Optimized Networking and Data Handling in iOS 4

Jim Dovey
iTeam Lead Developer
Kobo Inc.
http://alanquatermain.net

Been writing software professionally for 11 years, almost every project has involved networking as a major component

In this session…

Today we'll look at how to make the most of iOS networking frameworks
Asynchronous APIs and techniques, limited resource usage, and do's & don'ts of Reachability APIs
Also data— handling large amounts of data from the network with few resources.
Asynchronously.

In this session…

- **Networking**
  - Asynchronicity
  - Resource usage
  - Network availability

Today we'll look at how to make the most of iOS networking frameworks
Asynchronous APIs and techniques, limited resource usage, and do's & don'ts of Reachability APIs
Also data— handling large amounts of data from the network with few resources.
Asynchronously.

In this session…

- **Networking**
  - Asynchronicity
  - Resource usage
  - Network availability
- **Data**
  - Problems with large data sets
  - Asynchronicity

Today we'll look at how to make the most of iOS networking frameworks
Asynchronous APIs and techniques, limited resource usage, and do's & don'ts of Reachability APIs
Also data— handling large amounts of data from the network with few resources.
Asynchronously.

In this session…

- **Networking**
  - Asynchronicity
  - Resource usage
  - Network availability
- **Data**
  - Problems with large data sets
  - Asynchronicity

Today we'll look at how to make the most of iOS networking frameworks
Asynchronous APIs and techniques, limited resource usage, and do's & don'ts of Reachability APIs
Also data— handling large amounts of data from the network with few resources.
Asynchronously.

Asynchronicity

The Most Important Word In The World

Why asynchronous?

There are a few reasons people come up with to explain why they don’t need to plan for async behaviour.

Why asynchronous?

- “I only use the network occasionally.”

There are a few reasons people come up with to explain why they don’t need to plan for async behaviour.

Why asynchronous?

- “I only use the network occasionally.”
  - ⌧ Occasional = Unexpected

There are a few reasons people come up with to explain why they don’t need to plan for async behaviour.

Why asynchronous?

- “I only use the network occasionally.”
  - ⌧ Occasional = Unexpected
- “I only send very small packets.”

There are a few reasons people come up with to explain why they don’t need to plan for async behaviour.

Why asynchronous?

- “I only use the network occasionally.”
  - ☒ Occasional = Unexpected
- “I only send very small packets.”
  - ☒ Latency doesn’t care about packet size

There are a few reasons people come up with to explain why they don’t need to plan for async behaviour.

Why asynchronous?

- “I only use the network occasionally.”
  - ☒ Occasional = Unexpected
- “I only send very small packets.”
  - ☒ Latency doesn’t care about packet size
- “I only send, I don’t wait for a reply.”

There are a few reasons people come up with to explain why they don’t need to plan for async behaviour.

Why asynchronous?

- “I only use the network occasionally.”
  - ☒ Occasional = Unexpected
- “I only send very small packets.”
  - ☒ Latency doesn’t care about packet size
- “I only send, I don’t wait for a reply.”
  - ☒ You might not, but TCP will

There are a few reasons people come up with to explain why they don’t need to plan for async behaviour.

Why asynchronous?

- “I only use the network occasionally.”
  - ☒ Occasional = Unexpected
- “I only send very small packets.”
  - ☒ Latency doesn’t care about packet size
- “I only send, I don’t wait for a reply.”
  - ☒ You might not, but TCP will
- “It’s always prompted by the user— they expect to wait.”

There are a few reasons people come up with to explain why they don’t need to plan for async behaviour.

Why asynchronous?

- “I only use the network occasionally.”
  - ☒ Occasional = Unexpected
- “I only send very small packets.”
  - ☒ Latency doesn’t care about packet size
- “I only send, I don’t wait for a reply.”
  - ☒ You might not, but TCP will
- “It’s always prompted by the user— they expect to wait.”
  - ☒ What if they change their mind?

There are a few reasons people come up with to explain why they don’t need to plan for async behaviour.

Synchronous + Main Thread = Death

NEVER DO THIS

Watchdog vs. the Net

Operation	Default Timeout
Watchdog	20 seconds

Watchdog will kill your app long before any of your standard networking tasks times out

Watchdog vs. the Net

Operation	Default Timeout
Watchdog	20 seconds
DNS Lookup	30 seconds

Watchdog will kill your app long before any of your standard networking tasks times out

Watchdog vs. the Net

Operation	Default Timeout
Watchdog	20 seconds
DNS Lookup	30 seconds
TCP Connection	75 seconds

Watchdog will kill your app long before any of your standard networking tasks times out

Watchdog vs. the Net

Operation	Default Timeout
Watchdog	20 seconds
DNS Lookup	30 seconds
TCP Connection	75 seconds
NSURLConnection	60 seconds

Watchdog will kill your app long before any of your standard networking tasks times out

Why asynchronous?

Why asynchronous?

- Responsiveness
  - User Interface can update (e.g. to show progress)
  - Other app jobs aren't held up

Why asynchronous?

- Responsiveness
  - User Interface can update (e.g. to show progress)
  - Other app jobs aren't held up
- Cancellation
  - App quits, user cancels, or an error occurs

Why asynchronous?

