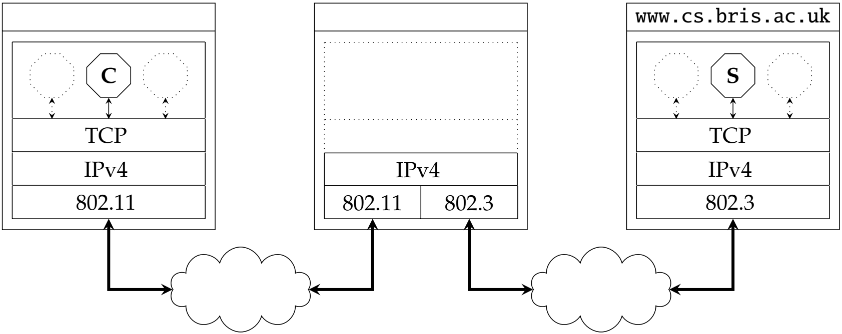
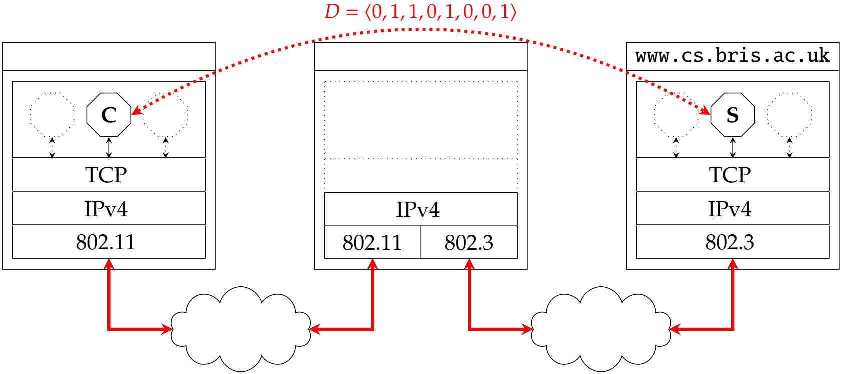


- Goal: *finally* investigate the **application layer**, e.g.,
  - the (mainly OS-based) network stack implementation,
  - the interface between application and network stack, i.e.,

    1. a **raw socket**, or
    2. the **POSIX sockets** API,

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



$F = H_{802.11} \parallel H_{IPv4} \parallel H_{TCP} \parallel \langle 0, 1, 1, 0, 1, 0, 0, 1 \rangle \parallel T_{802.11}$

- 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!

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	✓

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

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

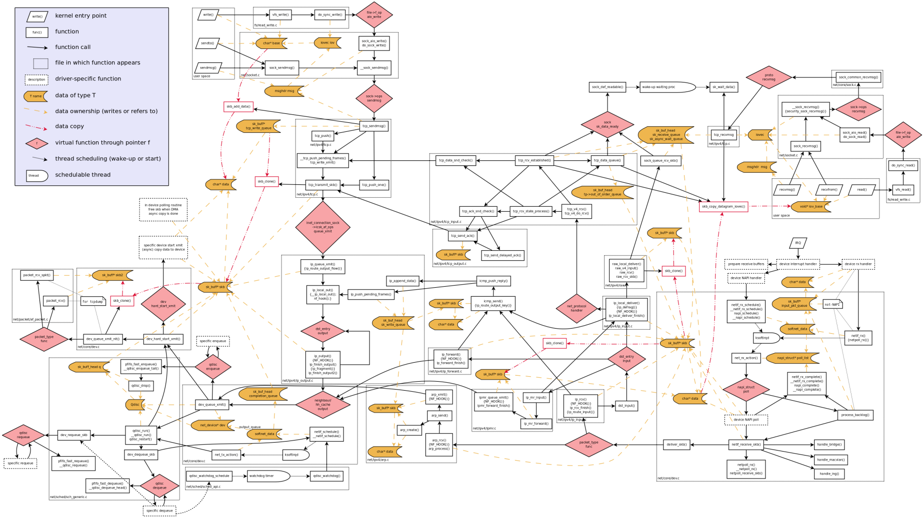
- Some (rough) design goals might include

