CSCI-1680
Network Programming II

Rodrigo Fonseca

Today

- **Network programming**
  - Programming Paradigms
  - Programming libraries
- **Final project**

Low-level Sockets

- **Address Family AF_PACKET**
  - Socket type: SOCK_RAW
    - See link-layer (Ethernet) headers. Can send broadcast on a LAN. Can get/create non-IP packets
  - Socket type: SOCK_DGRAM
    - See IP headers. Can get protocols other than TCP/UDP: ICMP, SCTP, DCCP, your own…
    - Can cook your own IP packets
  - Must have root privileges to play with these

Building High Performance Servers

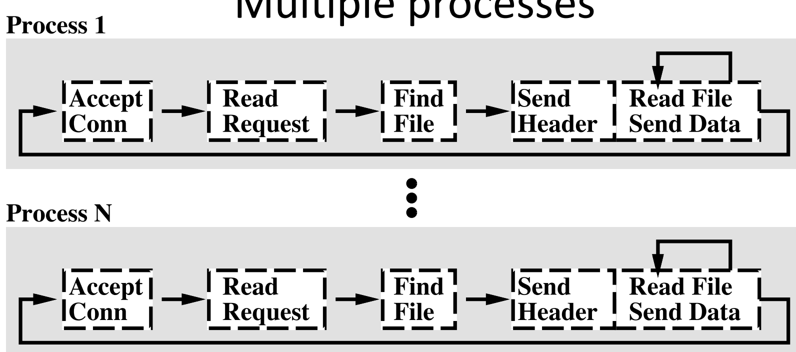
The need for concurrency

- **How to improve throughput?**
  - Decrease latency (throughput α 1/latency)
  - Hard to do!
    - Optimize code (this you should try!)
    - Faster processor (no luck here, recently)
    - Speed of light isn't changing anytime soon…
    - Disks have to deal with things like inertia!
  - Do multiple things at once
- **Concurrency**
  - Allows overlapping of computation and I/O
  - Allows use of multiple cores, machines

