Programming

G54ACC

Lecture 16

richard.mortier@nottingham.ac.uk

Recap

- Basics of Berkeley Sockets API
  - socket(), connect(), send(), recv(), &c
- Endianness
- Multiplexing at lower layers
  - We'll discuss multiplexing from the programmer point of view
- Remote Procedure Call (RPC)
  - Invocation of code on a remote machine

Contents

- Berkeley Sockets
- Web RPC
- RESTful web services
- Flavours of “cloud computing”
- Summary

Contents

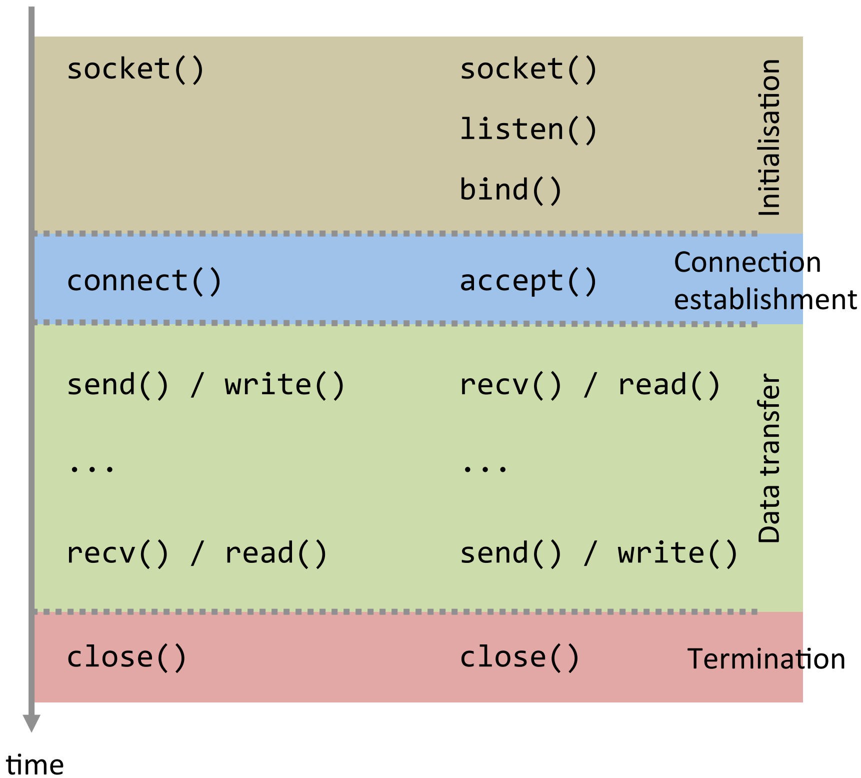
- Berkeley Sockets
  - TCP/UDP
  - Endianness
  - Multiplexing
- Web RPC
- RESTful web services
- Flavours of “cloud computing”
- Summary