  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.,

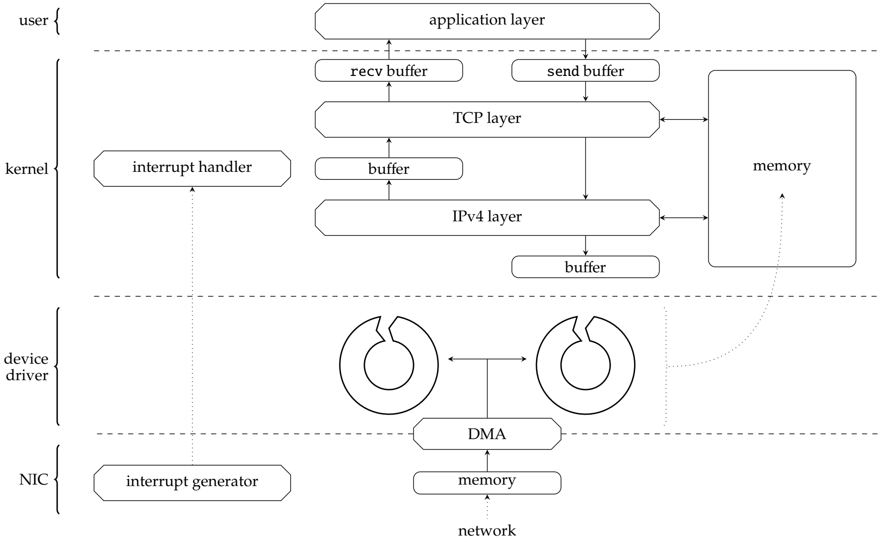
  - make use of all possible hardware support,
  - make use of effective data structures,
  - minimise copying,
  - optimise for common-case,
  - ensure correctness for corner-cases,
  - ...

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



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

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



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.

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).

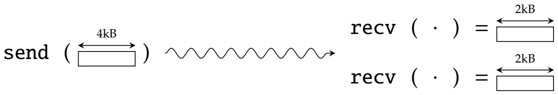
POSIX sockets API (5) – An Implementation (in Linux)

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

is possible.

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

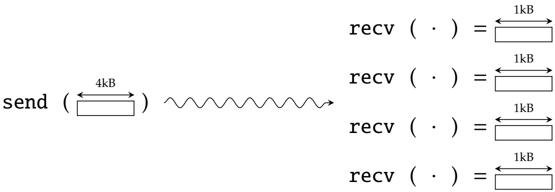
- Fact: ports are basically buffers within network stack.
- Implication #3: `send` and `recv` are decoupled ...
- ... any one of



is possible.

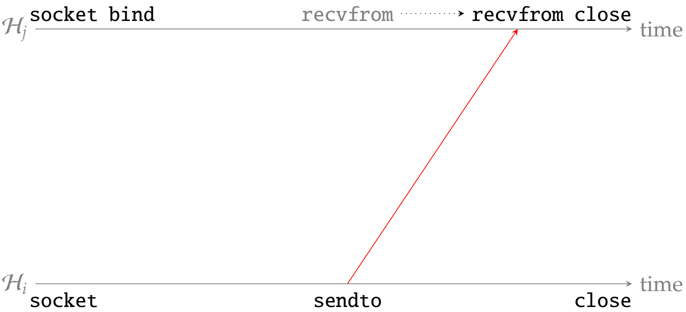
POSIX sockets API (5) – An Implementation (in Linux)

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

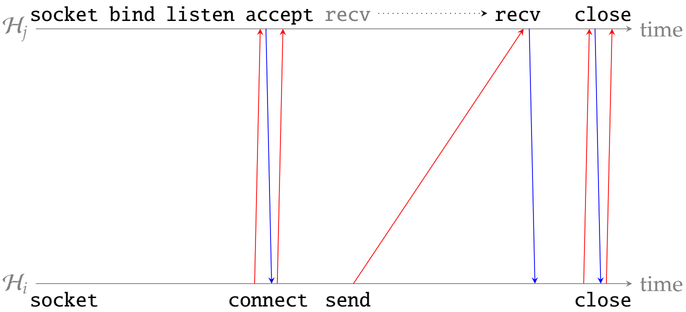


is possible.

Using POSIX sockets (1) – UDP



Using POSIX sockets (2) – TCP



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;
}
```

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;
}
```

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;
}
```

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,
    - ...

References

[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.