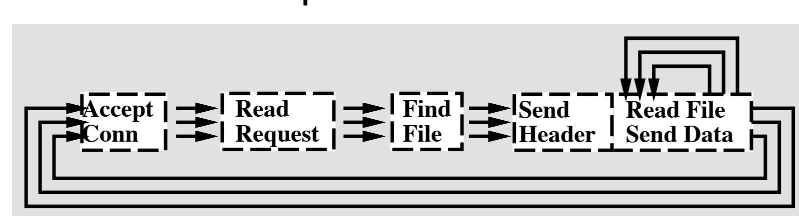
High-performance Servers

Common Patterns

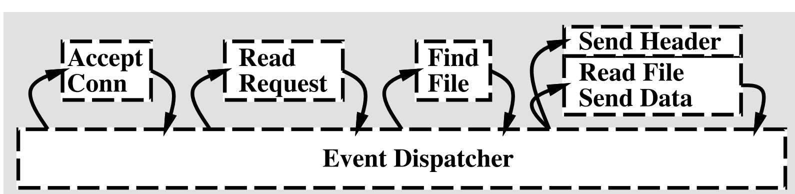
Multiple processes



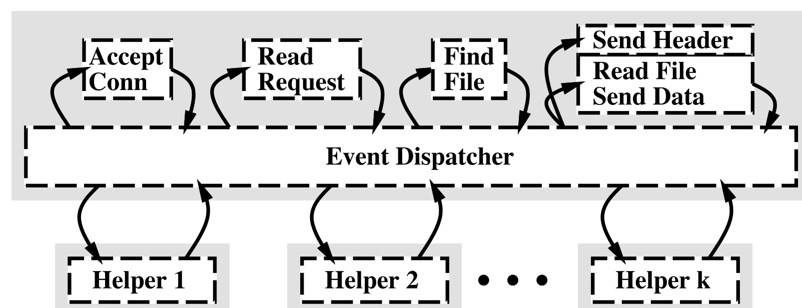
Multiple Threads



Single Process Event Driven



Single Process Event Driven with Helpers

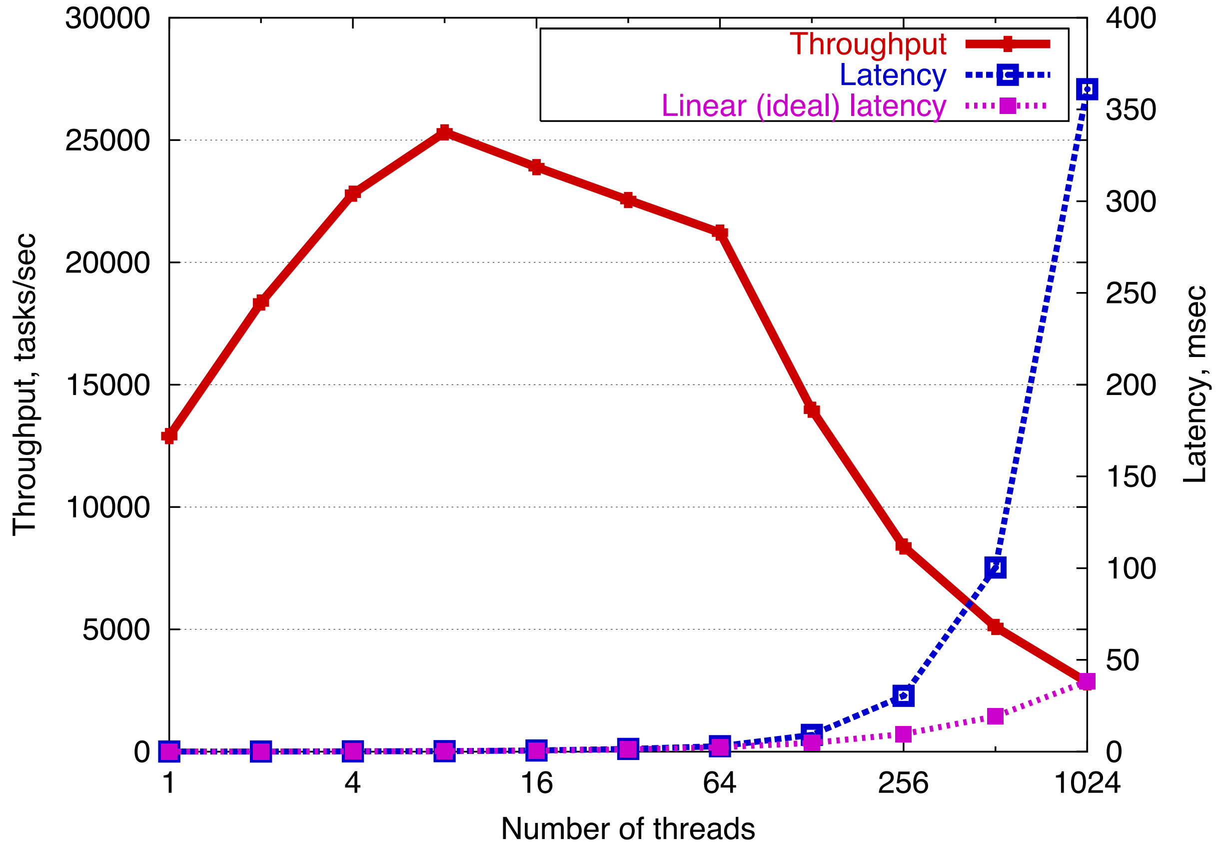


Figures from Pai, et al., 1999 "Flash: An efficient and portable Web server"

Threads

- **Usual model for achieving concurrency**
- **Uniform abstraction for single and multiple cores**
- **Concurrency with locks/mutexes**
  - Threads may block, hold locks for long time
- **Easy to reason about**
  - Each thread has own stack
- **Strong support from OS, libraries, debuggers**
- **Traditionally, problems with more than a few 100 threads**
  - Memory overhead, O(n) operations