Sockets

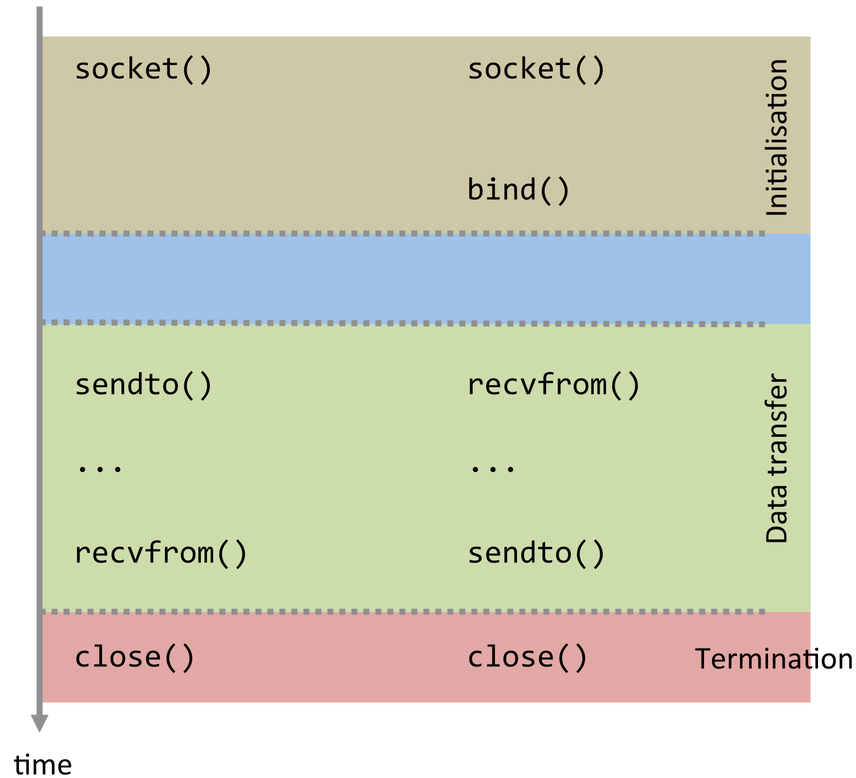
- Berkeley Sockets API, 4.2BSD (1983)
  - Use *Internet sockets*, presented as file descriptors
  - Extended to many other protocols
- `int` **`socket`**`(int` *`domain`*`, int` *`type`*`, int` *`protocol`*`)`
  - domain: communication domain
    - `PF_INET`, `PF_INET6`, `PF_LOCAL`, `PF_RAW`
  - type: communication semantics
    - `SOCK_STREAM`, `SOCK_DGRAM`, `SOCK_RAW`
  - protocol: usu. 1:1 with (domain,type)
    - `IPPROTO_ICMP`, `IPPROTO_TCP`, `IPPROTO_UDP`

TCP *vs.* UDP

- File-like interface
  - Streaming



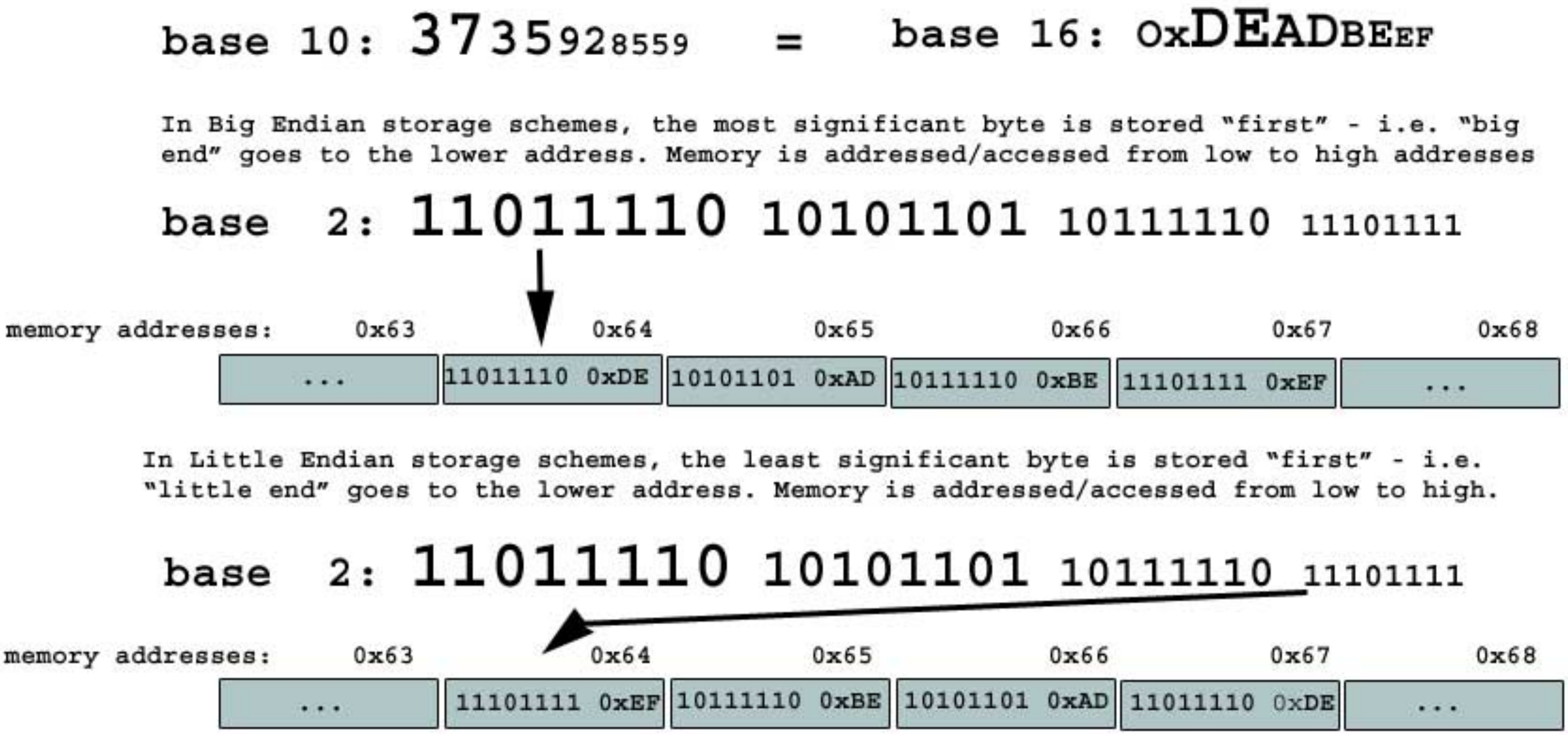
- Packet-like interface
  - Need to packetize



