Introduction to Computer Networks

COSC 4377

Lecture 2

Spring 2012

January 23, 2012

Announcements

- Several HW0 missing
- HW1 due this week
- HW2 released
- Re-assess if you found HW0/HW1 challenging

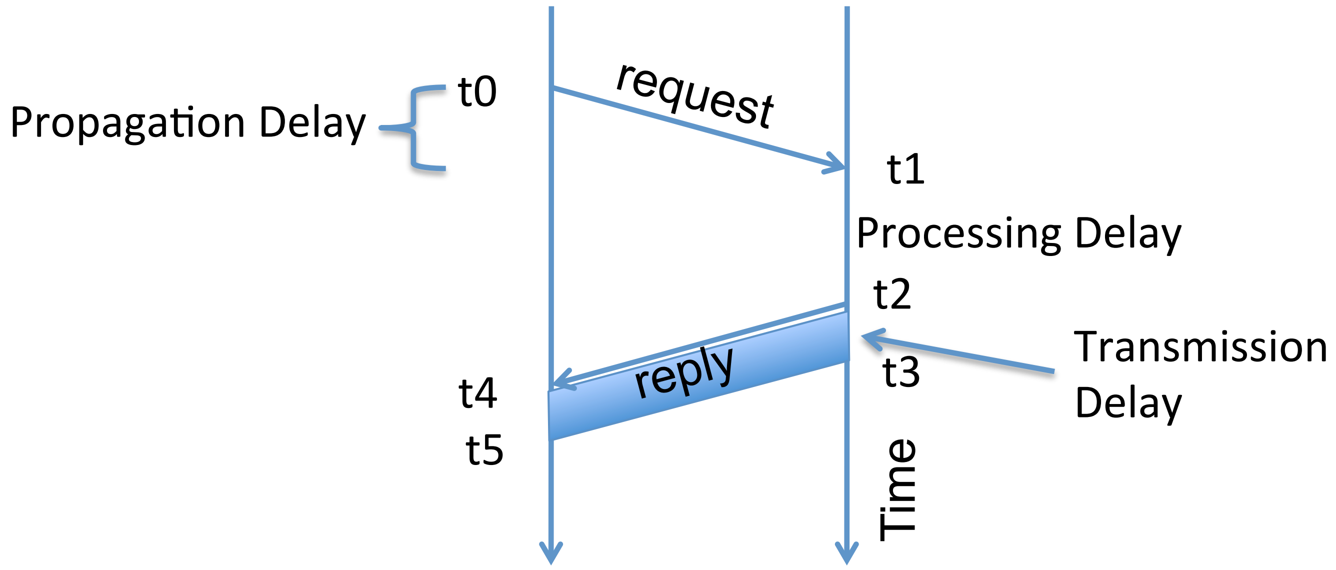
HW0

"You should submit a single file, the source code. No need to submit the binary. Please submit your source code through Blackboard. Please include a few lines of comments at the top of your file describing how to compile your file and if there is anything unusual about your implementation (e.g., it does not work with certain inputs)"

Other observations

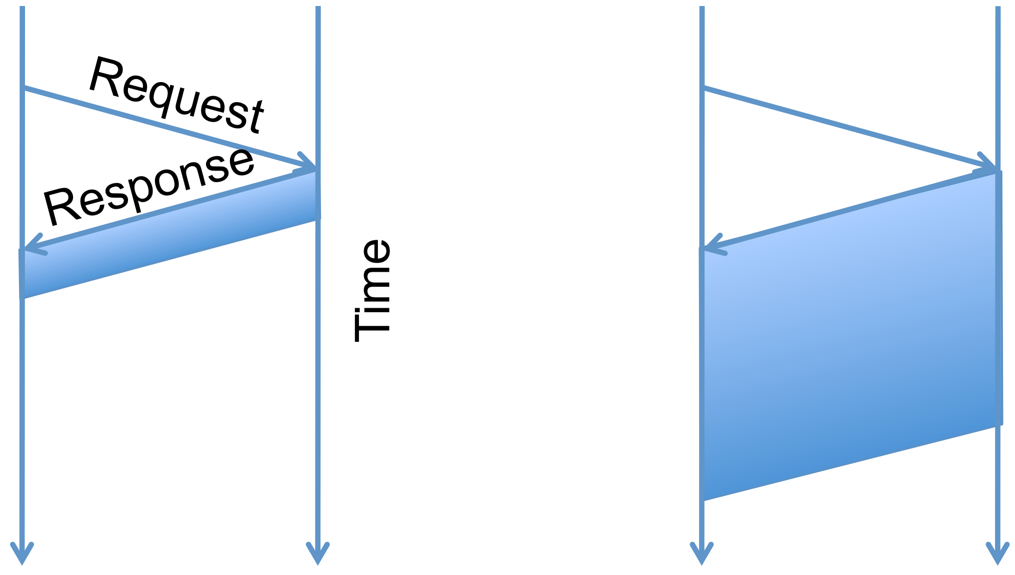
- Input sizes
- File/directory assumptions