Performance, Thread-based server



From Welsh, et al., SOSP 2001 "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services

Events

- **Small number of threads, one per CPU**
- **Threads do one thing:**

```
while(1) {
    get event from queue
    Handle event to completion
}
```

- **Events are network, I/O readiness and completion, timers, signals**
  - Remember select()?
- **Assume event handlers never block**
  - Helper threads handle blocking calls, like disk I/O

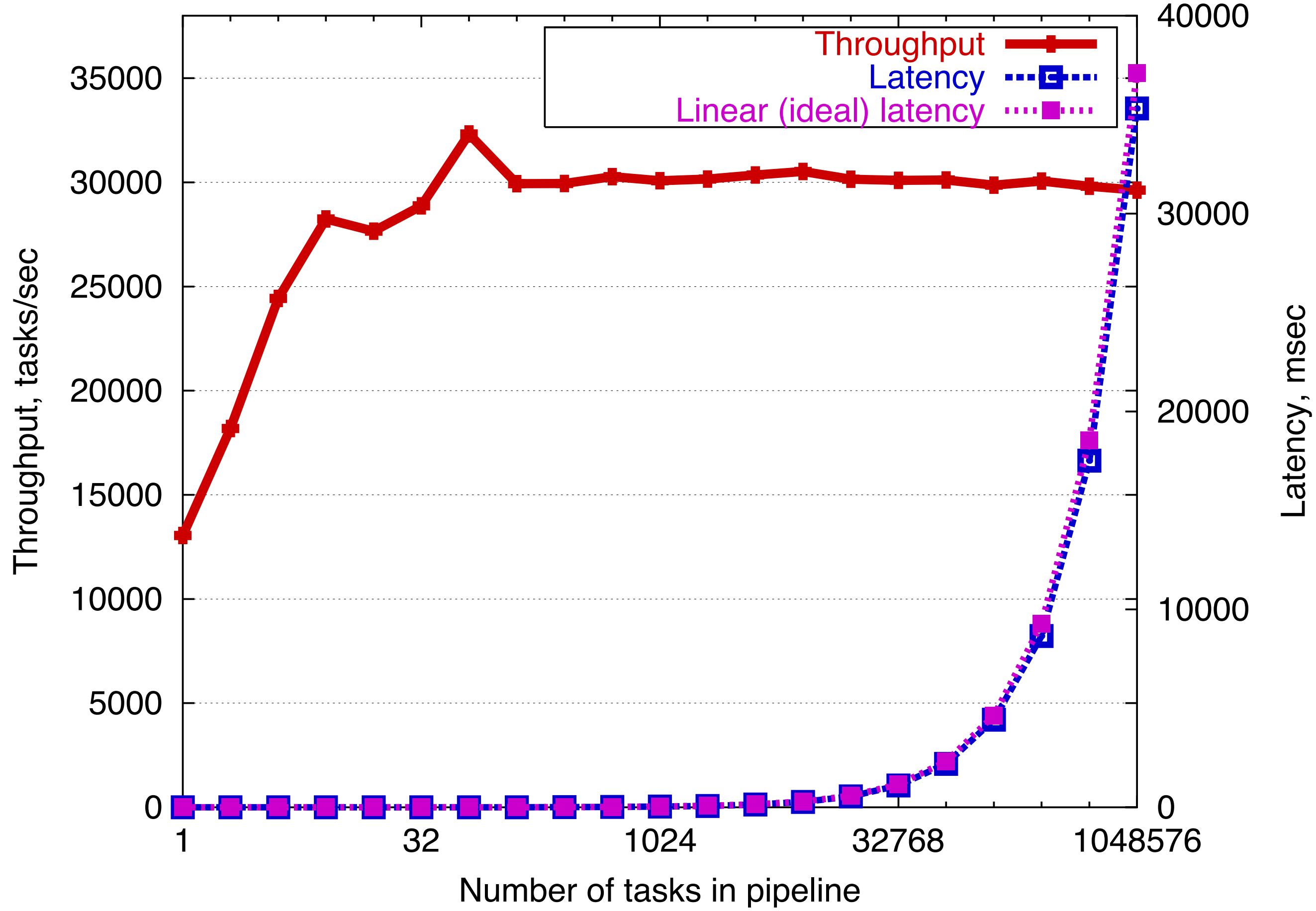
Events

- **Many works in the early 2000's claimed that events are needed for high performance servers**
  - E.g., Flash, thttpd, Zeus, JAWS web servers
- **Indeed, many of today's fastest servers *are* event-driven**
  - E.g., OKCupid, lighttpd, nginx, tornado