Endianness

- Host byte order need not be identical
  - Big endian (most-significant-byte first)
  - *vs*. Little endian (least-significant-byte first)
- Need to decide which we use on the network
  - Lest 0x0001 be received as 0x0100
  - Or negotiate every time?!
- Network byte order == big endian
  - No reason. It just is.
  - Convert via `u[16|32]` **`hton[s|l]/ntoh[s|l]`**`(u[16|32])`

http://flickeringtubelight.net/blog/2004/05/big-endian-and-little-endian-storage-schemes-how-to-remember/



Multiplexing

- Naive: read and block
  - But what if you need to handle >1 socket?
- Naive 2: poll non-blocking socket
  - Wastes CPU
- Use select(), either blocking or non-blocking
  - Enables waiting (until timeout) on any socket of a set becoming active
  - `int select( int nfds, fd_set *rfds, fd_set *wfds, fd_set *efds, struct timeval timeout)`
- Check W. Richard Stevens "*Unix Network Programming*" and "*TCP/IP vol.1*" for details

Contents

- Berkeley Sockets
- Web RPC
  - XML-RPC
  - SOAP
- RESTful web services
- Flavours of “cloud computing”
- Summary

Web RPC Systems

- (At least) Two: XML-RPC *vs*. SOAP
  - Appears to be/have been a religious war here
  - SOAP evolved from XML-RPC
- Both provide
  - Data encoded as XML
  - *Remote Procedure Call* interfaces
  - ...to remote web services

XML-RPC

- Older, simpler
  - http://www.xmlrpc.com/spec
- Community based

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 158
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:08 GMT
Server: UserLand Frontier/5.1.2-WinNT


<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <struct>
        <member>
          <name>lowerBound</name>
            <value><i4>18</i4></value>
        </member>
        <member>
          <name>upperBound</name>
          <value><i4>139</i4></value>
        </member>
      </struct>
    </param>
  </params>
</methodResponse>
```

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: betty.userland.com
Content-Type: text/xml
Content-length: 181


<?xml version="1.0"?>
<methodCall>
   <methodName>examples.getBounds</methodName>
   <params>
      <param>
         <value><i4>41</i4></value>
         </param>
      </params>
   </methodCall>
```

SOAP

- Newer, more featureful, more flexible
  - Doesn't require HTTP as transport
  - Provides e.g., discovery (WSDL), &c
- Not particularly simple
- W3C standard

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml;
charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/
soap-envelope"
soap:encodingStyle="http://www.w3.org/
2001/12/soap-encoding">