Protocol Timing Diagram



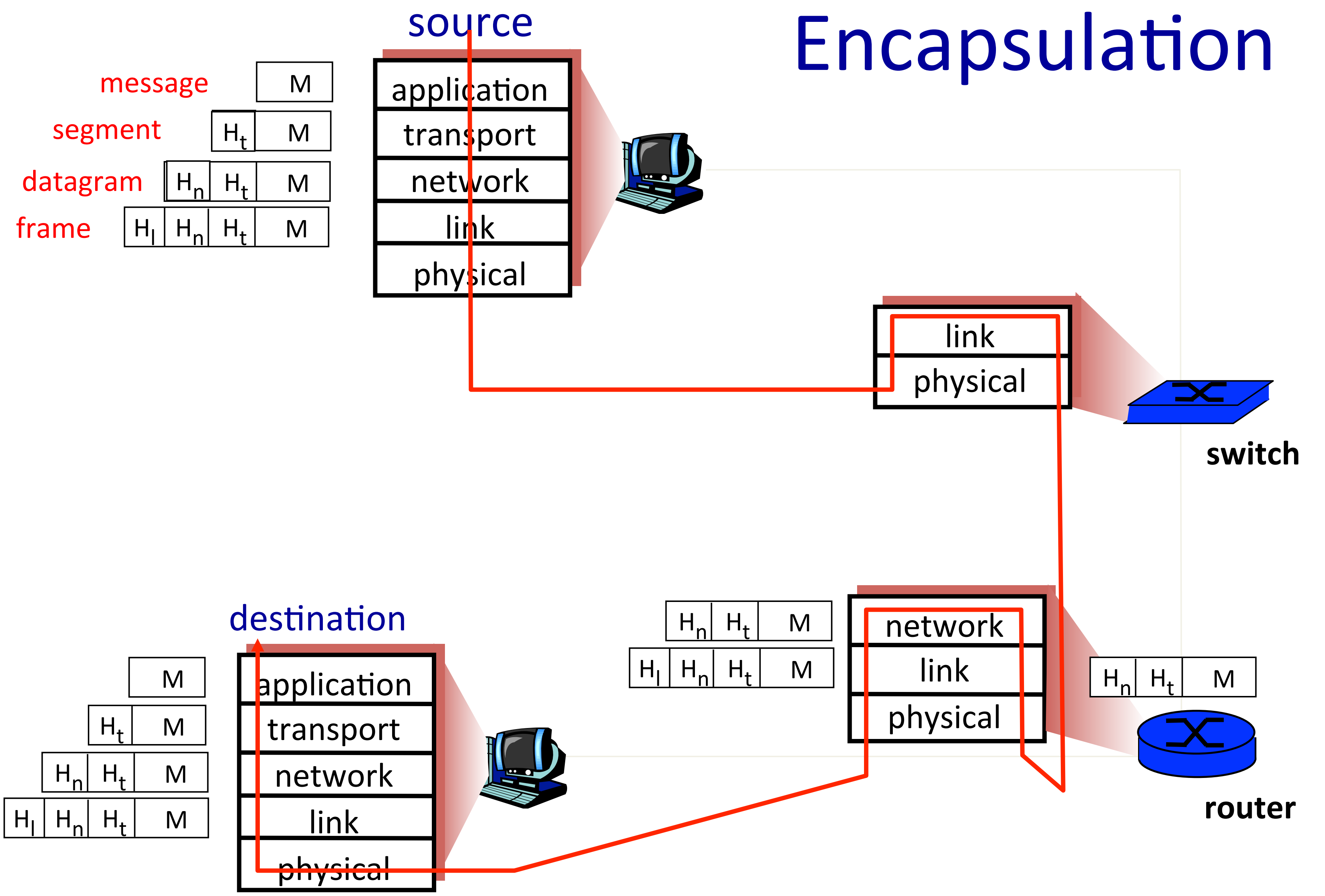
Bandwidth and Delay

- How much data can we send during one RTT?
- *E.g.*, send request, receive file



- **For small transfers, latency more important, for bulk, throughput more important**

