Duplicate Aware Scheduling

Joshua Pfosi & Nitesh Gupta

Table of Contents

List of Figures

List of Tables

1 Introduction

A common technique for application scalability at Internet-sized proportions is distributed systems of computers which communicate over some network. By leveraging the collective computing power of many machines, requests can be served far faster than even the most powerful single cores. A classic example of this paradigm is load balancing. Load balancing is an attempt to distribute workloads across multiple resources to reduce latency and maximize throughput. A load balancer is logically centralized entity which accepts incoming service requests and maps them to a distributed network of servers in such a way as to minimize average load on each server. There exist many load balancing algorithms such as random, round robin, power of n choice etc. Each scheme excels under different request patterns or workloads and is suboptimal or inadequate for others. This manifests as a skewing of load (requests / sec) across servers. Incoming requests to a heavily loaded server are, by definition, served less quickly. This results in what are known as "stragglers", or portions of a distributed request which stall the entire request as they were erroneously directed to a heavily loaded server. While load balancers attempt to reduce the likelihood of stragglers, this work attempts to minimize the effects of them.

The concept is to reduce the effects of poorly routed primary requests by sending multiple instances, or duplicates, of that request to different servers. If done carefully, the set of requests (the primary and the duplicates) will not all be directed to loaded servers. Therefore, at least one will be served in the expected amount of time, and a straggler will not hold up an entire request. Clearly, by doubling or tripling network traffic naively, this technique would increase server load significantly and be counterproductive. To meet this challenge, we propose duplicate aware scheduling. That is, proper instrumentation of specific load-sensitive applications to understand and prioritize primary and duplicate requests differently so that stragglers are minimized but duplicate requests are not served before primaries. We hypothesize that by reducing the effects of stragglers on extremely loaded servers, an increase in throughput and a decrease in average latency will be observed.

2 Experimental Setup

To verify the hypothesis, we required a method to create large and diverse workloads as well as a typical application to instrument. Fortunately, Yahoo! Cloud Serving Benchmark (YCSB) provided a workload generator for database stores which was sufficiently robust from which we could work. We chose Memcached as the application to instrument. Memcached is a distributed memory object caching system implemented as a in-memory key-value store (Fitzpatrick, 2003). We selected it for its relatively simple high level design and lack of replication strategy. Memcached is also built upon an event notification library, libevent, which implements a priority API which met our desired priority semantics.

2.1 Top Level Diagram

Fig. 1, below, presents the high level design for our experiments. YCSB behaves logically as one client but may be distributed among multiple physical computers to achieve desired server loads. Memcached is run independently on four different servers without a backend database layer. Each server is responsible for a fourth of the primary key space, and a fourth of the duplicate key space. With carefully chosen hash functions, the mass majority of duplicate and primary mappings should be distinct for a given key.

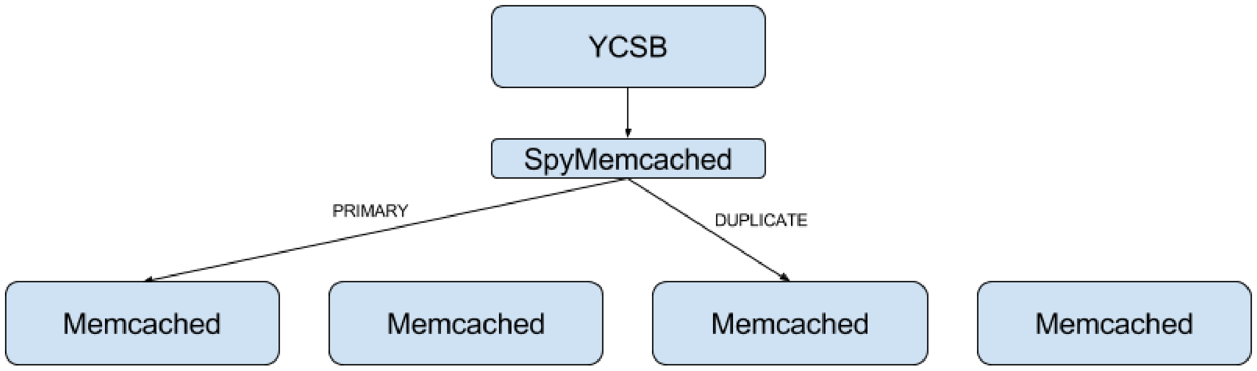

Figure 1: Top level diagram of overall modules

YCSB generates a workload, then a Java Memcached client, Spymemcached, implements the requests to Memcached servers. A requested key is hashed to choose a memcached instance to fulfill that request. Without modification, Spymemcached accepts a list of Memcached hosts and creates TCP connections to those hosts. Requests are hashed consistently via a selected hash function and the response is returned synchronously to YCSB which benchmarks the transaction. Throughput and latency statistics are aggregated and output by YCSB.

2.2 Duplicate Awareness

This work modifies both Spymemcached and Memcached to be *duplicate aware*. This implies instrumenting the former to hash and send duplicate requests distinctly from primary requests, and the latter to recognize and prioritize a request as PRIMARY or DUPLICATE. For every `insert` operation a YCSB workload generates, the YCSB client asynchronously makes two `set` requests to Spymemcached, one primary and one duplicate. For every `read` operation, YCSB asynchronously makes two `get` requests to Spymemcached, one primary and one duplicate. Once the `get` or SET resolve, YCSB cancels the remaining request and returns. Therefore, if the PRIMARY request is lagging, the DUPLICATE request can reduce latency by resolving first.

2.2.1 Yahoo! Cloud Serving Benchmark

We use YCSB to create workloads to stress the duplcate aware memcached instances. The critical use case in which duplicate aware scheduling shines is in the presence of skewed workloads. That is, when one or more servers are significantly more loaded than others. This allows duplicate requests, sent to underutilized resources, to fulfill the otherwise blocked request. To verify we have loaded the servers properly, we can leverage Memcached's `stats` API to query for `get` and `set` operation totals for a given workload. Comparing such plots with and without duplicate requests provides confidence that the hashing algorithms are distributing requests appropriately. A latent requirement for the utility of these graphs is a deterministic key sequence. By inspecting YCSB source code, however, it appears an additively increasing sequence starting at 0 is used by default. Therefore, the plots of requests per server per workload are valid.

2.2.2 Spymemcached

Spymemcached uses `getNodeForKey` to hash a key and return a `MemcachedNode`. Overwriting this method to hash DUPLICATE keys (i.e. `dupKey = ‘‘dup__’’ + key`) differently is sufficient to distribute the duplicate load differently. The code changes to implement this change are small and can be found in Appendix C.

2.2.3 Memcached

Memcached, on the other hand, was slightly more complex. Memcached listens for incoming connections in a master thread and dispatches new connections in a round robin fashion to a fixed number (typically 4) of worker threads. Each worker, in turn, listens for traffic on each of its open connections and transitions through a state machine based on incoming requests. The “listening” construct is implemented via Libevent which supports an event callback for configurable triggers on file descriptors. Memcached uses this to poll for actionable events on the TCP sockets of each worker thread. This way each worker thread sees a synchronized, serialized queue of requests to which it responds. Without modification, Memcached treats all incoming requests equivalently. To make it duplicate aware, however, requires a priority queue where PRIMARY requests are served before DUPLICATEs. Libevent allows a client to prioritize specific events over others. Events can be distinguished only by the file descriptor to which they are associated or the event flags for which they activate (Mathewson, 2009). As we are strictly using EV_READ, we must use separate file descriptors and therefore distinct TCP connections for PRIMARY versus DUPLICATE requests. A simple way to accomplish this is to leverage Memcached existing dispatching algorithm, round robin, and hardcode that the events on the first connection given to any worker thread are high priority, and every subsequent connection’s events are low priority. The code changes to implement this change are small and can be found in Appendix B.

2.3 Emulab

For consistency, all experiments were conducted on a networked testbed called Emulab, on Ubuntu 14.04 LTS (64-bit), over a 10Gb LAN (as in Fig. 2). The experiment was entitled `dup` on the `comp150` project. See Appendix A for the NS script which constructed the desired topology.

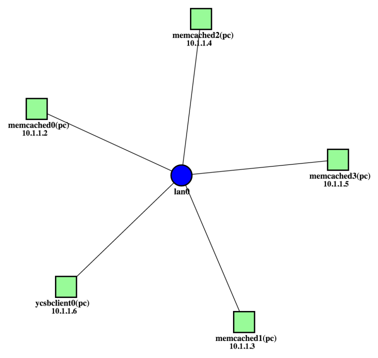

Figure 2: Emulab topology

3 Experimental Execution

The execution of experiments is entirely scripted. YCSB is prebuilt with each duplicate scheme in directories `YCSB_unaware` and `YCSB_aware` for control and scheduled duplicates, respectively. Running `sh run-experiments.sh` executes a number of trials of each of these test cases on all workloads under the YCSB workload directory for multiple request sizes. Specifically, 1000 records were loaded, and 10000 operations were executed. Record sizes were 1 kB and 5 kB. These values were selected to allow the experiments to run in a reasonable amount of time, and should be tweaked by future researchers.

3.1 Utilities

In addition, there are many provided utilities, summarized in Table 1.

Usage	Description
`sh kill-experiment.sh <NUM_CLIENTS> <NUM_SERVERS>`	Kills all running `memcached` and `ycsb` instances on all Emulab machines.
`sh build-{memcached,ycsb,libevent}.sh`	Compresses, copies, decompresses and builds the desired software on the appropriate Emulab machines.
`sh build-emulab.sh <NUM_CLIENTS> <NUM_SERVERS>`	After swap in, this command will install and build all dependencies and software to run experiments.
`sh run-experiments.sh`	Runs all types of experiments through a variety of parameters and outputs data to subdirectories for analysis.
`sh make-dup-aware.sh`	Compiles `memcached` to be duplicate aware on the appropriate Emulab machines.
`sh run-experiments.sh`	Runs all experiments for a variety of parameters.
`sh plot-{load,throughput}.sh <XLABEL> <YLABEL> <TITLE> <DATA> <OUTPUT>`	Generates basic scatter plots.

Table 1: Summarizes utilities provided as scripts

4 Results

Using the setup from Fig. 2, a single YCSB client made requests to four Memcached instances and recorded throughput. To test the efficacy of duplicate aware scheduling, we developed a set of workloads and measured the throughput and latency with and without duplicate aware scheduling as a function of load (i.e. YCSB `fieldlength`). Fig. 3 illustrate the load on each server for the control (black points) as well as with DUPLICATE requests (yellow points).

Fig. 4 depicts the variance in throughput seen under duplicate aware scheduling. We expected duplicate requests to reduce the effects of stragglers and therefore increase throughput. In some instances, this is observed, however, this is likely due to sampling error. With such low throughput numbers (typical Memcached systems see on the order of tens or hundreds of thousands), statistical variance in throughput obscures our results significantly.

5 Future Work

This work laid the ground work for future experimentation with Memcached and duplicate aware scheduling. We used a straightforward and naive duplicate scheduling technique. Future work could benefit from using other techniques such as dynamically sending DUPLICATE packets based on measured load of a given server.

Additionally, this work was majorly compromised by inexplicably low throughput on the Memcached servers. This reduces the resolution of our results and the feasibility of testing. Switching to another load generator or testbed to improve throughput and reduce latency may demonstrate the true efficacy of duplicate aware scheduling.

Finally, for simplicity we only used four Memcached servers and a single client. While sufficient, spreading load over multiple servers would exaggerate the skewing effect of the `latest` request distribution of YCSB and therefore illustrate the benefits of duplicate aware scheduling more clearly.

6 Conclusion

In conclusion, this paper contributes a robust, automated environment that, combined with a remote testbed such as Emulab, can be used to experiment with and verify predictions associated with duplicate aware scheduling. Unfortunately, due to technical and logistical limitations, the original hypothesis could neither be accepted nor refuted, but by leveraging the scripts documented here, future work is better poised to do so.

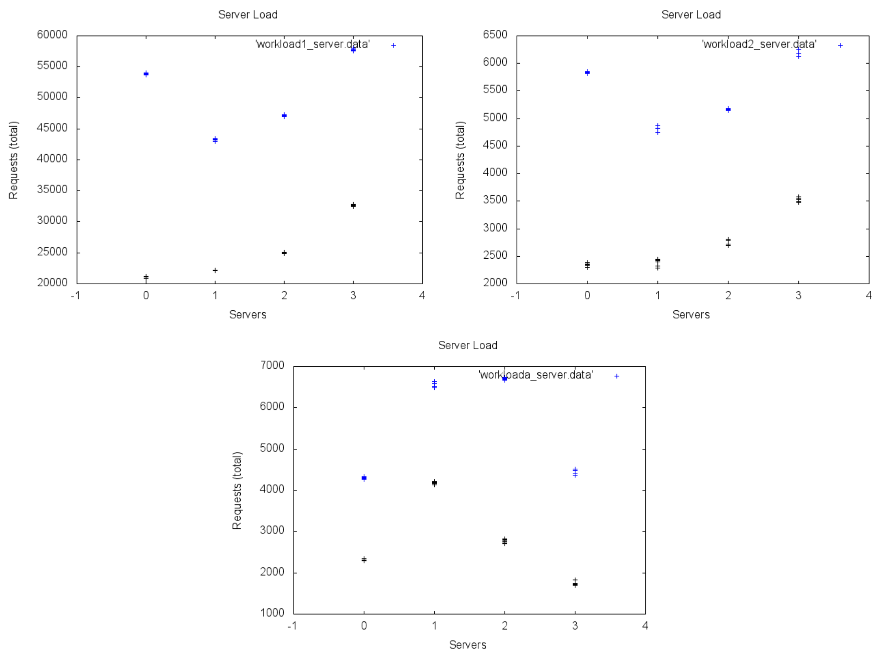

Figure 3: The servers' load for PRIMARY-only and DUPLICATE requests. Load under, from right to left, workload 1, 2, and a.

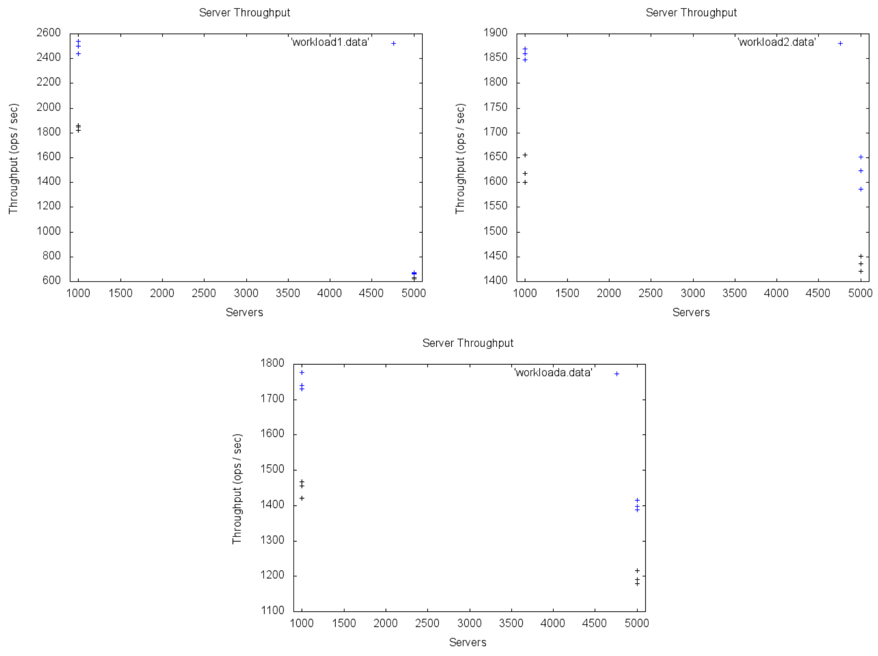

Figure 4: The client's throughput as a function of record size for control and duplicate-aware (black and yellow, respectively).

A emulab.tcl

```
# This is a simple ns script developed from https://wiki.emulab.net/wiki/Emulab/wiki/
    Tutorial
set ns [new Simulator]
source tb_compat.tcl

set memcached0 [$ns node]
set memcached1 [$ns node]
set memcached2 [$ns node]
set memcached3 [$ns node]
set ycsbclient0 [$ns node]
set ycsbclient1 [$ns node]
set ycsbclient2 [$ns node]
set ycsbclient3 [$ns node]

set lan0 [$ns make-lan "$memcached0␣$memcached1␣$memcached2␣$memcached3␣$ycsbclient0␣
    $ycsbclient1␣$ycsbclient2␣$ycsbclient3␣" 10Gb 0ms]

# Set the OS on a couple.
tb-set-node-os $memcached0 UBUNTU14-64-STD
tb-set-node-os $memcached1 UBUNTU14-64-STD
tb-set-node-os $memcached2 UBUNTU14-64-STD
tb-set-node-os $memcached3 UBUNTU14-64-STD
tb-set-node-os $ycsbclient0 UBUNTU14-64-STD
```

```
tb-set-node-os $ycsbclient1 UBUNTU14-64-STD
tb-set-node-os $ycsbclient2 UBUNTU14-64-STD
tb-set-node-os $ycsbclient3 UBUNTU14-64-STD

# Allows routing
$ns rtproto Static

$ns run
```

B Memcached Modifications

```
diff --git a/memcached.c b/memcached.c
index ff1af50..898f2f4 100644
--- a/memcached.c
+++ b/memcached.c
@@ -353,7 +353,7 @@ static const char *prot_text(enum protocol prot) {
 conn *conn_new(const int sfd, enum conn_states init_state,
                const int event_flags,
                const int read_buffer_size, enum network_transport transport,
-               struct event_base *base) {
+               struct event_base *base, const int priority) {
     conn *c;

     assert(sfd >= 0 && sfd < max_fds);
@@ -472,6 +472,13 @@ conn *conn_new(const int sfd, enum conn_states init_state,
     c->noreply = false;

     event_set(&c->event, sfd, event_flags, event_handler, (void *)c);
+
+#ifdef DUP_AWARE
+    event_priority_set(&c->event, priority);
+    fprintf(stderr, "<%d event priority is %s\n", sfd, (priority == PRIMARY) ? "PRIMARY"
     : "DUPLICATE");
+#else
+    (void)priority;
+#endif
     event_base_set(base, &c->event);
     c->ev_flags = event_flags;

@@ -4595,7 +4602,7 @@ static int server_socket(const char *interface,
         } else {
             if (!(listen_conn_add = conn_new(sfd, conn_listening,
                                              EV_READ | EV_PERSIST, 1,
-                                             transport, main_base))) {
+                                             transport, main_base, -1))) {
                 fprintf(stderr, "failed to create listening connection\n");
                 exit(EXIT_FAILURE);
             }
@@ -4742,7 +4749,7 @@ static int server_socket_unix(const char *path, int access_mask) {
     }
     if (!(listen_conn = conn_new(sfd, conn_listening,
                              EV_READ | EV_PERSIST, 1,
-                             local_transport, main_base))) {
+                             local_transport, main_base, -1))) {
        fprintf(stderr, "failed to create listening connection\n");
        exit(EXIT_FAILURE);
    }
diff --git a/memcached.h b/memcached.h
index df972f5..eae9bd6 100644
--- a/memcached.h
+++ b/memcached.h
@@ -23,6 +23,19 @@

 #include "sasl_defs.h"

+#ifdef DUP_AWARE
+
+#if defined NUM_THREADS && NUM_THREADS > 0
+#else
+#error "NUM_THREADS must be defined if DUP_AWARE"
+#endif
+
+#define NUM_PRIORITIES 2
+#define PRIMARY      0
+#define DUPLICATE    1
+
+#endif
+
 /** Maximum length of a key. */
 #define KEY_MAX_LENGTH 250

@@ -548,7 +561,7 @@ enum delta_result_type do_add_delta(conn *c, const char *key,
                                 const int64_t delta, char *buf,
                                 uint64_t *cas, const uint32_t hv);
 enum store_item_type do_store_item(item *item, int comm, conn* c, const uint32_t hv);
-conn *conn_new(const int sfd, const enum conn_states init_state, const int event_flags,
    const int read_buffer_size, enum network_transport transport, struct event_base *base
    );
+conn *conn_new(const int sfd, const enum conn_states init_state, const int event_flags,
    const int read_buffer_size, enum network_transport transport, struct event_base *base
    , const int priority);
 extern int daemonize(int nochdir, int noclose);

 #define mutex_lock(x) pthread_mutex_lock(x)
diff --git a/thread.c b/thread.c
index 7c56445..de80bb5 100644
--- a/thread.c
+++ b/thread.c
@@ -24,6 +24,7 @@ struct conn_queue_item {
    int               event_flags;
    int               read_buffer_size;
    enum network_transport   transport;
```

```
+    int              priority;
     CQ_ITEM         *next;
 };

@@ -332,6 +333,10 @@ static void setup_thread(LIBEVENT_THREAD *me) {
         exit(1);
     }

+#ifdef DUP_AWARE
+    event_priority_init(NUM_PRIORITIES);
+#endif
+
     /* Listen for notifications from other threads */
     event_set(&me->notify_event, me->notify_receive_fd,
               EV_READ | EV_PERSIST, thread_libevent_process, me);
@@ -398,7 +403,8 @@ static void thread_libevent_process(int fd, short which, void *arg) {

     if (NULL != item) {
         conn *c = conn_new(item->sfd, item->init_state, item->event_flags,
-                           item->read_buffer_size, item->transport, me->base);
+                           item->read_buffer_size, item->transport, me->base,
+                           item->priority);
         if (c == NULL) {
             if (IS_UDP(item->transport)) {
                 fprintf(stderr, "Can't listen for events on UDP socket\n");
@@ -447,12 +453,35 @@ void dispatch_conn_new(int sfd, enum conn_states init_state, int
    event_flags,
     LIBEVENT_THREAD *thread = threads + tid;

     last_thread = tid;
+#ifdef DUP_AWARE
+    fprintf(stderr, "last_thread = %d, tid = %d\n", last_thread, tid);
+#endif

     item->sfd = sfd;
     item->init_state = init_state;
     item->event_flags = event_flags;
     item->read_buffer_size = read_buffer_size;
     item->transport = transport;
+    item->priority = -1; // ignored unless DUP_AWARE
+
+#ifdef DUP_AWARE
+    if (!IS_UDP(transport)) {
+
+        if (settings.num_threads != NUM_THREADS) {
+            fprintf(stderr, "settings.num_threads must be %d for duplicate awareness"
+                    "to work in its current implementation. You specified %d "
+                    "threads.", NUM_THREADS, settings.num_threads);
+            exit(1);
+        }
+
+       static int is_priority = NUM_THREADS;
+
+       // The first set of ‘num_thread‘ connections will all be primary, while any
+       // subsequent will be duplicate.
+       item->priority = (is_priority-- > 0) ? PRIMARY : DUPLICATE;
+       fprintf(stderr, "<%d item->priority is %s\n", sfd, (item->priority == PRIMARY) ?
    "PRIMARY" : "DUPLICATE");
+    }
+#endif

     cq_push(thread->new_conn_queue, item);
```

C Spymemcached Modifications

```
diff --git a/pom.xml b/pom.xml
index 4a07d5f..c97c12e 100644
--- a/pom.xml
+++ b/pom.xml
@@ -12,7 +12,7 @@
     -->

     <modelVersion>4.0.0</modelVersion>
-    <groupId>net.spy</groupId>
+    <groupId>spy</groupId>
     <artifactId>spymemcached</artifactId>
     <version>2.999.999-SNAPSHOT</version> <!-- not used -->

diff --git a/src/main/java/net/spy/memcached/KetamaNodeLocator.java b/src/main/java/net/
    spy/memcached/KetamaNodeLocator.java
index 981dc2d..d3e4385 100644
--- a/src/main/java/net/spy/memcached/KetamaNodeLocator.java
+++ b/src/main/java/net/spy/memcached/KetamaNodeLocator.java
@@ -100,7 +100,17 @@ public final class KetamaNodeLocator extends SpyObject implements
    NodeLocator {
   }

   public MemcachedNode getPrimary(final String k) {
-    MemcachedNode rv = getNodeForKey(hashAlg.hash(k));
+    long hashVal;
+
+    if (k.startsWith("dup__")) {
+      hashVal = hashAlg.hash(k);
+    }
+    else {
+      hashVal = DefaultHashAlgorithm.valueOf("CRC_HASH").hash(k);
+    }
+
+    MemcachedNode rv = getNodeForKey(hashVal);
+
     assert rv != null : "Found no node for key " + k;
     return rv;
   }
@@ -121,6 +131,8 @@ public final class KetamaNodeLocator extends SpyObject implements
    NodeLocator {
        hash = tailMap.firstKey();
      }
    }
+
+    hash = getKetamaNodes().firstKey();
     rv = getKetamaNodes().get(hash);
     return rv;
   }
```

References

Fitzpatrick, B. (2003). *Memcached.* Retrieved from `http://memcached.org/`

Mathewson, N. (2009). *Fast portable non-blocking network programming with libevent.* Retrieved from `http://www.wangafu.net/ nickm/libevent-book/Ref4_event.html`

Mathewson, N., & Provos, N. (2003). *Libevent.* Retrieved from `http://libevent.org/`

of Utah, U. (2003). *Emulab.* Retrieved from `http://www.emulab.net/index.php3`