<soap:Body xmlns:m="http://
www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
```

Contents

- Berkeley Sockets
- Web RPC
- RESTful web services
  - REST constraints and interfaces
  - Creating a RESTful service
- Flavours of "cloud computing"
- Summary

RESTful Web Service

- Roy Fielding during HTTP/1.1 development (1994–2002)
  - doi:10.1145/514183.514185
  - http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- Goals
  - Minimize latency, communication
  - Maximise implementation scalability, component independence

RESTful Web Service

- A RESTful service conforms to the REST constraints
  - Representational State Transfer
  - Makes more extensive use of HTTP features
    - In principle can use other transports
- Client requests generate server responses
- Both built around *representations* of *resources*
  - HTML, JPEG, whatever in principle. JSON or XML in practice.

REST Constraints

- Six architectural constraints
  - Client-server, Cacheable, Layered
    - Separate concerns, beware existence of caches, proxies, &c
  - Code on demand (optional)
    - Server can push e.g., JavaScript to client as part of response
  - Stateless, Uniform interface
    - …

Stateless

- Does *not* mean the server must be stateless!
- Simply that no *client* context is stored on the server between requests
  - Each client request contains enough to service it
  - If server does store state, it is addressable as a resource via URL in the usual way
- Contrast

```
GET /resources/nextPage?

...

<?xml version="1.0"?>
<rsp status="ok">
  <resource id="1" />
  <resource id="2" />
</rsp>
```

```
GET /resources/?page=2

...

<?xml version="1.0"?>
<rsp page="2" nextPage="3">
  <resource id="21" />
  <resource id="22" />
</rsp>
```

Uniform Interface

- Client uses representation to modify server state, addressed via URL
- RESTful service maps CRUD to HTTP methods
  - CREATE: POST
  - READ/RETRIEVE: GET, idempotent, side-effect free
  - UPDATE: PUT, idempotent (since uses specified name)
  - DELETE: idempotent
- CREATE/UPDATE as POST *vs*. PUT appears debated
  - Differ in treatment of Request-URI and request entity
  - POST: treat entity as "new subordinate"
  - PUT: entity "stored under supplied Request-URI"

Creating a RESTful Service

- What are the URIs?
  - Find the nouns, define *collections*
- What's the format?
  - XML, JSON, &c
- What methods are supported at each URI?
  - Mapping to the CRUD functions
- What status codes could be returned?
  - 1xx (informational), 2xx (successful), 3xx (redirection), 4xx (client error), 5xx (server error)

Notable Points

- Collections
  - http://.../people : GET (list), POST (append), PUT (replace), DELETE
  - http://.../people/1 : GET (retrieve), PUT (update), DELETE
- Tickets used to provide *at-most-once* semantics
  - Unique, server-generated GUIDs
  - POST to /ticket/o to create a “slot” in the server
- Redirects help clients handle tickets
  - Ticket response redirects to /o/<n>
  - PUT /o/<n> creates new item, idempotently

Contents

- Berkeley Sockets
- Web RPC
- RESTful web services
- Flavours of “cloud computing”
- Summary

Cloud Computing

- Datacenter as a utility
  - Buy it (rent it) when you need it (OpEx *vs*. CapEx)
  - Scales “on demand”
- Beware!
  - Variable performance
  - Little control over data placement
  - Unknown trust implications
  - Cloud provider as single-point-of-failure
    - Not just their hosts, but their and your network connectivity
  - Building scalable services is still hard!

Flavours

- X as a Service for X =
  - Software: Salesforce
    - Provides you with a Customer Relationship Manager service
    - No longer need to manage software, upgrades, &c
  - Platform: Google App Engine, PiCloud, Heroku, Azure
    - Hosts web apps backed by shared store (SQL-like)
    - Fires up (Python, Java, Ruby, CLR) VM as required
  - Infrastructure: Amazon EC2, Rackspace
    - Provides XEN hosted VMs (Linux, Windows, custom)
    - Also persistent store (S3), other infrastructure APIs

Summary

- There are a wide range of APIs that attempt hide the complexity of network programming
  - Sockets is the (old) canonical Internet API on which all else is built
  - Web RPC was briefly popular
  - RESTful platforms with APIs are "web2.0" and all the rage (and jolly useful)
- Cloud computing in some ways goes a stage further, hiding even more of the complexity
  - But you introduce other problems due to, e.g., the scale of the application
- Failures are always a pain to deal with, anywhere