Encapsulation



Today's Topics

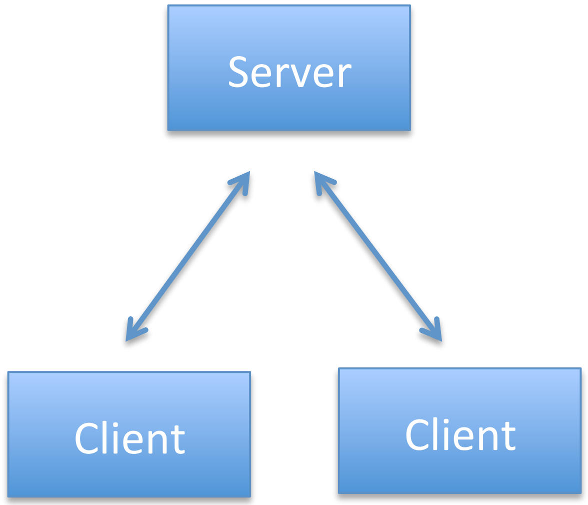
- Platform and Tools for HWs
- Network Applications
- Socket Programming

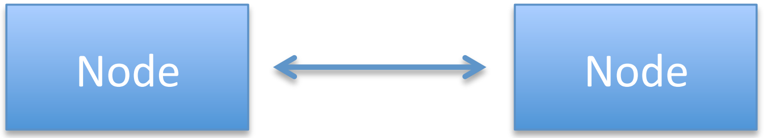
Platform and Tools for HWs

- We will compile/test HWs on bayou
  - Linux
  - C language unless other specified

- Two options
  - Scp back and forth between your machine and bayou
  - Do all the development on bayou

Network Applications





Client-Server

Hybrid

Peer-to-peer

http://en.wikipedia.org/wiki/Peer-to-peer

Inter-Application Communication

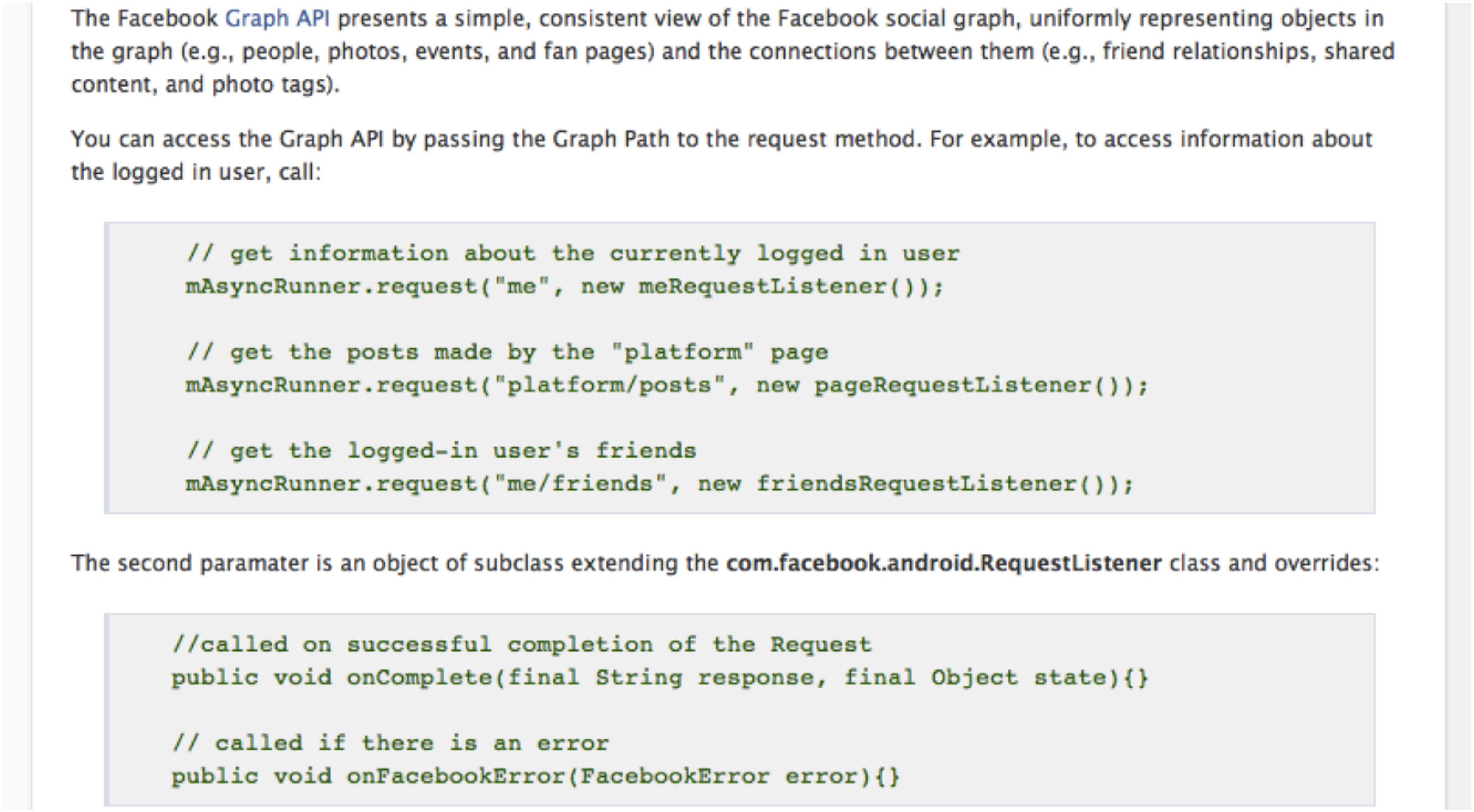
- Need a way to send and receive messages
- Inter-process communication
- Need naming, routing, transport
- Transport using TCP and UDP
  - On top of IP

