Application Protocols

- each application can define its own protocol
  - types of messages exchanged
  - syntax of messages: what fields, how they are separated
  - semantics of messages: meaning of each field
  - rules for how to process and respond to messages
- to encourage interoperability, the protocol should be standardized
  - IETF
  - W3C
- to discourage interoperability, create a proprietary protocol
  - AOL Instant Messenger
Reading a Message

- an application protocol must read and write messages
  - clients typically send a request and then read a response
  - servers typically read a request and then send a response
- TCP and the socket API treat messages as a string of bytes
  - the application has to designate where a message starts and ends
- two options for reading a message
  1. variable length: read until a sentinel (end-of-message marker)
  2. known length: read a length field and then read the listed number of bytes
Binary versus ASCII Protocols

- most network protocols exchange messages coded in binary
  - conserve space for small packets or expensive bandwidth
  - requires standardizing a byte-level format
  - must be careful about transmitting numeric values in network byte order
- application protocols exchange messages coded in ASCII
  - large messages, cheap bandwidth
  - easier to write, debug, extend
clients request objects from servers using the HTTP protocol
- client sends an HTTP request
- server sends an HTTP response
does not necessarily have a GUI
- text based
- a spider
- any other program (e.g. collecting hourly reports on competitor’s book prices)
Web Objects

- **Object names**: Uniform Resource Identifier (URI)
  - a name that refers to a resource
  - a Uniform Resource Locator (URL) is one type of URI
  - popular URL schemes: http, ftp, gopher, mailto

- **Object delivery**: Hypertext Transfer Protocol (HTTP)
  - IETF standard
  - defines message format for making requests and receiving responses

- **Object format**: Hypertext Markup Language (HTML)
  - representation of documents in ASCII format
  - many other formats - XHTML, XML, PNG, JPG, PDF, etc.
**URIs, URLs, and URNs**

- **URI**: Uniform Resource Identifier
  - The generic set of all names that refer to resources

- **URL**: Uniform Resource Locator
  - An informal term (no longer used in technical specifications) associated with popular URI schemes: http, ftp, gopher, mailto, etc.

- **URN**: Uniform Resource Name
  - A URI that has an institutional commitment to persistence, availability, etc. May also be a URL: see PURLs.
  - persistent, location-independent resource identifiers, urn: specified by RFC 2141

**http://www.w3.org/Addressing/**
Container Objects

- A web page consists of a container object, which may link to other objects.
- Fetching a web page consists of requesting the container object and then requesting any linked objects.
Handling Objects

- determines how responses are handled
  - appearance (fonts)
  - content transformations (language)
  - whether to accept cookies
  - whether to allow javascript, popups
  - MIME types and handlers
- see Firefox preferences
- Java versus ActiveX
  - sandbox versus trusted certificates
HTTP Standards

- HTTP 1.0
  - Informational: not intended to be a standard
  - very basic protocol, documenting what earliest servers and browsers used

- HTTP 1.1
  - Standards Track: either proposed standard, draft standard, or a full standard
  - backward compatibility with HTTP/1.0, plus many improvements and features
  - what all modern servers and browsers uses
Specification Language

- specification language is precise
  - **MUST**: absolutely essential - if you don’t implement this feature you are not compliant
  - **SHOULD**: recommendation - you are compliant if you don’t implement this feature, but you should implement it if at all possible
  - **MAY**: optional - not considered necessary
  - there are two obvious counterparts: **MUST NOT**, **SHOULD NOT**

- see RFC 2119
HTTP Request Format

- **request line**: method, URI, version
- **header lines**: additional method parameters, meta-data
- ends with a carriage return and line feed
- optional entity body, with a header that indicates the length of the body in bytes
Example HTTP Request

- **HTTP 1.0 Request:**

  1. GET /index.html HTTP/1.0
  2. User-Agent: Mozilla/5.0

- **HTTP 1.1 Request:**

  1. GET /index.html HTTP/1.1
  2. Host: ilab.cs.byu.edu
  3. User-Agent: Mozilla/5.0
HTTP Response Format

- **response line**: version, status code, status phrase
- **header lines**
- ends with a carriage return and line feed
- optional entity body, with a header that indicates the length of the body in bytes
Example HTTP Response

- HTTP Response:

```
HTTP/1.1 200 OK
Date: Thu, 10 Jan 2008 18:36:18 GMT
Server: Apache
Last-Modified: Thu, 12 Oct 2006 21:44:06 GMT
ETag: "588107-b26-121f9580"
Accept-Ranges: bytes
Content-Length: 2854
Content-Type: text/html

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
...
```

- use `telnet ilab.cs.byu.edu 80` to experiment
Send an HTTP Request

1. GET /index.html HTTP/1.1
2. Host: ilab.cs.byu.edu
3. User-Agent: Downloader/1.0

- Host header is required in HTTP/1.1
- User-Agent provides browser software version
Writing an Entire Message

```c
ptr = message.c_str();
nleft = strlen(ptr);
while (nleft) {
    if ((nwritten = send(client_, ptr, nleft, 0)) < 0) {
        if (errno == EINTR) {
            nwritten = 0;
        } else {
            perror("write");
            break;
        }
    } else if (nwritten == 0) {
        // the socket is closed
        break;
    }
    nleft -= nwritten;
    ptr += nwritten;
}```
Receive an HTTP Response

1. HTTP/1.1 200 OK
2. Date: Thu, 10 Jan 2008 18:36:18 GMT
3. Server: Apache
5. ETag: "588107-b26-121f9580"
6. Accept-Ranges: bytes
7. Content-Length: 2854
8. Content-Type: text/

1. read until you receive a CRLF
2. parse HTTP headers
3. use value of Content-Length header to determine the length of the entity body
4. read the number of bytes indicated
5. parse/display the HTTP object
Reading Until a Sentinel

1. initialize a string cache
2. loop forever
   - read data into a buffer
     - EINTR means continue
     - any other error means halt
     - 0 bytes means the socket is closed
   - append the data to the cache
   - if the buffer contains the sentinel, break from loop
3. return portion of cache prior to and including the sentinel, leave remaining portion in the cache
Reading a Known Message Length

1. initialize a string cache
2. initialize length
3. loop until length is zero
   - read data into a buffer
     - EINTR means continue
     - any other error means halt
     - 0 bytes means the socket is closed
   - append the data to the cache
   - decrement the length by amount read
4. return portion of cache up to the message length, leave remaining portion in the cache