- Responsiveness
  - User Interface can update (e.g. to show progress)
  - Other app jobs aren't held up
- Cancellation
  - App quits, user cancels, or an error occurs
- Prioritization
  - Not all messages are of equal importance

```
- (void) doNetworkStuff {
  [self fetchHeader];
  [self inspectHeader];
  [self fetchAttrs];
  [self inspectAttrs];
  [self fetchData];
  [self storeData];
}
```

```
- (void) start {
  [self requestHeader];
}
- (void) gotHeader {
  [self inspectHeader];
  [self requestAttrs];
}

- (void) gotAttrs {
  [self inspectAttrs];
  [self requestData];
}

- (void) gotData {
  [self storeData];
}
```

Grabbing file data from a server: basic info (header), attributes (ACLs, resource data), content

```
- (void) doNetworkStuff {
  [self fetchHeader];
  [self inspectHeader];
  [self fetchAttrs];
  [self inspectAttrs];
  [self fetchData];
  [self storeData];
}
```

```
- (void) start {
  [self requestHeader];
}
- (void) gotHeader {
  [self inspectHeader];
  [self requestAttrs];
}

- (void) gotAttrs {
  [self inspectAttrs];
  [self requestData];
}

- (void) gotData {
  [self storeData];
}
```

Split it up. Send a request, let the system tell you when the response is ready to be handled

Not just ‘network in the background’

- Connecting
- Retrieving data
- Parsing data
- Display updates

Lots of different things involved around the use of networked data

New Thread + Synchronous Calls

- Pros
  - Easily re-use existing synchronous code
  - Fast and simple to implement
- Cons
  - Cancellation is difficult
  - Many connections means many threads
  - Not the best use of processing time per thread

Wrapping sync code in new threads
Threads take up precious resources, are not cheap enough to spawn & throw away over and over

```
void MySyncNetworkCall {
  int s = socket(...);
  if (!connect(s, ...)) {
    ...
  }
  close(s);
}
```

```
void MySyncNetworkCall {
  int s = socket(...);
  if (!connect(s, ...)) {
    ...
  }
  close(s);
}
```

```
- (void) networkCall {
  [self
performSelectorInBackground:
@selector(runNetworkCall)];
}
```

```
void MySyncNetworkCall {
  int s = socket(...);
  if (!connect(s, ...)) {
    ...
  }
  close(s);
}
```

```
- (void) networkCall {
  [self
performSelectorInBackground:
@selector(runNetworkCall)];
}
```

```
- (void) runNetworkCall {
  NSAutoreleasePool * pool =
[[NSAutoreleasePool alloc]
init];

  MySyncNetworkCall();

  [pool drain];
}
```

This approach necessitates the creation of a thread-main function of some kind

```
void MySyncNetworkCall {
  int s = socket(...);
  if (!connect(s, ...)) {
    ...
  }
  close(s);
}
```

```
- (void) networkCall {
  dispatch_async(q, ^{
    MySyncNetworkCall();
  }
}
```

This approach does not— but more of that later

NSURLConnection and NSRunLoop

- Default API model is asynchronous using run loops
- Uses delegation to decouple networking details from program logic
- Delegation API allows for many things:
  - Authentication
  - Redirection
- Common network protocols completely abstracted (HTTP, FTP, and TLS/SSL)
- One-click support for HTTP pipelining

Asynchronous by default, encapsulates the major protocols
HTTP pipelining

NSURLConnection

```
- (void) startRequestForURL: (NSURL *) url {
  NSURLRequest *r = [NSURLRequest requestWithURL: url];
  myConn = [[NSURLConnection connectionWithRequest: r
                                          delegate: self] retain];
}
```

NSURL is used to designate the remote resource

NSURLConnection

```
- (void) startRequestForURL: (NSURL *) url {
  NSURLRequest *r = [NSURLRequest requestWithURL: url];
  myConn = [[NSURLConnection connectionWithRequest: r
                                          delegate: self] retain];
}
```

Create a request— can be as simple as this, or optionally more involved for particular protocols i.e. HTTP

NSURLConnection

```
- (void) startRequestForURL: (NSURL *) url {
  NSURLRequest *r = [NSURLRequest requestWithURL: url];
  myConn = [[NSURLConnection connectionWithRequest: r
                                          delegate: self] retain];
}
```

Create the URL connection, which starts the data transfer going

NSURLConnection

```
- (void) startRequestForURL: (NSURL *) url {
  NSURLRequest *r = [NSURLRequest requestWithURL: url];
  myConn = [[NSURLConnection connectionWithRequest: r
                                          delegate: self] retain];
}
```

Keep the connection around, or it will close when it's autoreleased or garbage-collected

NSURLConnection

```
- (void) connection:(NSURLConnection *)c
     didReceiveData:(NSData *)data

- (void) connectionDidFinishLoading:(NSURLConnection*)c

- (void) connection:(NSURLConnection*)c
    didFailWithError:(NSError *)error
```

The most-used delegate methods

NSURLConnection

```
- (void) connection:(NSURLConnection *)c
     didReceiveData:(NSData *)data {
  [myData appendData: data];
}
```

Can accumulate data in memory, but better to send it to a file— don't know in advance how much data you'll get

NSURLConnection

```
- (void) connection:(NSURLConnection *)c
     didReceiveData:(NSData *)data {
  [myData appendData: data];
}

- (void) connection:(NSURLConnection *)c
     didReceiveData:(NSData *)data {
  [myFileHandle writeData: data];
}
```

Can accumulate data in memory, but better to send it to a file— don't know in advance how much data you'll get

NSURLConnection

```
- (void) connectionDidFinishLoading:(NSURLConnection*)c
{
   [myFileHandle synchronizeFile];
  NSData *d = [NSData dataWithContentsOfMappedFile:
path];
   // do something with data...
}
```

Get data when done by using file mapping to let the memory subsystem page bits in and out to save space when it gets tight

NSURLConnection

```
- (void) connection:(NSURLConnection*)c
   didFailWithError:(NSError *)error {
  [myDelegate reportError: error];
  // connection will receive no more data
}
```

Report the error to user, logs, other app components, etc.