Server
Client

Application Protocols

- Messages between processes, typically encapsulated within TCP or UDP

- Need agreement between
  – Sending process
  – Receiving process



The Facebook Graph API presents a simple, consistent view of the Facebook social graph, uniformly representing objects in the graph (e.g., people, photos, events, and fan pages) and the connections between them (e.g., friend relationships, shared content, and photo tags).

You can access the Graph API by passing the Graph Path to the request method. For example, to access information about the logged in user, call:

```
// get information about the currently logged in user
mAsyncRunner.request("me", new meRequestListener());

// get the posts made by the "platform" page
mAsyncRunner.request("platform/posts", new pageRequestListener());

// get the logged-in user's friends
mAsyncRunner.request("me/friends", new friendsRequestListener());
```

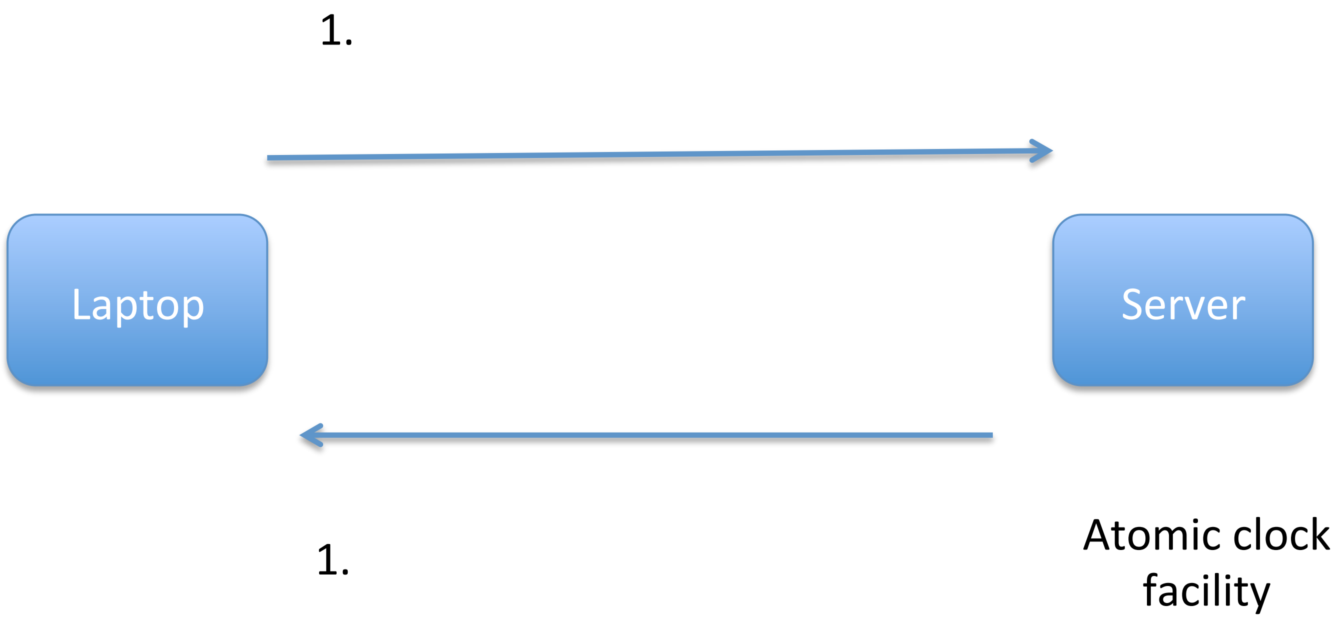
The second paramater is an object of subclass extending the **com.facebook.android.RequestListener** class and overrides:

```
//called on successful completion of the Request
public void onComplete(final String response, final Object state){}

// called if there is an error
public void onFacebookError(FacebookError error){}
```