Lighttpd: "Its event-driven architecture is optimized for a large number of parallel connections"

Tornado: "Because it is non-blocking and uses epoll, it can handle thousands of simultaneous standing connections"

Performance, Event-Driven Web server



From Welsh, et al., SOSP 2001 "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services

Flash Web Server

- **Pai, Drushel, Zwaenepoel, 1999**
- **Influential work**
- **Compared four architectures**
  - Multi-process servers
  - Multi-threaded servers
  - Single-process event-driven
  - Asymmetric Multi-process event driven
- **AMPED was the fastest**

Events (cont)

- **Highly efficient code**
  - Little or no switching overhead
  - Easy concurrency control
- **Common complaint: hard to program and reason about**
  - For people and tools
- **Main reason:** ***stack ripping***

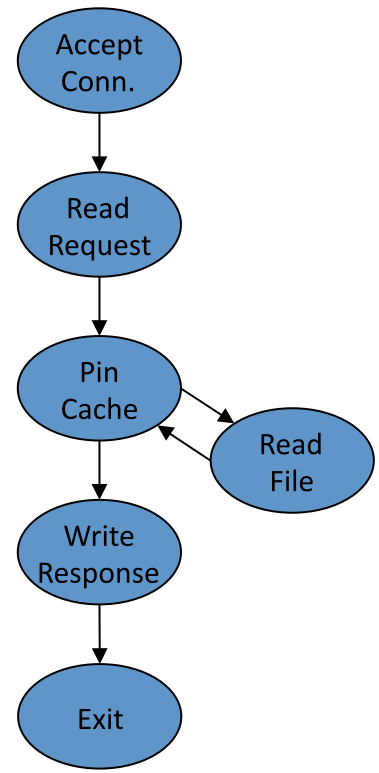
Events criticism: control flow

- **Events obscure control flow**
  - For programmers *and* tools

Threads	*Events*

thread_main(int sock) {
    struct session s;
    accept_conn(sock, &s);
    read_request(&s);
    pin_cache(&s);
    write_response(&s);
    unpin(&s);
}
 	
pin_cache(struct session *s) {
    pin(&s);
    if(!in_cache(&s))
        read_file(&s);
}
	CacheHandler(struct session *s) {
	    pin(s);
	    if(!in_cache(s)) ReadFileHandler.enqueue(s);
	    else ResponseHandler.enqueue(s);
	}
	RequestHandler(struct session *s) {
	    …; CacheHandler.enqueue(s);
	}
	. . .
	ExitHandlerr(struct session *s) {
	    …; unpin(&s); free_session(s);
	}
	AcceptHandler(event e) {
	    struct session *s = new_session(e);
	    RequestHandler.enqueue(s); }

Web Server



Events criticism: Exceptions

- **Exceptions complicate control flow**
  - Harder to understand program flow
  - Cause bugs in cleanup code