NSURLConnection

```
- (NSURLRequest*)connection:
            willSendRequest:redirectionResponse:
- (BOOL)connectionShouldUseCredentialStorage:
- (void)connection:didReceiveAuthenticationChallenge:
- (void)connection:didCancelAuthenticationChallenge:
- (void)connection:didReceiveResponse:
- (void)connection:didSendBodyData:totalBytesWritten:
        totalBytesExpectedToWrite:
- (NSCachedURLResponse*)connection:willCacheResponse:
```

Other things you can do— observe/change/deny redirection
Authentication details
Monitor body data being sent and whether responses are cached (can modify caches)

CFNetwork

- Lower level than NSURLConnection
- Finely-grained network operations
- Greater control over certain protocols

CFNetwork lets you work around more edge cases than NSURLConnection
Streams are the BOMB

CFNetwork

- Lower level than NSURLConnection
- Finely-grained network operations
- Greater control over certain protocols
- Stream input/output APIs

CFNetwork lets you work around more edge cases than NSURLConnection
Streams are the BOMB

CFNetwork HTTP

```
- (NSInputStream *) startRequestForURL: (NSURL *) url {
  CFHTTPMessageRef msg = CFHTTPMessageCreateRequest(
      NULL, CFSTR("GET"), (CFURLRef)[url absoluteURL],
      kCFHTTPVersion1_1);
  CFHTTPMessageSetHeaderFieldValue(msg, header, value);
  NSInputStream * s =
      (NSInputStream*)CFReadStreamCreateForHTTPRequest(msg);
  return ( [s autorelease] );
}
```

Create a message ref — analogous to NSURLRequest

CFNetwork HTTP

```
- (NSInputStream *) startRequestForURL: (NSURL *) url {
  CFHTTPMessageRef msg = CFHTTPMessageCreateRequest(
      NULL, CFSTR("GET"), (CFURLRef)[url absoluteURL],
      kCFHTTPVersion1_1);
  CFHTTPMessageSetHeaderFieldValue(msg, header, value);
  NSInputStream * s =
      (NSInputStream*)CFReadStreamCreateForHTTPRequest(msg);
  return ( [s autorelease] );
}
```

Always provide an absolute URL, it doesn't handle relative URLs with a base URL well

CFNetwork HTTP

```
- (NSInputStream *) startRequestForURL: (NSURL *) url {
  CFHTTPMessageRef msg = CFHTTPMessageCreateRequest(
      NULL, CFSTR("GET"), (CFURLRef)[url absoluteURL],
      kCFHTTPVersion1_1);
  CFHTTPMessageSetHeaderFieldValue(msg, header, value);
  NSInputStream * s =
      (NSInputStream*)CFReadStreamCreateForHTTPRequest(msg);
  return ( [s autorelease] );
}
```

Can set HTTP header field values just like NSURLRequest

CFNetwork HTTP

```
- (NSInputStream *) startRequestForURL: (NSURL *) url {
  CFHTTPMessageRef msg = CFHTTPMessageCreateRequest(
      NULL, CFSTR("GET"), (CFURLRef)[url absoluteURL],
      kCFHTTPVersion1_1);
  CFHTTPMessageSetHeaderFieldValue(msg, header, value);
  NSInputStream * s =
      (NSInputStream*)CFReadStreamCreateForHTTPRequest(msg);
  return ( [s autorelease] );
}
```

Get a stream from it— but remember to autorelease, since it comes from a CF Create API

CFNetwork FTP

```
CFReadStreamRef reader = NULL;
CFURLRef url = /* ftp://ftp.cdrom.com/public/os/linux/ */;
reader = CFReadStreamCreateWithFTPURL(NULL, url);

CFReadStreamOpen( reader );
CFReadStreamRead( ... );
... // receive all the data
```

FTP gives very simple access to FTP servers

CFNetwork FTP

```
CFReadStreamRef reader = NULL;
CFURLRef url = /* ftp://ftp.cdrom.com/public/os/linux/ */;
reader = CFReadStreamCreateWithFTPURL(NULL, url);

CFReadStreamOpen( reader );
CFReadStreamRead( ... );
... // receive all the data

CFDictionaryRef dirListing = NULL;
CFFTPCreateParsedResourceListing(NULL, dataBuf, dataLen,
    &dirListing);
```

Including parsing out the contents of directory listings

CFNetwork FTP

```
CFDictionaryRef dirListing = NULL;
CFFTPCreateParsedResourceListing(NULL, dataBuf, dataLen,
    &dirListing);

kCFFTPResourceMode
            ...Name
            ...Owner
            ...Group
            ...Link
            ...Size
            ...Type
            ...ModDate
```

Parsed dir listings let you inspect resource attributes via the parsed dictionary

CFNetwork Sockets

```
CFReadStreamRef input = NULL;
CFWriteStreamRef output = NULL;

CFStreamCreatePairWithSocketToCFHost(NULL, aCFHostRef, aPort,
    &input, &output);

CFStreamCreatePairWithSocketToNetService(NULL, aServiceRef,
    &input, &output);
```

Can be used to create streams from socket connection information
Including NetServices (Bonjour)

NSNetwork Sockets

```
NSInputStream * input = nil;
NSOutputStream * output = nil;

[NSStream getStreamsToHost: anNSHost port: aPort
                inputStream: &input outputStream: &output];

[anNSNetService getInputStream: &input
                  outputStream: &output];
```

…but NSStream/NSNetService provide analogous and simpler ways of doing the same.