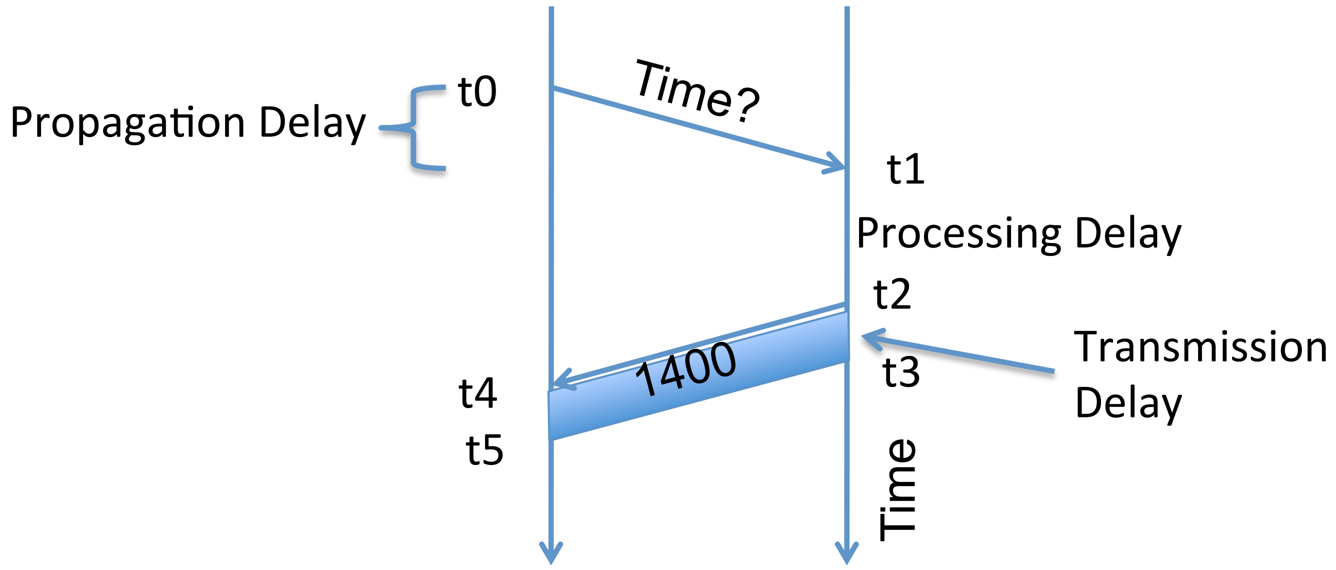
From: http://developers.facebook.com/docs/mobile/android/build/

Network Time Service

Client-server or peer-to-peer?



Protocol Timing Diagram



Cloud-based File Backup Application

- Client-server or peer-to-peer?
- Where do the applications run?
- Who/how to run these applications?
- What messages are exchanged?

1.

2.

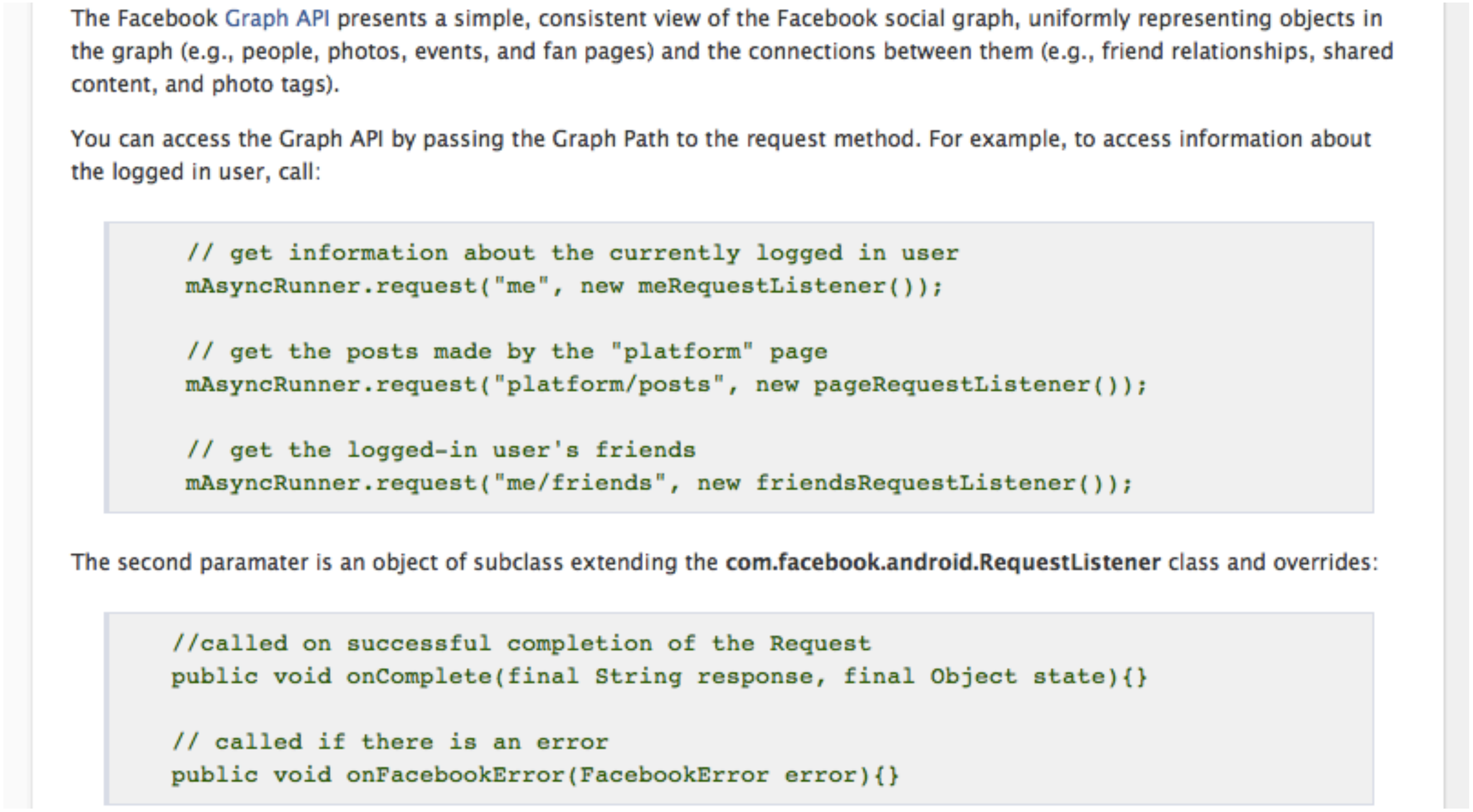
3.