Threads

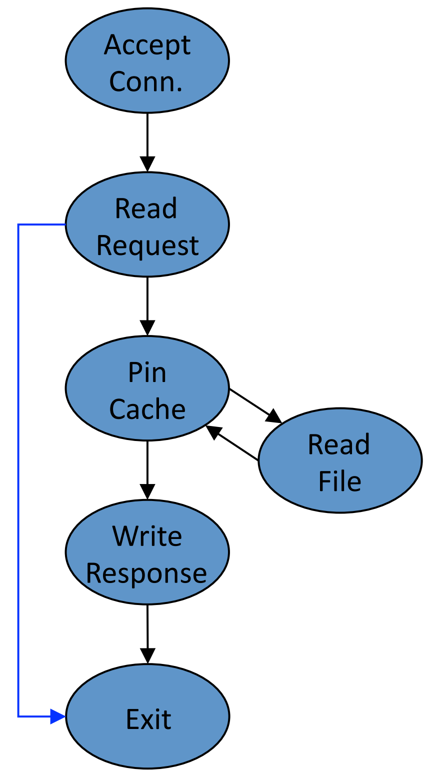
```
thread_main(int sock) {
    struct session s;
    accept_conn(sock, &s);
    if( !read_request(&s) )
        return;
    pin_cache(&s);
    write_response(&s);
    unpin(&s);
}

pin_cache(struct session *s) {
    pin(&s);
    if( !in_cache(&s) )
        read_file(&s);
}
```

Events

```
CacheHandler(struct session *s) {
    pin(s);
    if( !in_cache(s) )  ReadFileHandler.enqueue(s);
    else                ResponseHandler.enqueue(s);
}
RequestHandler(struct session *s) {
    ...; if( error ) return;  CacheHandler.enqueue(s);
}
. . .
ExitHandlerr(struct session *s) {
    ...;  unpin(&s);  free_session(s);
}
AcceptHandler(event e) {
    struct session *s = new_session(e);
    RequestHandler.enqueue(s); }
```

Web Server



Events criticism: State Management

- **Events require manual state management**
- **Hard to know when to free**
  - Use GC or risk bugs

Threads

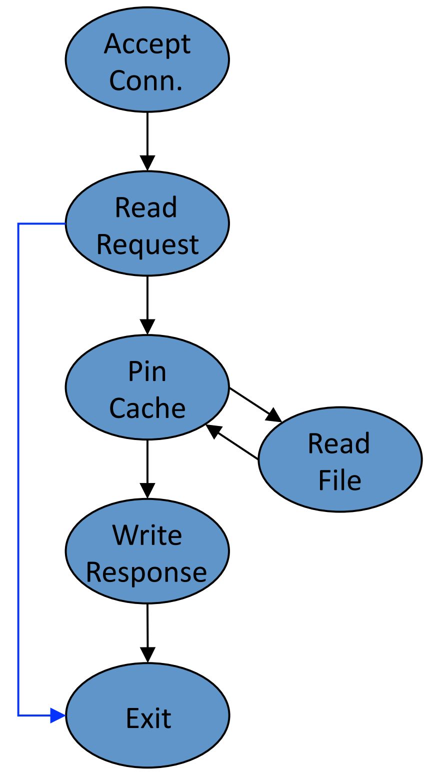
```
thread_main(int sock) {
    struct session s;
    accept_conn(sock, &s);
    if( !read_request(&s) )
        return;
    pin_cache(&s);
    write_response(&s);
    unpin(&s);
}

pin_cache(struct session *s) {
    pin(&s);
    if( !in_cache(&s) )
        read_file(&s);
}
```

Events

```
CacheHandler(struct session *s) {
    pin(s);
    if( !in_cache(s) )  ReadFileHandler.enqueue(s);
    else                ResponseHandler.enqueue(s);
}
RequestHandler(struct session *s) {
    ...; if( error ) return;  CacheHandler.enqueue(s);
}
. . .
ExitHandlerr(struct session *s) {
    ...;  unpin(&s);  free_session(s);
}
AcceptHandler(event e) {
    struct session *s = new_session(e);
    RequestHandler.enqueue(s); }
```