CFNetwork Sockets

```
kCFStreamPropertySSLSettings
- kCFStreamPropertySSLPeerTrust
- kCFStreamSSLValidatesCertificateChain
- kCFStreamSSLLevel
- kCFStreamSSLPeerName
- kCFStreamSSLAllowsExpiredCertificates
- kCFStreamSSLAllowsExpiredRoots

kCFStreamPropertySOCKSProxyHost
kCFStreamPropertySOCKSProxyPort
kCFStreamPropertySOCKSVersion
kCFStreamPropertySOCKSUser
kCFStreamPropertySOCKSPassword
```

CF versions support much more fine-grained properties
e.g. SSL peer name— virtual host whose server certificate doesn't match its name
Can set the expected certificate name so it won't fail to validate like NSURLConnection would

Using Streams

```
- (void) startStreaming {
    [myStream open];
}

- (void) stopStreaming {
    [myStream close];
}
```

Streams are simple to use— open/close them

Using Streams

```
- (void) stream:(NSStream*)stream
    handleEvent:(NSStreamEvent)event {
  switch ( event ) {
    case NSStreamEventOpenCompleted:
      ...
    case NSStreamEventHasBytesAvailable:
      ...
    case NSStreamEventHasSpaceAvailable:
      ...
    case NSStreamEventErrorOccurred:
      ...
    case NSStreamEventEndEncountered:
      ...
  }
}
```

Single delegation callback with five events
HasBytes/HasSpace are only for Input/Output streams respectively

Using Streams

```
switch ( event ) {
   case NSStreamEventOpenCompleted:
      [self postConnectedNotification];
      break;
   ...
}
```

When it opens you can assume that data or error will happen, so can notify other components or the user

Using Streams

```
switch ( event ) {
  case NSStreamEventHasBytesAvailable:
    uint8_t buf[1024];
    int len = [stream read: buf maxLength: 1024];
    if ( len > 0 )
      ...
    break;
  ...
}
```

When bytes are available, read using a simple API. Read as much or as little as you want— API returns the amount actually read.

Using Streams

```
switch ( event ) {
  case NSStreamEventHasSpaceAvailable:
    uint8_t *buf = [myData bytes] + numSent;
    int numLeft = [myData length] - numSent;
    int len = [stream write:buf length:numLeft];
    if ( len > 0 )
      numSent += len;
    break;
  ...
}
```

When an output stream has space to send, you can write as much as you like
It will tell you how much it was able to send

Using Streams

```
switch ( event ) {
  case NSStreamEventErrorOccurred:
    [self postErrorNotification:[stream streamError]
                      forStream:stream];
    self.complete = YES;
    break;
  ...
}
```

If an error occurs, you can get the NSError from the stream
Stream operations are now complete— an error is one which prevents further use of the stream

Using Streams

```
switch ( event ) {
   case NSStreamEventEndEncountered:
      self.complete = YES;
      [self handleDownloadedData];
      break;
   ...
}
```

When you reach the end of the stream, you can’t use it any more (no seeking on standard streams)

Case Study: Outpost

NSURLConnection

Started out using NSURLConnection. Found ridiculously large files from server.
NSURLConnection's internals were creating some buffers
Storing downloaded data in memory before parsing it = bad

NSURLConnection

- Large XML files. Very large XML files

Started out using NSURLConnection. Found ridiculously large files from server.
NSURLConnection's internals were creating some buffers
Storing downloaded data in memory before parsing it = bad

NSURLConnection

- Large XML files. Very large XML files
- High memory consumption from NSURL protocol buffers

Started out using NSURLConnection. Found ridiculously large files from server.
NSURLConnection's internals were creating some buffers
Storing downloaded data in memory before parsing it = bad

NSURLConnection

- Large XML files. Very large XML files
- High memory consumption from NSURL protocol buffers
- Download **then** parse

Started out using NSURLConnection. Found ridiculously large files from server.
NSURLConnection's internals were creating some buffers
Storing downloaded data in memory before parsing it = bad

NSURLConnection + memory-mapped files

Write out data to disk, then map into memory
Only pages out when memory gets full— so memory still fills up
Many concurrent similar tasks can prevent this working, causing Low Memory crashes still

NSURLConnection + memory-mapped files

- `+[NSData dataWithContentsOfMappedFile:]`

Write out data to disk, then map into memory
Only pages out when memory gets full— so memory still fills up
Many concurrent similar tasks can prevent this working, causing Low Memory crashes still

NSURLConnection + memory-mapped files

- `+[NSData dataWithContentsOfMappedFile:]`
- Reduces in-memory footprint, but only until memory limit is reached

Write out data to disk, then map into memory
Only pages out when memory gets full— so memory still fills up
Many concurrent similar tasks can prevent this working, causing Low Memory crashes still

NSURLConnection + memory-mapped files

- `+[NSData dataWithContentsOfMappedFile:]`
- Reduces in-memory footprint, but only until memory limit is reached
- Data still takes some time to download

Write out data to disk, then map into memory
Only pages out when memory gets full— so memory still fills up
Many concurrent similar tasks can prevent this working, causing Low Memory crashes still

NSURLConnection + memory-mapped files

- `+[NSData dataWithContentsOfMappedFile:]`
- Reduces in-memory footprint, but only until memory limit is reached
- Data still takes some time to download
- Still downloading **then** parsing

Write out data to disk, then map into memory
Only pages out when memory gets full— so memory still fills up
Many concurrent similar tasks can prevent this working, causing Low Memory crashes still

NSURLConnection + Streaming XML Parser

Download slowly to disk, but then *stream* from disk into a new XML parser
Low in-memory footprint, only loads 1K at a time into RAM to parse
XML parser cleans its working heap more often (every 1K opp. to just once at the end)

NSURLConnection + Streaming XML Parser

- Large files will *always* take a long time to download

Download slowly to disk, but then *stream* from disk into a new XML parser
Low in-memory footprint, only loads 1K at a time into RAM to parse
XML parser cleans its working heap more often (every 1K opp. to just once at the end)

NSURLConnection + Streaming XML Parser

- Large files will *always* take a long time to download
- Very low in-memory footprint

Download slowly to disk, but then *stream* from disk into a new XML parser
Low in-memory footprint, only loads 1K at a time into RAM to parse
XML parser cleans its working heap more often (every 1K opp. to just once at the end)

NSURLConnection + Streaming XML Parser

- Large files will *always* take a long time to download
- Very low in-memory footprint
- Stream from file to XML parser

Download slowly to disk, but then *stream* from disk into a new XML parser
Low in-memory footprint, only loads 1K at a time into RAM to parse
XML parser cleans its working heap more often (every 1K opp. to just once at the end)

NSURLConnection + Streaming XML Parser

- Large files will *always* take a long time to download
- Very low in-memory footprint
- Stream from file to XML parser
- *Still* downloading then parsing

Download slowly to disk, but then *stream* from disk into a new XML parser
Low in-memory footprint, only loads 1K at a time into RAM to parse
XML parser cleans its working heap more often (every 1K opp. to just once at the end)

CFHTTPStream + Streaming XML Parser

Using CFNetwork we get a stream from the server
Feeds into the XML parser as the data arrives from the server
When all data is received, 95% or more of parsing is already done
Asynchronously handling data and network = WIN

CFHTTPStream + Streaming XML Parser

- Download still takes a long time **BUT:**

Using CFNetwork we get a stream from the server
Feeds into the XML parser as the data arrives from the server
When all data is received, 95% or more of parsing is already done
Asynchronously handling data and network = WIN

CFHTTPStream + Streaming XML Parser

- Download still takes a long time **BUT:**
- Memory usage is minimal

Using CFNetwork we get a stream from the server
Feeds into the XML parser as the data arrives from the server
When all data is received, 95% or more of parsing is already done
Asynchronously handling data and network = WIN

CFHTTPStream + Streaming XML Parser

- Download still takes a long time **BUT:**
- Memory usage is minimal
- Parses and collates data *while downloading*

Using CFNetwork we get a stream from the server
Feeds into the XML parser as the data arrives from the server
When all data is received, 95% or more of parsing is already done
Asynchronously handling data and network = WIN

CFHTTPStream + Streaming XML Parser

- Download still takes a long time **BUT:**
- Memory usage is minimal
- Parses and collates data *while downloading*
- Maybe an extra 1%-5% of total download time before all tasks are complete

Using CFNetwork we get a stream from the server
Feeds into the XML parser as the data arrives from the server
When all data is received, 95% or more of parsing is already done
Asynchronously handling data and network = WIN

Stepping back…

Now let’s look back at something we saw earlier

```
void MySyncNetworkCall {
  int s = socket(...);
  if (!connect(s, ...)) {
    ...
  }
  close(s);
}
```

```
- (void) networkCall {
  dispatch_async(q, ^{
    MySyncNetworkCall();
  }
}
```

We saw this…

```
void MySyncNetworkCall {
  int s = socket(...);
  if (!connect(s, ...)) {
    ...
  }
  close(s);
}
```

```
- (void) networkCall {
  dispatch_async(q, ^{
    MySyncNetworkCall();
  }
}
```

...and this is the interesting bit

Caret character is where the magic lies
Caret = blocks

Blocks

Caret character is where the magic lies
Caret = blocks

Do block *n* times

```
typedef void (^work_t)(void);

void
repeat(int n, work_t work) {
    for (int i = 0; i < n; i++)
        work();
}
```

Provides a way to encapsulate work without needing to define functions and pass/retain variables used by that work

Why Blocks?

- Smaller, easier, more precise code
- Encapsulates variables from enclosing scope (closures)
- More than 100 uses of blocks in Snow Leopard and iOS4 APIs
  - Callbacks
  - Concurrency
  - Traditional ‘collection’ uses: iterate, map, reduce, wrap

Blocks let you skip the setup associated with most work-queue implementations (variables, context pointers) by copying enclosing scope
Used everywhere in OS X 10.6 and and iOS4
Has many different use-cases

Grand Central Dispatch

Blocks were created so that GCD could happen

Asynchronous Blocks

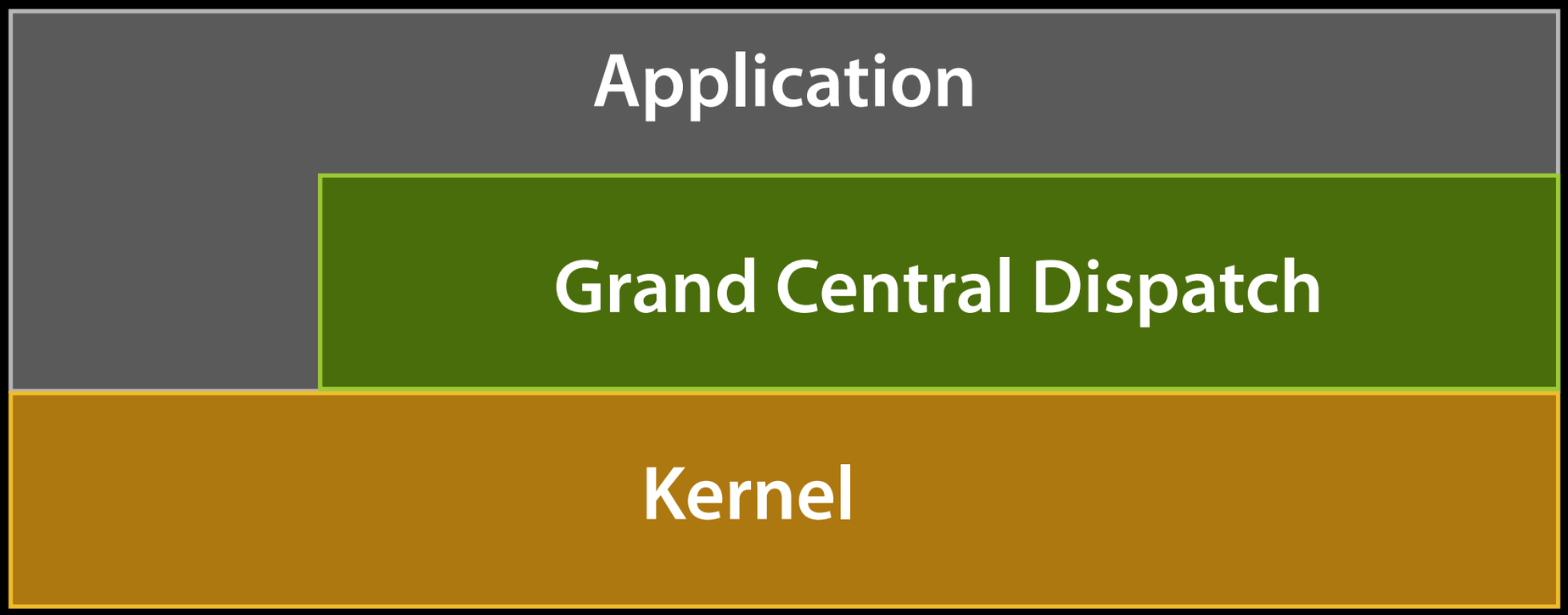
GCD lets you run blocks of work asynchronously

Asynchronous Blocks

- Provides serial and concurrent queues, semaphores, timers, event sources and more
- Utilizes highly efficient user-space atomic locking mechanisms
  - Avoids kernel traps in the case where resources are not in contention from multiple threads
  - 95+% of the time

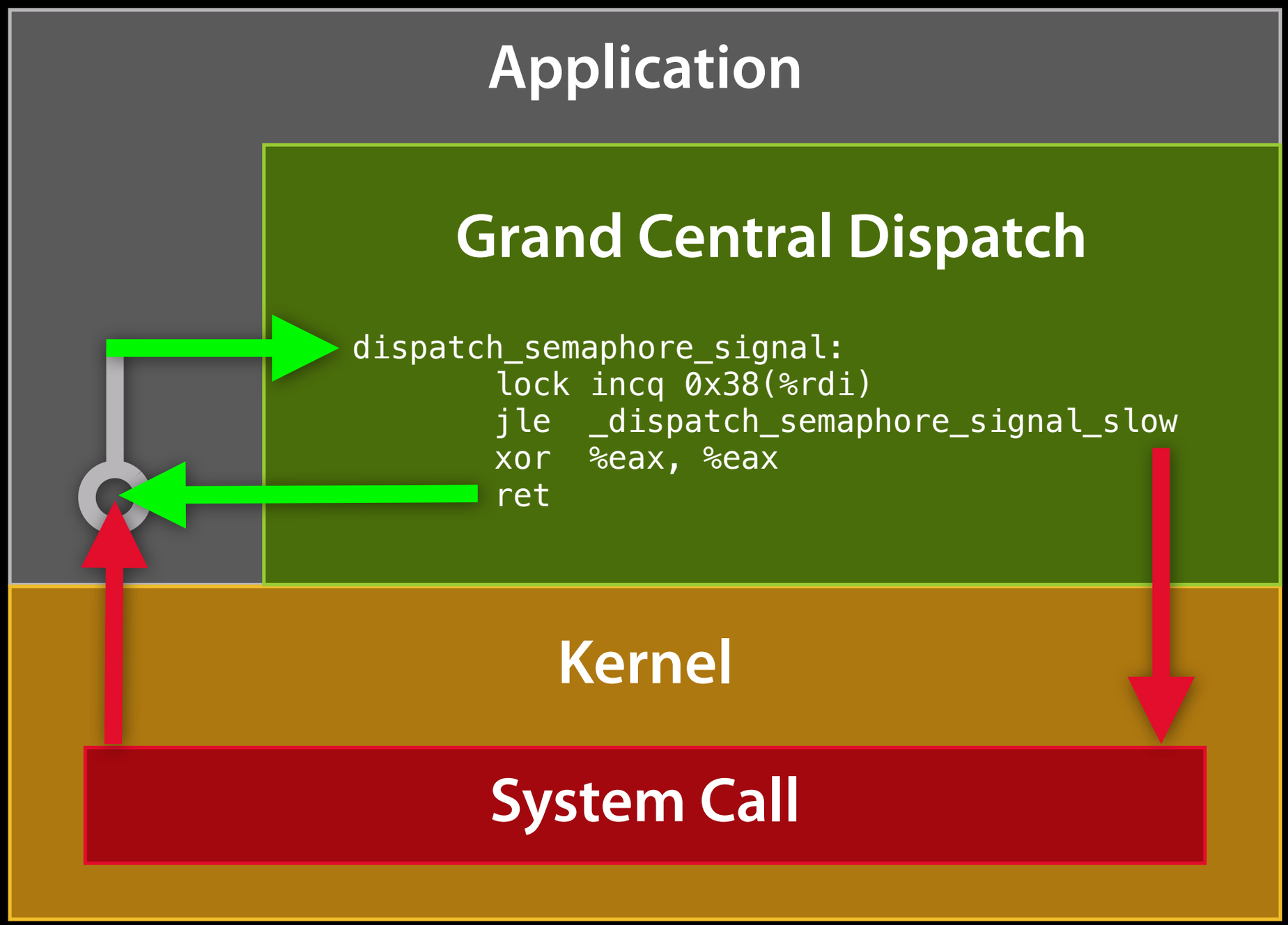
GCD provides serial and concurrent queues to which you can dispatch discrete work blocks
Highly efficient user-space locking mechanism avoids trapping to the kernel unless resource is actually contended, which is actually fairly rare

Stacking up



GCD is in user-space, but acts as an optional layer between the app and the kernel

Fast local dispatch



GCD semaphores try to assume that resources are not contended
When they aren't contended, the routine can perform a very fast atomic operation and return
When they are contended, it will trap to the kernel — but ONLY then

200x

Gives quite a speed-up

```
dispatch_async(queue, ^{
    printf(“Hello\n”);
});
```

It’s as simple as this

Overview

- GCD is part of libSystem
  - Available to all applications
  - `#include <dispatch/dispatch.h>`
- C, but uses object-oriented approach
- Uses reference-counted memory management
- Most APIs use a fast local path and make system calls only when necessary

Automatically included in all apps via libSystem
Pseudo-object system with polymorphism
Reference-counted— doesn't use garbage collection yet
APIs use fast local paths and trap to kernel only if necessary

Component APIs

- Objects
  - Similar to CF's polymorphic API
- Queues
  - Pass blocks or functions to run on a queue
- Semaphores
  - Fast, lightweight counted semaphores
- Groups
  - Collect blocks/functions, for notification
- Sources
  - Monitor files, ports, signals using blocks

Polymorphic object API implements queues, semaphores, groups (of blocks) to monitor async or sync, and event sources for file descriptors, signals, basic integer operations, and more

Queues

Queues first

Queues

- Provides a low-level yet almost API-free means of creating and dispatching concurrent tasks
- Very fast, using local (user-space) locking for enqueue and dequeue operations
- Can be used to synchronize access to any resource or method
- Very easy to run any snippet of code either in ‘the background’ or on the main thread

Very simple API. A queue is also very small, so you can create lots and lots of them
Can be used for concurrency or resource access synchronization

From Sync to Async

```
- (void) myFunction {
  [self openDocument];
  [self processDocument];
  [self displayDocument];
}
```

Making a sync function async with dispatch queues

From Sync to Async

```
- (void) myFunction {
  [self openDocument];
  dispatch_async(dispatch_get_global_queue(0, 0), ^{
    [self processDocument];
    [self displayDocument];
  });
}
```

First grab the long-running bit and run in the background on a global (concurrent) queue

From Sync to Async

```
- (void) myFunction {
  [self openDocument];
  dispatch_async(dispatch_get_global_queue(0, 0), ^{
    [self processDocument];
    dispatch_async(dispatch_get_main_queue(), ^{
      [self displayDocument];
    });
  });
}
```

That work block, when done, submits the final step to the main-thread's serial queue
Very simple to run code on the main thread with GCD

Lock-free Synchronization

```
- (void) addInterest:(float)rate {
  float newAmount = _amount * rate;
  _amount = base;
}
```

Resource access synchronization
Thread could be preempted between those two lines
Secondary thread runs this code
First thread overwrites second thread’s result = lost calculation

Lock-free Synchronization

```
- (void) addInterest:(float)rate {
    dispatch_async(_myQueue, ^{
        float newAmount = _amount * rate;
        _amount = base;
    });
}
```

By dispatching to your own queue (all of your own queues are serial in nature) you can ensure that the two events happen in order and complete before any other thread's version
Even under high contention, one is guaranteed to beat the other into the queue, guaranteeing synchronized access

Lock-free Synchronization

```
- (void) addInterest:(float)rate {
    dispatch_sync(_myQueue, ^{
        float newAmount = _amount * rate;
        _amount = base;
    });
}
```

Can also run synchronously if the API contract requires that the function not return until the work is done

Be careful— will deadlock if current code is already running on _myQueue

Lock-free Initialization

```
+ (id) sharedObject {
  if (__sharedInstance == nil) {
    __sharedInstance = [[self alloc] init];
  }
  return (__sharedInstance);
}
```

Initializing singletons— this might get preempted and create two instances, losing reference to first

Lock-free Initialization

```
+ (id) sharedObject {
    static dispatch_once_t onceControl = 0;
    dispatch_once(&onceControl, ^{
        __sharedInstance = [[self alloc] init];
    });
    return (__sharedInstance);
}
```

Use dispatch_once to run something only once per *process*
Atomic locking ensures this code will definitely only run once. Avoids CPU speculative look-ahead and instruction re-ordering.

Event Sources

Event sources— great for networking

GCD Event Sources

- Your new best friend — `dispatch_source_t`
- Implements a cleaner way of handling asynchronous (and ordered) events on network streams, amongst other things
- Based on the `kevent()` API, but hides that complexity from you

Use it. Use it a lot.
Asynchronous event handling, optionally in an ordered fashion depending on the source's target queue
Hides implementation details from you

For the network

```
- (void) setupSockets {
    int s = socket(...);
    cfSocket = CFSocketCreateWithNative(s,...);
    CFSocketEnableCallbacks(cfSocket, ...);
    CFSocketConnectToAddress(cfSocket, ...);
}
void MyCFSocketCallback(...) {
    switch (type) {
        ...
    }
}
```

Could create sockets and connect like this
Needs a separate callback function, context, switch on event types, etc.

For the network

```
- (void) setupSockets {
    int s = socket(...);
    rSrc = dispatch_source_create(
            DISPATCH_SOURCE_TYPE_READ, s, 0);
    dispatch_source_set_event_handler(rSrc, ^{
        uint8_t buf[1024];
        int len = read(s, buf, 1024);
        if (len > 0)
            ...
    });
}
```

This way you can create a source for readability, one for writability, and have events for each Set and event handler with this API, handing in a work block

For the network

```
- (void) setupSockets {
    int s = socket(...);
    rSrc = dispatch_source_create(
            DISPATCH_SOURCE_TYPE_READ, s, 0);
    dispatch_source_set_event_handler(rSrc, ^{
        uint8_t buf[1024];
        int len = read(s, buf, 1024);
        if (len > 0)
            ...
    });
}
```

Code to handle each event is in-line with the rest of network setup code, which makes maintenance easier

For UI updates

- The ‘data add’ source is perfect for progress updates
- Provides a simple way to atomically accumulate a numeric value
- Anything can update the source, and the events are run on a queue designated by the source’s creator

Updating UI is easy too — for instance showing progress of a download/upload or other determinate time-consuming operation

For UI updates

```
- (void) setupProgressWithMax:(float)max {
  progSrc = dispatch_source_create(
        DISPATCH_SOURCE_TYPE_DATA_ADD, 0, 0);
  dispatch_set_target_queue(progSrc,
        dispatch_get_main_queue());
  dispatch_source_set_event_handler(progSrc, ^{
    int d = dispatch_source_get_data(progSrc);
    _progressView.progress = (float)d / max;
  });
}
```

Create a source of type ‘data add’
Accumulates an integer value based on values handed in from any other thread/queue/code

For UI updates

```
- (void) setupProgressWithMax:(float)max {
  progSrc = dispatch_source_create(
        DISPATCH_SOURCE_TYPE_DATA_ADD, 0, 0);
  dispatch_set_target_queue(progSrc,
        dispatch_get_main_queue());
  dispatch_source_set_event_handler(progSrc, ^{
    int d = dispatch_source_get_data(progSrc);
    _progressView.progress = (float)d / max;
  });
}
```

Set its target queue— for UI, can have it run on the main thread
Don’t need to be on the main thread to set this up at all

For UI updates

```
- (void) setupProgressWithMax:(float)max {
  progSrc = dispatch_source_create(
        DISPATCH_SOURCE_TYPE_DATA_ADD, 0, 0);
  dispatch_set_target_queue(progSrc,
        dispatch_get_main_queue());
  dispatch_source_set_event_handler(progSrc, ^{
    int d = dispatch_source_get_data(progSrc);
    _progressView.progress = (float)d / max;
  });
}
```

Set the event handler just like before— again, handler code is inline making maintenance easier

For UI updates

```
- (void) setupProgressWithMax:(float)max {
  progSrc = dispatch_source_create(
        DISPATCH_SOURCE_TYPE_DATA_ADD, 0, 0);
  dispatch_set_target_queue(progSrc,
        dispatch_get_main_queue());
  dispatch_source_set_event_handler(progSrc, ^{
    int d = dispatch_source_get_data(progSrc);
    _progressView.progress = (float)d / max;
  });
}
```

Get current accumulated value through this API
Only call it from within a source's event block

For UI updates

```
- (void) setupProgressWithMax:(float)max {
  progSrc = dispatch_source_create(
        DISPATCH_SOURCE_TYPE_DATA_ADD, 0, 0);
  dispatch_set_target_queue(progSrc,
        dispatch_get_main_queue());
  dispatch_source_set_event_handler(progSrc, ^{
    int d = dispatch_source_get_data(progSrc);
    _progressView.progress = (float)d / max;
  });
}
```

It’s a closure, so you don’t need to keep max value around anywhere, the blocks runtime does it for you

For UI updates

```
- (void) receivedData:(NSData*)data {
    dispatch_source_merge_data(progSource,
            [data length]);
}
```

To update the source, your other code just needs to use this API to hand in a value to the accumulator
The event handler will not be queued if it is already present on its target queue
Many merges can happen while target queue is busy, single event handler gets *current* value

Availability

Tempting to use Reachability APIs the wrong way

Don’t call us, we’ll call you

- **Do:**
  1. Attempt to use the network
  2. Check the error value

Do’s and don’ts— should use network first and check error, not test for reachability first

Don’t call us, we’ll call you

- **Do:**
  1. Attempt to use the network
  2. Check the error value
- **Don’t:**
  1. Check availability first
  2. Use network if available

Do’s and don’ts— should use network first and check error, not test for reachability first

- Reachability APIs describe the network state *as it is now*.
- Network connections are dropped automatically by the OS when they haven't been used for a while
- May have 3G available but not powered up
- May have a WiFi network nearby
- The WiFi network might need user interaction to initiate a connection

Reachability returns the truth— and network hardware might simply not be connected
Doesn't mean it can't connect— means it hasn't done so already

The Right Way

- Assume the network is available— try to use it
  - The OS will attempt to raise a WWAN or WiFi connection
  - Presents WiFi choices if in a WiFi app
  - Prompts for passwords
  - Presents modal dialog for captive network logon

Just Do It
System handles all the details of making the network connection
Only works if you use CFNetwork or higher
Wrap BSD sockets in CFSocketRef, use CFSocketConnectToAddress()

The Right Way

- Assume the network is available— try to use it
  - The OS will attempt to raise a WWAN or WiFi connection
  - Presents WiFi choices if in a WiFi app
  - Prompts for passwords
  - Presents modal dialog for captive network logon
- **Note: automatic network hardware bringup requires the use of CFNetwork or ObjC networking APIs (not raw BSD sockets).**

Just Do It
System handles all the details of making the network connection
Only works if you use CFNetwork or higher
Wrap BSD sockets in CFSocketRef, use CFSocketConnectToAddress()

The Right Way

- If a connection is established, your call goes through transparently
- If the connection fails, you'll receive an error
  - Error codes are found in:
    - `<Foundation/NSURLErrors.h>`
    - `<CFNetwork/CFNetworkErrors.h>`
  - If the error indicates 'no internet' or 'no connection', then proceed to **Plan B**

It Just Works
If no network, you'll hear about it
NOW you can install a reachability observer

Plan B

- Your network connection failed
- ***Now*** you can install a reachability monitor
  - This will tell you when the network status changes
  - A positive reachability notification only means you ***might*** be able to use the network, not ***can*** use it
  - It's a good idea to periodically retry your network calls to potentially prompt a connection
    - The network only goes up on request, and there's no guarantee that another application will cause that to happen

Observer will tell you that it's worth *trying* to connect again, doesn't guarantee success
If nothing else on system tries to use network, no connection will happen
Worthwhile to retry occasionally to make system attempt connection again

Optimized Networking and Data Handling in iOS 4

Jim Dovey
iTeam Lead Developer
Kobo Inc.
http://alanquatermain.net

That’s All Folks— questions?