Laptop
Server

1.

2.

3.


The Facebook Graph API presents a simple, consistent view of the Facebook social graph, uniformly representing objects in the graph (e.g., people, photos, events, and fan pages) and the connections between them (e.g., friend relationships, shared content, and photo tags).

You can access the Graph API by passing the Graph Path to the request method. For example, to access information about the logged in user, call:

```
// get information about the currently logged in user
mAsyncRunner.request("me", new meRequestListener());

// get the posts made by the "platform" page
mAsyncRunner.request("platform/posts", new pageRequestListener());

// get the logged-in user's friends
mAsyncRunner.request("me/friends", new friendsRequestListener());
```

The second paramater is an object of subclass extending the **com.facebook.android.RequestListener** class and overrides:

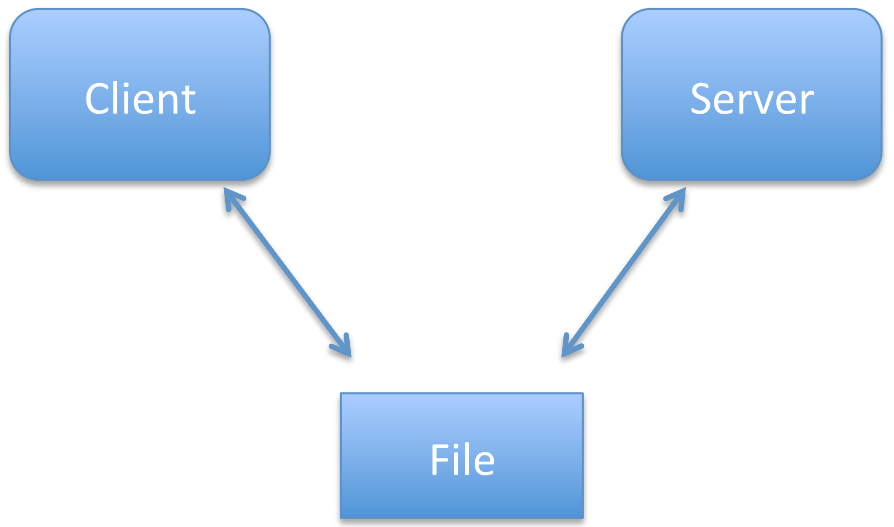
```
//called on successful completion of the Request
public void onComplete(final String response, final Object state){}

// called if there is an error
public void onFacebookError(FacebookError error){}
```

From: http://developers.facebook.com/docs/mobile/android/build/

Using TCP/IP

- How can applications use the network?
- *Sockets* API.
  - Originally from BSD, widely implemented (*BSD, Linux, Mac OS X, Windows, …)
  - Higher-level APIs build on them
- After basic setup, much like files

One could test network protocols with r/w on a file



System Calls

- Problem: how to access resources other then CPU
  - Disk, network, terminal, other processes
  - CPU prohibits instructions that would access devices
  - Only privileged OS kernel can access devices
- Kernel supplies well-defined system call interface
  - Applications request I/O operations through syscalls
  - Set up syscall arguments and trap to kernel
  - Kernel performs operation and returns results
- Higher-level functions built on syscall interface
  - `printf, scanf, gets,` all user-level code

File Descriptors

- Most I/O in Unix done through *file descriptors*
  - Integer *handles* to per-process table in kernel
- `int open(char *path, int flags, ...);`
- Returns file descriptor, used for all I/O to file

http://en.wikipedia.org/wiki/File_descriptor

Error Returns

- What if `open` fails? Returns -1 (invalid fd)
- Most system calls return -1 on failure
  - Specific type of error in global `int errno`
- `#include <sys/errno.h>` for possible values
  - 2 = ENOENT “No such file or directory”
  - 13 = EACCES “Permission denied”

Some operations on File Descriptors

- `ssize_t read (int fd, void *buf, int nbytes);`
  - Returns number of bytes read
  - Returns 0 bytes at end of file, or -1 on error
- `ssize_t write (int fd, void* buf, int nbytes);`
  - Returns number of bytes written, -1 on error
- `off_t lseek (int fd, off_t offset, int whence);`
  - whence: `SEEK_SET`, `SEEK_CUR`, `SEEK_END`
  - returns new offset, or -1 on error
- `int close (int fd);`

Sockets: Communication Between Machines

- Network sockets are file descriptors too
- Datagram sockets: unreliable message delivery
  - With IP, gives you UDP
  - Send atomic messages, which may be reordered or lost
  - Special system calls to read/write: `send`/`recv`
- Stream sockets: bi-directional pipes
  - With IP, gives you TCP
  - Bytes written on one end read on another
  - Reads may not return full amount requested, must re-read

System calls for using TCP

Client	Server
	`socket` – make socket
	`bind` – assign address, port
	`listen` – listen for clients
`socket` – make socket	
`bind*` – assign address	
`connect` – connect to listening socket	
	`accept` – accept connection

- This call to bind is optional, connect can choose address & port.

Socket Naming

- Naming of TCP & UDP communication endpoints
  - IP address specifies host (129.7.240.18)
  - 16-bit port number demultiplexes within host
  - Well-known services listen on standard ports (*e.g.* ssh – 22, http – 80, see /etc/services for list)
  - Clients connect from arbitrary ports to well known ports
- A connection is named by 5 components
  - Protocol, local IP, local port, remote IP, remote port
  - TCP requires connected sockets, but not UDP

Socket Address Structures

- Socket interface supports multiple network types
- Most calls take a generic `sockaddr`:

```
struct sockaddr {
  uint16_t sa_family;   /* address family */
  char     sa_data[14]; /* protocol-specific addr */
};
```

- *E.g.* `int connect(int s, struct sockaddr* srv,`
```
                             socklen_t addrlen);
```
- Cast `sockaddr *` from protocol-specific struct, *e.g.,*

```
struct sockaddr_in {
   short   sin_family;        /* = AF_INET */
   u_short sin_port;          /* = htons (PORT) */
   struct  in_addr sin_addr; /*32-bit IPv4 addr */
   chars    in_zero[8];
};
```

Dealing with Address Types

- All values in network byte order (Big Endian)
  - `htonl()`, `htons()`: host to network, 32 and 16 bits
  - `ntohl()`, `ntohs()`: network to host, 32 and 16 bits
  - Remember to always convert!
- All address types begin with family
  - `sa_family` in `sockaddr` tells you actual type
- Not all addresses are the same size
  - e.g., `struct sockaddr_in6` is typically 28 bytes, yet generic `struct sockaddr` is only 16 bytes
  - So most calls require passing around socket length
  - New `sockaddr_storage` is big enough

Client Skeleton (IPv4)

```
struct sockaddr_in {
        short   sin_family;  /* = AF_INET */
        u_short sin_port;    /* = htons (PORT) */
        struct  in_addr sin_addr;
        char    sin_zero[8];
} sin;

int s = socket (AF_INET, SOCK_STREAM, 0);
bzero (&sin, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_port = htons (13);  /* daytime port */
sin.sin_addr.s_addr = htonl (IP_ADDRESS);
connect (s, (sockaddr *) &sin, sizeof (sin));
while ((n = read (s, buf, sizeof (buf))) > 0)
  write (1, buf, n);
```

Server Skeleton (IPv4)

```
int s = socket (AF_INET, SOCK_STREAM, 0);
struct sockaddr_in sin;
bzero (&sin, sizeof (sin));
sin.sin_family = AF_INET;
sin.sin_port = htons (9999);
sin.sin_addr.s_addr = htonl (INADDR_ANY);
bind (s, (struct sockaddr *) &sin, sizeof (sin));
listen (s, 5);

for (;;) {
  socklen_t len = sizeof (sin);
  int cfd = accept (s, (struct sockaddr *) &sin, &len);
  /* cfd is new connection; you never read/write s */
  do_something_with (cfd);
  close (cfd);
}
```

Looking up a socket address with `getaddrinfo`

```
struct addrinfo hints, *ai;
int err;
memset (&hints, 0, sizeof (hints));
hints.ai_family = AF_UNSPEC;    /* or AF_INET or AF_INET6 */
hints.ai_socktype = SOCK_STREAM;/* or SOCK_DGRAM for UDP */

err = getaddrinfo ("www.brown.edu", "http", &hints, &ai);
if (err)
   fprintf (stderr, "%s\n", gia_strerror (err));
else {
   /* ai->ai_family = address type (AF_INET or AF_INET6) */
   /* ai->ai_addr = actual address cast to (sockaddr *) */
   /* ai->ai_addrlen = length of actual address */
   freeaddrinfo (ai); /* must free when done! */
}
```

getaddrinfo() [RFC3493]

- Protocol-independent node name to address translation
  - Can specify port as a service name or number
  - May return multiple addresses
  - You must free the structure with freeaddrinfo
- Other useful functions to know about
  - getnameinfo – Lookup hostname based on address
  - inet_ntop – Convert IPv4 or 6 address to printable
  - Inet_pton – Convert string to IPv4 or 6 address

EOF in more detail

- What happens at end of store?
  - Server receives EOF, renames file, responds OK
  - Client reads OK, *after* sending EOF: didn't close fd
- `int shutdown(int fd, int how);`
  - Shuts down a socket w/o closing file descriptor
  - how: 0 = read, 1 = write, 2 = both
  - Note: applies to *socket*, not descriptor, so copies of descriptor (through fork or dup affected)
  - Note 2: with TCP, can't detect if other side shuts for reading

Using UDP

- Call socket with SOCK_DGRAM, bind as before
- New calls for sending/receiving individual packets
  - `sendto(int s, const void *msg, int len, int flags, const struct sockaddr *to, socklen t tolen);`
  - `recvfrom(int s, void *buf, int len, int flags, struct sockaddr *from, socklen t *fromlen);`
  - Must send/get peer address with each packet
- Can use UDP in connected mode (Why?)
  - connect assigns remote address
  - `send`/`recv` syscalls, like `sendto`/`recvfrom` w/o last two arguments

Serving Multiple Clients

- A server may block when talking to a client
  - Read or write of a socket connected to a slow client can block
  - Server may be busy with CPU
  - Server might be blocked waiting for disk I/O
- Concurrency through multiple processes
  - Accept, fork, close in parent; child services request
- Advantages of one process per client
  - Don't block on slow clients
  - May use multiple cores
  - Can keep disk queues full for disk-heavy workloads

Threads

- One process per client has disadvantages:
  - High overhead – fork + exit ~100μsec
  - Hard to share state across clients
  - Maximum number of processes limited
- Can use threads for concurrency
  - Data races and deadlocks make programming tricky
  - Must allocate one stack per request
  - Many thread implementations block on some I/O or have heavy thread-switch overhead

Rough equivalents to `fork()`, `waitpid()`, `exit()`, `kill()`, plus locking primitives.

Non-blocking I/O

- fcntl sets O_NONBLOCK flag on descriptor

```
int n;
if ((n = fcntl(s, F_GETFL)) >= 0)
  fcntl(s, F_SETFL, n|O_NONBLOCK);
```

- Non-blocking semantics of system calls:
  - read immediately returns -1 with errno EAGAIN if no data
  - write may not write all data, or may return EAGAIN
  - connect may fail with EINPROGRESS (or may succeed, or may fail with a real error like ECONNREFUSED)
  - accept may fail with EAGAIN or EWOULDBLOCK if no connections present to be accepted

How do you know when to read/write?

```
struct timeval {
  long    tv_sec;         /* seconds */
  long    tv_usec;        /* and microseconds */
};

int select (int nfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, struct timeval *timeout);
FD_SET(fd, &fdset);
FD_CLR(fd, &fdset);
FD_ISSET(fd, &fdset);
FD_ZERO(&fdset);
```

- Entire program runs in an *event loop*

Event-driven servers

- Quite different from processes/threads
  - Race conditions, deadlocks rare
  - Often more efficient
- But…
  - Unusual programming model
  - Sometimes difficult to avoid blocking
  - Scaling to more CPUs is more complex