Web Server

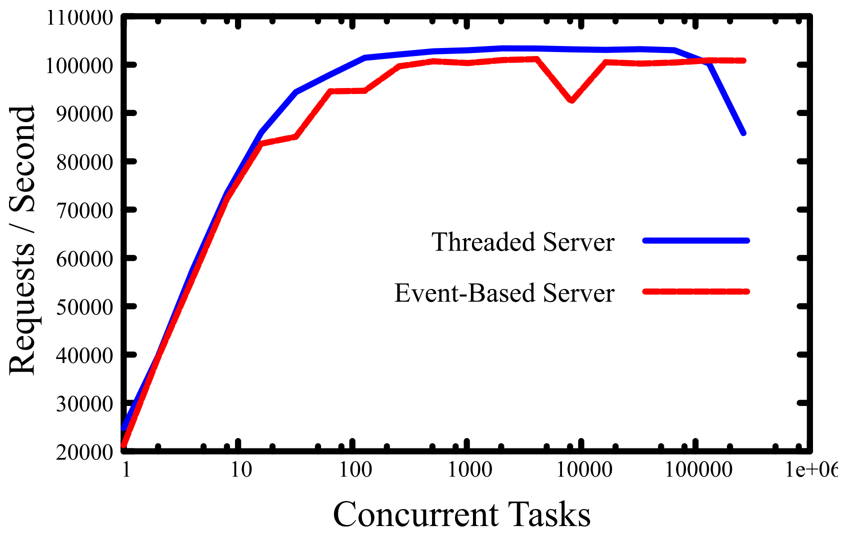


Usual Arguments

- **Events:**
  - Hard to program (stack ripping)
  - Easy to deal with concurrency (cooperative task management)
    - Shared state is more explicit
  - High performance (low overhead, no switching, no blocking)
- **Threads**
  - Easy to reason about flow, state (automatic stack management)
  - Hard to deal with concurrency (preemptive task management)
    - Everything is shared
  - Lower performance (thread switching cost, memory overhead)

Capriccio (2003)

- **Showed threads can perform as well as events**
  - Avoid O(n) operations
  - Cooperative lightweight user-level threads
    - (still one kernel thread per core)
  - Asynchronous I/O
    - Handled by the library
  - Variable-length stacks
  - The thread library runs an event-based system underneath!



Artificial Dichotomy!

- **Old debate! Lauer and Needham, 78**
  - Duality between process-based and message-passing
  - Updated by the Capriccio folks, 2003

Threads	*Events*
▪ Monitors	▪ Event handler & queue
▪ Exported functions	▪ Events accepted
▪ Call/return and fork/join	▪ Send message / await reply
▪ Wait on condition variable	▪ Wait for new messages

- **Performance should be similar**
  - No inherent reason for threads to be worse
  - Implementation is key

Artificial Dichotomy

- **Threads**
  - Preemptive multitasking
  - Automatic stack management
- **Events**
  - Cooperative multitasking
  - Manual stack management (stack ripping)
- **Adya, 2002: you can choose your features!**
  - They show that you can have cooperative multitasking with automatic stack managment

Adya, A. et al., 2002. "Cooperative Task Management without Manual Stack Managementor, Event-driven Programming is Not the Opposite of Threaded Programming

Threads vs. Events

- **Today you still have to mostly choose either style (complete packages)**
  - Thread-based servers very dependent on OS, threading libraries
- **Some promising directions!**
  - TAME allows you to write sequential C++ code (with some annotations), converts it into event-based
  - Scala (oo/functional language that runs on the JVM) makes threaded and event-based code look almost identical

Popular Event-Based Frameworks

- **libevent**
- **libasync (SFS, SFS-light)**
- **Javascript**
  - All browser code
  - Node.js at the server side
- **GUI programming**

Some available libraries

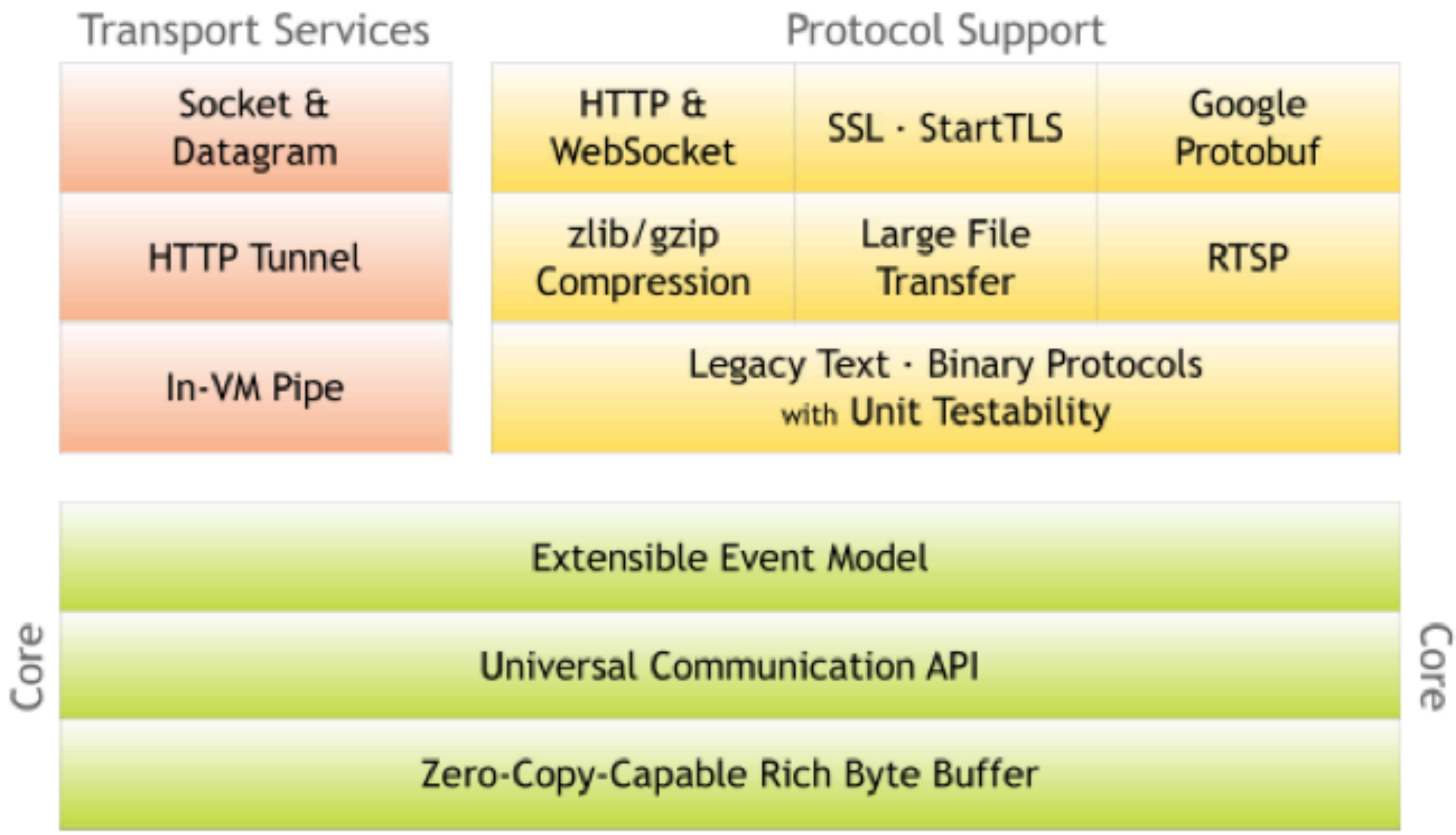
With material from Igor Ganichev

Python

- **Rich standard library**
  - url/http/ftp/pop/imap/smtp/telnet
  - SocketServer, HTTPServer, DocXMLRPCServer, etc
- **Twisted**
  - Very popular
  - Has *a lot* of stuff, but quite modular
  - Event-driven, many design patterns. Steep learning curve…
  - Well maintained and documented

Java

- **Mature RPC library: RMI**
- **River: RMI + service discovery, mobile code**
- **Java.NIO**
  - High-level wrapping of OS primitives
    - Select -> Selector . Socket -> Channel
  - Good, efficient buffer abstraction
- **Jetty**
  - Extensible, event-driven framework
  - High-performance
  - Avoid unnecessary copies
  - Other side doesn't have to be in Java


Transport Services
Protocol Support
Socket & Datagram
HTTP & WebSocket
SSL · StartTLS
Google Protobuf
HTTP Tunnel
zlib/gzip Compression
Large File Transfer
RTSP
In-VM Pipe
Legacy Text · Binary Protocols with Unit Testability
Extensible Event Model
Universal Communication API
Zero-Copy-Capable Rich Byte Buffer
Core
Core

C

- **Sockets!**
- **Direct access to what the OS provides**
- **Libevent**
  – Simple, somewhat portable abstraction of select() with uniform access to events: I/O, timers, signals
  – Supports /dev/poll, kqueue(2), event ports, select(2), poll(2) and epoll(4).
  – Well maintained, actively developed
  – Behind many very high-performance servers
    - Memcached

C++

- **Boost.ASIO**
  - Clean, lightweight, portable abstraction of sockets and other features
  - Not a lot of higher-level protocol support
  - Has support for both synchronous and asynchronous operations, threads (from other parts of Boost)
- **Others: ACE, POCO**

ICE

- **Cross-language middleware + framework**
  - Think twisted + protocol buffers
- **Open source but owned by a company**
- **SSL, sync/async, threads, resource allocation, firewall traversal, event distribution, fault tolerance**
- **Supports many languages**
  - C++, Java, .NET-languages (such as C# or Visual Basic), Objective-C, Python, PHP, and Ruby

Other "cool" approaches

- **Erlang, Scala, Objective C**
  - Support the Actor model: program is a bunch of actors sending messages to each other
  - Naturally extends to multi-core and multiple machines, as sending messages is the same
- **Go**
  - Built for concurrency, uses 'Goroutines', no shared state
  - "Don't share memory to communicate, communicate to share memory"

Node.js

- Javascript server framework
- Leverages highly efficient Chrome V8 Javascript JIT runtime
- Completely event-based
- Many high-level libraries

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(8124, "127.0.0.1");
console.log('Server running at http://127.0.0.1:8124/');
```

Final Assignment

Final Project

- **IP Over DNS**
- **Problem: suppose you connect to a network that only gives you (very) limited access:**

  recursive DNS queries through local DNS server
- **Can you use this to route *any* IP traffic?**

Disclaimer: this is provided as an educational exercise so you can learn how tunnels, NATs, and virtual interfaces work. You should not use these techniques to gain access to unauthorized network resources.

IP Over DNS

- **DNS queries can carry information: domain name is arbitrary string**
  - Maximum 255 characters
  - Name is sequence of labels, each label max 63 characters
  - Labels preceded by single byte length
  - Terminated by a 0-length label (0 byte)

IP over DNS

- **DNS Responses can carry arbitrary information**
  - In TXT records
  - Maximum length?
    - Maximum UDP DNS packet is 512 bytes
    - Other restrictions may be imposed by DNS servers, e.g. maximum 255 bytes per TXT record, maximum number of TXT records per packet… Should test with your recursive resolver.
  - Should you repeat the query?
    - Not required by the standard (RFC1035)
    - Common practice (e.g. Bind) is to reject the response if it doesn't match the query, but again, YMMV.

Talk about possible solution

Some questions

- **How to encode data?**
- **Virtual interfaces: TUN or TAP?**
- **Client: setting up routes**
- **MTU**
- **Server: what to do with the packets you receive?**
  – Linux has support for NATs
- **Asymmetries**
  – Only client can initiate requests
  – What if server has many 'responses' to send?

Some Resources

- **TUN/TAP Interfaces**
  - VTUN
- **DNS Packets**
  - You can build your own (RFC1035)
  - There are libraries to help (e.g. FireDNS)
- **Base64 Encoding**
  - http://www.ietf.org/rfc/rfc3548.txt
- **Linux Routing and NAT**
  - Route configuration and basic NAT: iproute2
  - More sophisticated NAT: iptables
  - BE CAREFUL NOT TO LOSE CONNECTIVITY WHEN YOU CHANGE ROUTES!