It should *really* use a type cast around `&ca`, i.e., use `( struct sockaddr* )( &ca )`, when calling `accept`. This highlights a subtle design choice, stemming in part from how C structures work. If you look at the definitions of `sockaddr` and `sockaddr_in`, it's clear you can cast the latter into the former: they share the same first field, so in a sense `sockaddr` is just a more general version of `sockaddr_in`.
  - An important issue with the code relates to `send`: it doesn't in fact *guarantee* to transmit all the data you give to it as input. As a result, it is common to use an auxiliary function of the form

```c
void sendall( int s, uint8_t* x, int n ) {
  int l = n;

  while( n > 0 ) {
    if( ( n -= send( s, x + l - n, n, 0 ) ) < 1 ) {
      break;
    }
  }
}
```

    to give a more useful outcome.
  - The IP address and port are specified via the command-line, but there are alternatives. For example, the following

IP address	Port	Semantics
hard-coded, `INADDR_ANY`	hard-coded, zero	kernel chooses IP address, kernel chooses port
hard-coded, `INADDR_ANY`	user-supplied, non-zero	kernel chooses IP address, user specifies port
user-supplied	hard-coded zero	user specifies IP address, kernel chooses port
user-supplied	user-supplied, non-zero	user specifies IP address, user specifies port

    shows one can defer to the kernel to some extent. Even then, however, IP addresses are demanded over domain names: it is of course possible to perform DNS resolution via the (local) resolver with appropriate additions.

Using POSIX sockets (4) – An "echo uppercase" TCP client/server

Listing (Python)

```python
import socket, sys

def handle( cs ) :
  while( True ) :
    # terminal -> t
    t = sys.stdin.readline()
    # server   <- t
    cs.send( t )
    # server   -> t'
    t = cs.recv( 1024 )
    # terminal <- t'
    sys.stdout.write( t )

if( __name__ == "__main__" ) :
  host =      sys.argv[ 1 ]
  port = int( sys.argv[ 2 ] )

  # open   socket
  cs = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
  # open   connection
  cs.connect( ( host, port ) )
  # handle connection
  handle( cs )
  # close  connection
  cs.close()
```

Notes:

Listing (Python)

```
import socket, sys

def handle( cs, ca ) :
  while( True ) :
    # client -> t
    t = cs.recv( 1024 )
    # t' = toupper( t )
    t = t.upper()
    # client <- t'
    cs.send( t )

  # close connection
  cs.close()

if( __name__ == "__main__" ) :
  host =      sys.argv[ 1 ]
  port = int( sys.argv[ 2 ] )

  # open  socket
  ss = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
  # bind  socket
  ss.bind( ( host, port ) )
  # listen for connections
  ss.listen( 10 )

  while( True ) :
    # open   connection
    ( cs, ca ) = ss.accept()
    # handle connection
    handle( cs, ca )

  # close socket
  ss.close()
```

Notes:

Listing (Python)

```
import socket, sys, thread

def handle( cs, ca ) :
  while( True ) :
    # client -> t
    t = cs.recv( 1024 )
    # t' = toupper( t )
    t = t.upper()
    # client <- t'
    cs.send( t )

  # close connection
  cs.close()

if( __name__ == "__main__" ) :
  host =      sys.argv[ 1 ]
  port = int( sys.argv[ 2 ] )

  # open  socket
  ss = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
  # bind  socket
  ss.bind( ( host, port ) )
  # listen for connections
  ss.listen( 10 )

  while( True ) :
    # open   connection
    ( cs, ca ) = ss.accept()
    # handle connection
    thread.start_new_thread( handle, ( cs, ca ) )

  # close socket
  ss.close()
```

Notes:

Listing (Python)

```
import SocketServer, sys

class server( SocketServer.BaseRequestHandler ):
  def handle( self ) :
    while( True ) :
      # client -> t
      t = self.request.recv( 1024 )
      # t' = toupper( t )
      t = t.upper()
      # client <- t'
      self.request.sendall( t )

if( __name__ == "__main__" ) :
  host =      sys.argv[ 1 ]
  port = int( sys.argv[ 2 ] )

  SocketServer.TCPServer( ( host, port ), server ).serve_forever()
```

Notes:

Conclusions

- Take away points:
  - Ultimately, the POSIX sockets API is an abstraction of the network ...
  - ... even so, it's hard to argue you can totally avoid having to understand the underlying technology.
  - As with any design, it has good and bad features: for example,
    - it offers a uniform interface to analogous concepts (cf. **domain sockets**, for IPC),
    - it allows special-case implementation choices such as use of **TCP offload**,
    - the abstraction offered is still low-level so can be hard to use (directly),
    - numerous requirements have changed over time (e.g., network vs. host order, new protocols, new use-cases), but the API hasn't,
    - ...

Notes:

[1] Wikipedia: Berkeley sockets.
http://en.wikipedia.org/wiki/Berkeley_sockets.

[2] Wikipedia: Raw socket.
http://en.wikipedia.org/wiki/Raw_socket.

[3] Wikipedia: TCP offload engine.
http://en.wikipedia.org/wiki/TCP_offload_engine.

[4] Wikipedia: UNIX domain socket.
http://en.wikipedia.org/wiki/Unix_domain_socket.

[5] Wikipedia: Winsock.
http://en.wikipedia.org/wiki/Winsock.

[6] Standard for information technology - portable operating system interface (POSIX).
Institute of Electrical and Electronics Engineers (IEEE) 1003.1, 2008.
http://standards.ieee.org/findstds/standard/1003.1-2008.html.

[7] W.R. Stevens, B. Fenner, and A.M. Rudoff.
UNIX Network Programming Volume 1: The Sockets Networking API.
Addison Wesley, 3rd edition, 2003.

Notes: