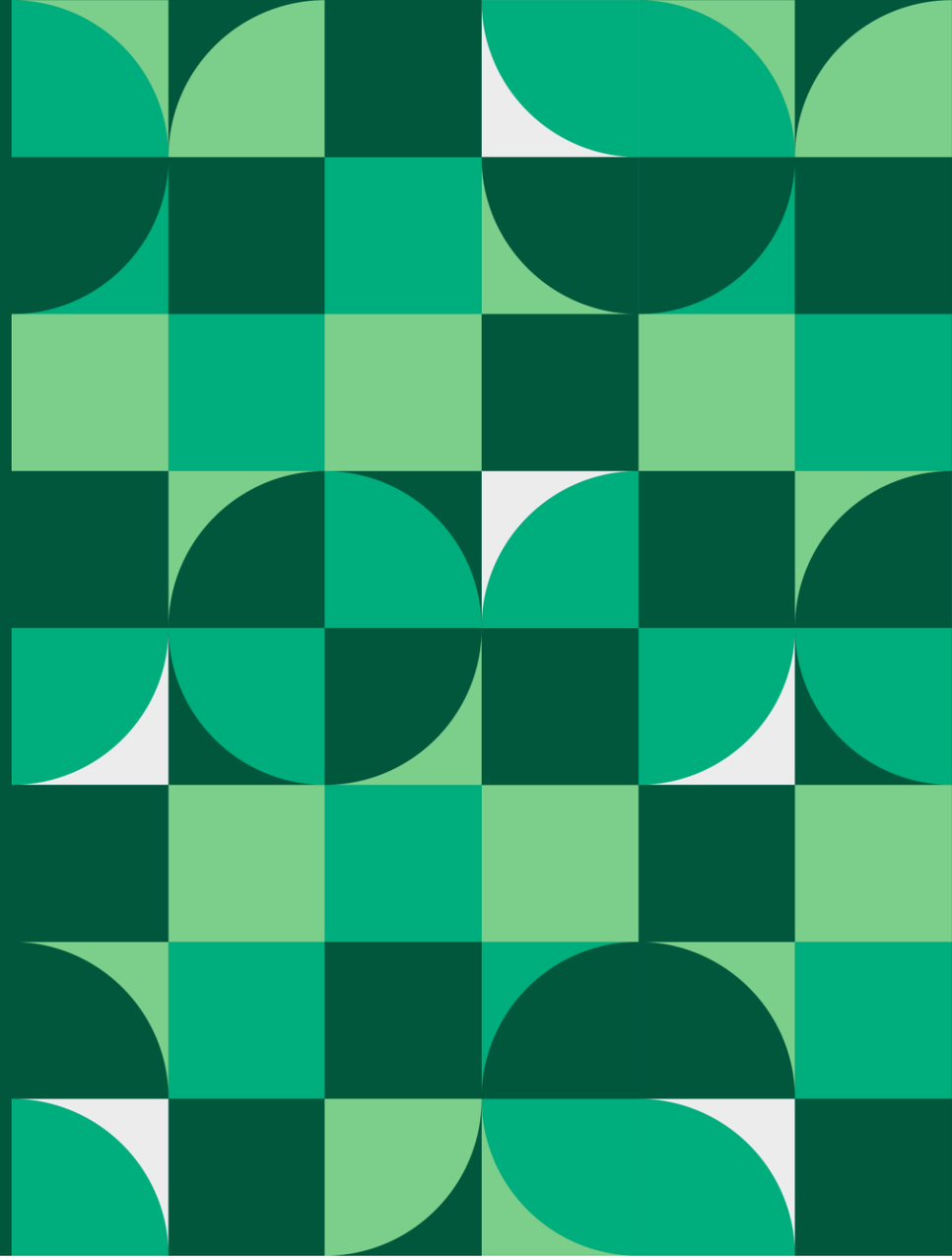
Streaming, Fast and Slow

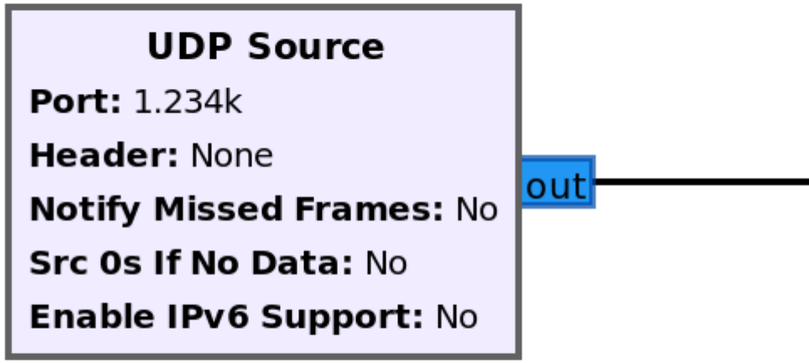
Martin Braun

Chief Engineer



This block sucks

- UDP source block
- Sub-par performance
- Fixed packet sizes required

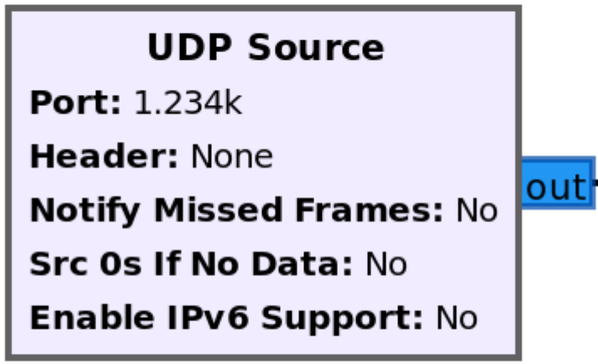


~~This block sucks~~
This block does some things well, others not (*cough* throughput)

- UDP source block
- Sub-par performance
- Fixed packet sizes required

- Handles headers/sequence numbers
- Handles partial receives
- **Highly portable!** (Operating Systems, IPv6, …)

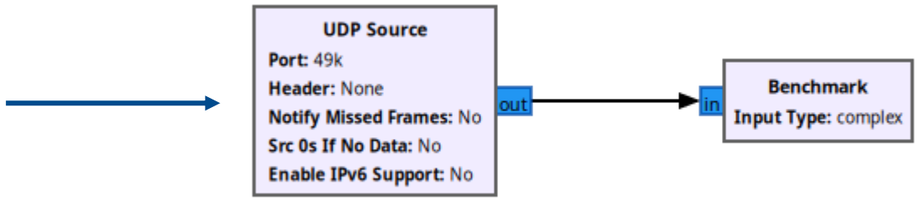
Please excuse while I only focus on the downsides!



What do you mean, this block has sub-par performance?

Simple experiment:

- Use USRP X310 as data source
- Instruct USRP to dump 400 Million samples into a UDP port (200 Msps, single-channel)
- Count the number of successfully received samples

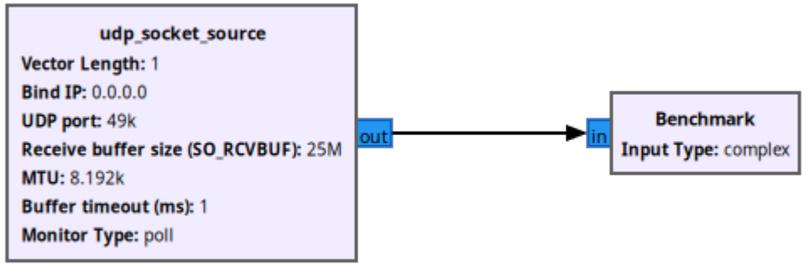


- Host computer: Intel i7-5930K, 3.5GHz, Intel 82599ES 2x10GbE NIC, 32 GiB RAM
- **Success rate: 4-8%**

I’d like to see you do better!

`github.com/mbr0wn/gr-netring`

- How about this:



- Success rate: Approx. 50%

Networking Fundamentals: Sockets

Back to Basics

EMERSON

Getting data from a raw UDP socket

- Even for a low-level C API, this is fairly simple
- POSIX compliant (this will work on Linux, Windows, Mac… you might need to specify different include files)
- Your socket is a file: Store a file descriptor (integer)

```
// Create a UDP socket
_sock_fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

- Steps required:
  - Create a UDP socket
  - Bind it to a specific port and IP address (can be 0.0.0.0)
  - Allocate memory to copy socket data into
  - Call `recv()` and watch it go!
  - `send()` goes the other way

```
// Bind the socket to the address and port
if (bind(_sock_fd, (struct sockaddr*)&addr, sizeof(addr)) < 0) {
    throw std::runtime_error(std::string("Failed to bind socket: ") +
                             strerror(errno));
}
```

```
ssize_t n = ::recv(_sock_fd, dst_buf /* void* */, buf_size, 0 /* flags */);
```

- Problems?
  - Yes! Loads of problems. Let's pick two:
  - Problem 1: `recv()` is blocking
  - Problem 2: System calls are expensive

Unblocking recv() [Part 1]

- Blocking behaviour: recv() will not return until there was a successful read on the socket (i.e., a UDP packet was successfully received by the NIC), or an error occurs (…or there is a corresponding signal).
- Not good! Our work() function has to return ASAP.
- First step: Simply unblock the socket
  - Either specify flags…

```
ssize_t n = 0;
while (n == 0) {
    n = ::recv(fg, buf, buf_size, MSG_DONTWAIT);
    if (n < 0) {
        if (errno == EAGAIN || errno == EWOULDBLOCK) {
            // that's fine
            continue;
        } else {
            // ...you have to handle this error
        }
    }
}
```

  - …or make it more permanent. This is good practice and is in fact required later.

```
/* Get the current flags */
int flags = fcntl(fd, F_GETFL);
/* Now add the O_NONBLOCK flag */
if (fcntl(fd, F_SETFL, flags | O_NONBLOCK) < 0) {
    // ...handle error
}
```

  - None of these are portable ☹

```
ssize_t n = 0;
while (n == 0) {
    n = ::recv(fg, buf, buf_size, 0 /* no flag required! */);
    if (n < 0) {
        if (errno == EAGAIN || errno == EWOULDBLOCK) {
            // that's fine
            continue;
```

  - So, you want me to spin on recv()?

Unblocking recv() [Part 2]

- What if we don't want to spin on recv()? We check the socket before reading! ("Are there any data available?")
- This can be done with a timeout: We let our application sleep until there are data; much more CPU efficient than spinning on recv().
- On Linux, there are three methods available:
  - select: POSIX-compliant, available on Windows. But very slow! O(n) complexity.
  - poll: Better. Also POSIX, but less portable.
  - epoll: Even better than poll! Supports edge trigger! O(1) complexity. Linux only.
- Mac OS / BSDs have `kselect()`

```
class NETRING_API select_monitor
{
public:
    select_monitor(int sock_fd, int buf_timeout_ms)
    {
        _sock_fd = sock_fd;
        _timeout.tv_sec = int(buf_timeout_ms / 1000);
        _timeout.tv_usec = int((buf_timeout_ms % 1000) * 1000);

        FD_ZERO(&_read_fds);
        FD_SET(_sock_fd, &_read_fds);
    }

    bool wait_for_data()
    {
#ifndef TEMP_FAILURE_RETRY
#define TEMP_FAILURE_RETRY(x) (x)
#endif
        // Wait for data on the socket
        const int ret = TEMP_FAILURE_RETRY(
            ::select(_sock_fd + 1, &_read_fds, nullptr, nullptr, &_timeout));
        if (ret < 0) {
            throw std::runtime_error("Failed to select socket: " +
                                     std::string(strerror(errno)));
        }
        return ret > 0; // Data is available
    }


private:
    //! Copy of the socket file descriptor we're monitoring. This class does not
    //  close the socket.
    int _sock_fd;
    //! Timeout value for select
    struct timeval _timeout;
    //! Set of file descriptors to monitor (basically, _sock_fd)
    fd_set _read_fds;
};
```

EMERSON

Unblocking recv() [Part 2]

- What if we don’t want to spin on recv()? We check the socket before reading! (“Are there any data available?”)
- This can be done with a timeout: We let our application sleep until there are data; much more CPU efficient than spinning on recv().
- On Linux, there are three methods available:
  - select: POSIX-compliant, available on Windows. But very slow! O(n) complexity.
  - poll: Better. Also POSIX, but less portable.
  - epoll: Even better than poll! Supports edge trigger! O(1) complexity. Linux only.
- Mac OS / BSDs have `kselect()`

```cpp
class NETRING_API poll_monitor
{
public:
    poll_monitor(int sock_fd, int buf_timeout_ms)
    {
        _pfd_read.fd = sock_fd;
        _pfd_read.events = POLLIN; // Monitor for input events
        _timeout = buf_timeout_ms;
    }

    bool wait_for_data()
    {
        const int ret = ::poll(&_pfd_read, 1, _timeout);
        if (ret < 0) {
            throw std::runtime_error(std::string("Failed to poll socket: ") +
                                     strerror(errno));
        }
        return ret > 0;
    }


private:
    // Variables for poll
    struct pollfd _pfd_read;
    int _timeout;
};
```

Unblocking recv() [Part 2]

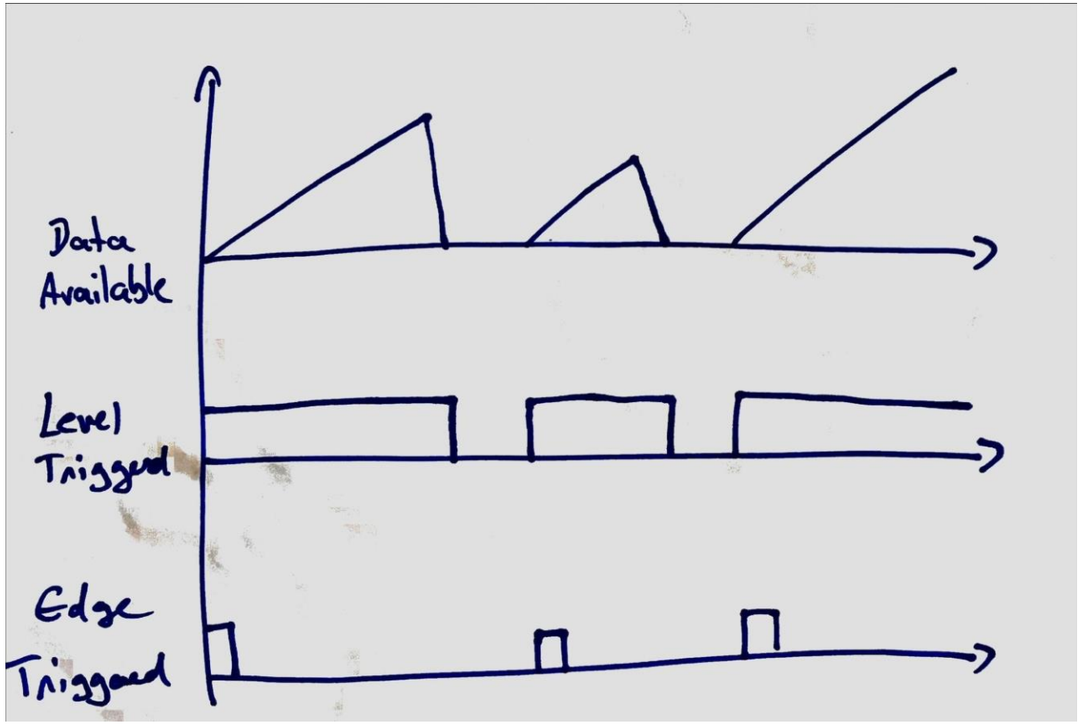
- What if we don't want to spin on recv()? We check the socket before reading! ("Are there any data available?")
- This can be done with a timeout: We let our application sleep until there are data; much more CPU efficient than spinning on recv().
- On Linux, there are three methods available:
  - select: POSIX-compliant, available on Windows. But very slow! O(n) complexity.
  - poll: Better. Also POSIX, but less portable.
  - epoll: Even better than poll! Supports edge trigger! O(1) complexity. Linux only.
- Mac OS / BSDs have `kselect()`

```cpp
template <bool edge_triggered = true>
class NETRING_API epoll_monitor
{
public:
    epoll_monitor(const int sock_fd, const int buf_timeout_ms)
        : _sock_fd(sock_fd), _timeout(buf_timeout_ms)
    {
        // Create an epoll instance
        const int epoll_flags = 0;
        _epoll_fd = epoll_create1(epoll_flags);
        if (_epoll_fd == -1) {
            throw std::runtime_error("Failed to create epoll instance: " +
                                     std::string(strerror(errno)));
        }

        // Add the socket file descriptor to the epoll instance
        _event.events = edge_triggered ? EPOLLET | EPOLLIN : EPOLLIN;
        _event.data.fd = sock_fd;
        if (epoll_ctl(_epoll_fd, EPOLL_CTL_ADD, sock_fd, &_event) == -1) {
            throw std::runtime_error("Failed to add socket to epoll: " +
                                     std::string(strerror(errno)));
        }
    }

    ~epoll_monitor()
    {
        // Remove the socket file descriptor from the epoll instance
        epoll_ctl(_epoll_fd, EPOLL_CTL_DEL, _sock_fd, nullptr);
        close(_epoll_fd);
    }

    bool wait_for_data()
    {
        // Wait for events on the socket
        int ret = epoll_wait(_epoll_fd, &_event, 1, _timeout);
        if (ret < 0) {
            throw std::runtime_error("Failed to wait for epoll event: " +
                                     std::string(strerror(errno)));
        }
        return ret > 0; // Data is available
    }


private:
    int _sock_fd;
    int _timeout;
    int _epoll_fd;
    epoll_event _event;
};
```

Edge-triggered vs. Level-triggered

- When does epoll report that data is available?
- Level-Triggered: Anytime there is any data available
- Edge-Triggered: Only when there was no data, and now there is new data
- Consequence: Always read all available data from socket when an edge is detected! (fewer `epoll_wait()` calls!)

```
while (true) {
    // Wait for edge
    if (!monitor.wait_for_data()) {
        continue; // No data available
    }

    while (true) {
        const ssize_t n = ::recv(fd, buf, buf_len, MSG_DONTWAIT);
        if (n < 0) {
            if (errno == EAGAIN || errno == EWOULDBLOCK) {
                break; // Wait for next edge
            }
            throw std::runtime_error("Failed to receive data: " +
                                     std::string(strerror(errno)));
        }

        // Process new data in buf
    }

    // Exit condition
}
```



EMERSON

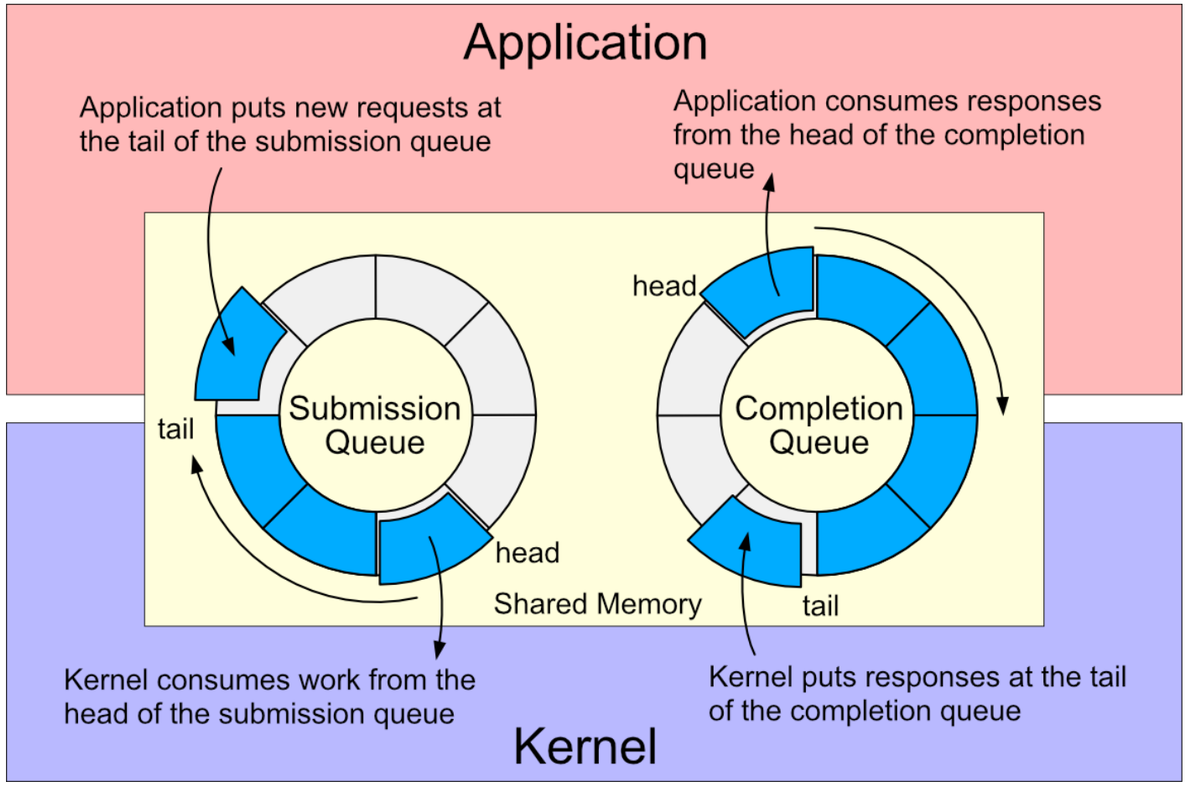
Networking Fundamentals: io_uring

OK, not quite so basic any more

EMERSON

Submission/Completion Queues

- With io_uring, system calls are not *synchronously executed*, but *asynchronously requested*, and the kernel can deal with them whenever it feels like it
- When the kernel has completed a request, it places the result in another queue, and the application can read it back when it is ready



[Source]

```
constexpr int RECV_FLAGS = 0;

// Synchronous
const ssize_t n = ::recv(fd, buf, buf_len, RECV_FLAGS);

// Asynchronous
struct io_uring ring;
int ret = io_uring_queue_init(num_entries, &ring, 0);
struct io_uring_sqe* sqe;
struct io_uring_cqe* cqe;
sqe = io_uring_get_sqe(&ring);
io_uring_prep_recv(sqe, fd, buf, buf_len, RECV_FLAGS);
ret = io_uring_submit(&ring);
ret = io_uring_wait_cqe(&ring, &cqe);
const ssize_t n = cqe->res;
```

EMERSON

Looks complicated? Make sync call non-blocking!

Still more code, but no polling or edge trigger handling!

This is obviously nonsense as asynchronous code, why not fill up the Submission Queue to the brim?

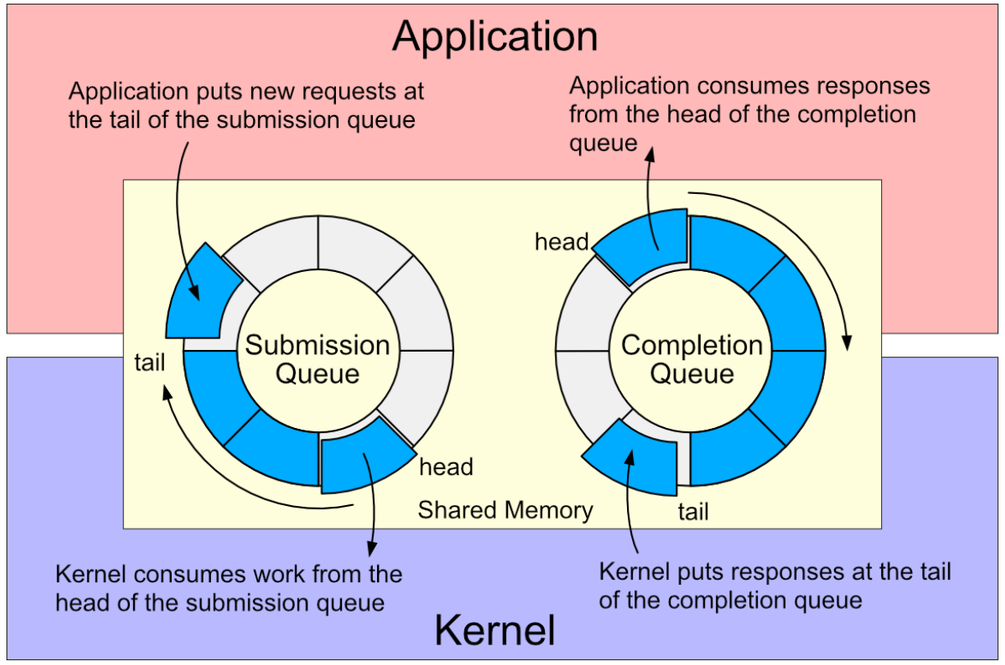
```
while (true) {
    // Wait for edge
    if (!monitor.wait_for_data()) {
        continue; // No data available
    }

    while (true) {
        const ssize_t n = ::recv(fd, buf, buf_len, MSG_DONTWAIT);
        if (n < 0) {
            if (errno == EAGAIN || errno == EWOULDBLOCK) {
                break; // Wait for next edge
            }
            throw std::runtime_error("Failed to receive data: " +
                                     std::string(strerror(errno)));
        }

        // Process new data in buf
    }

    // Exit condition
}
```

```
struct io_uring ring;
int ret = io_uring_queue_init(num_entries, &ring, 0);
struct io_uring_sqe* sqe;
struct io_uring_cqe* cqe;
struct __kernel_timespec to = { .tv_sec = 0, .tv_nsec = 1000 };
while (true) {
    // Submit one recv() call
    sqe = io_uring_get_sqe(&ring);
    io_uring_prep_recv(sqe, fd, buf, buf_len, RECV_FLAGS);
    ret = io_uring_submit(&ring);

    // Wait for completion, with timeout
    ret = io_uring_wait_cqe_timeout(&ring, &cqe, &to);
    if (ret == -ETIME) {
        // Handle exit condition (e.g., timeout)
        continue;
    }
    const ssize_t n = cqe->res;
    if (n < 0) {
        // Handle error
        break;
    }

    // Process new data in buf

    // Exit condition
}
```

EMERSON

Asynchronous programming looks different!

- We actually need to make use of the asynchronous nature of the API
- For this problem: Requesting a recv() is cheap, handling the received result is the expensive part
- => Keep submission queue full!
- => Let kernel handle async recv() operations as fast as it can!
- => If your consumer can’t handle the rate, process received UDP packets on different cores!



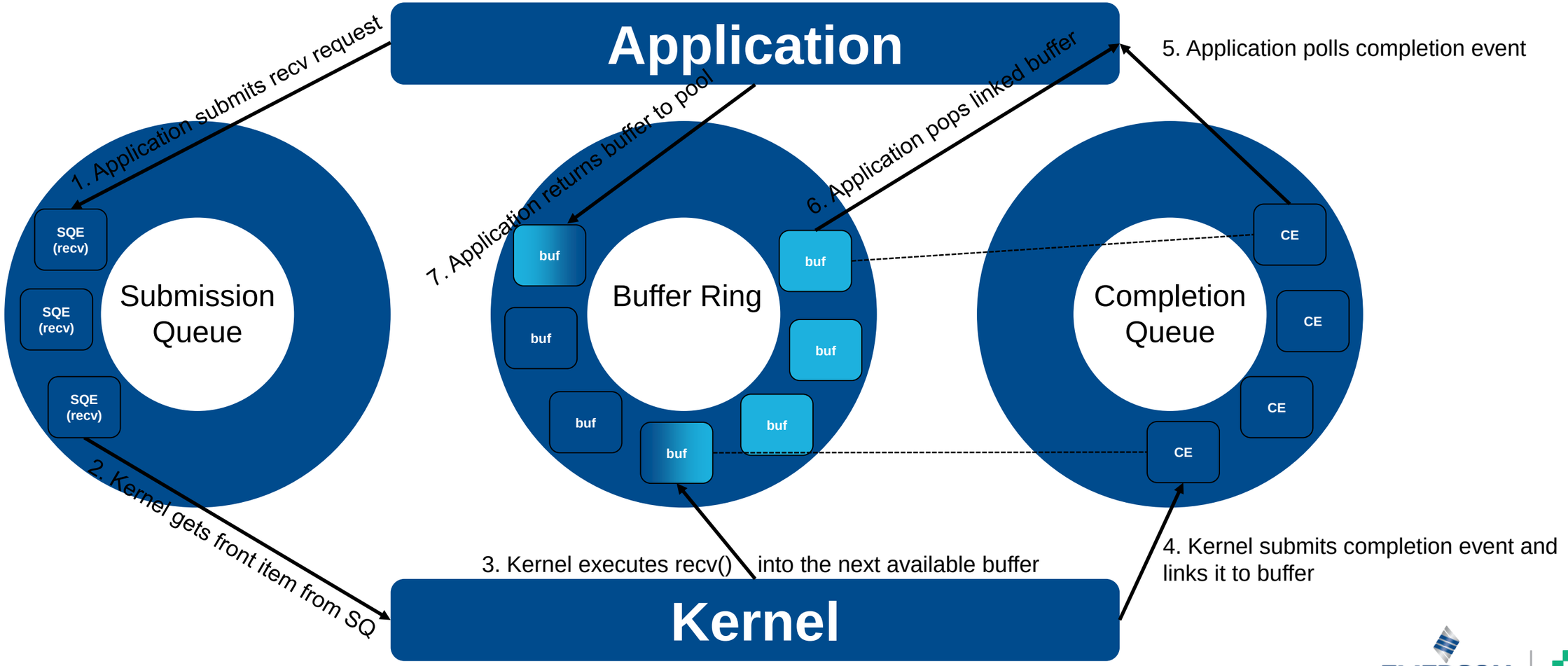
io_uring multishot operation

- Machine gun mode of io_uring: Submit a single submission queue entry, kernel will produce one completion queue entry for every successful receive
- Another ring is required: The buffer ring.
  - A buffer pool in which the recv() results can be copied
  - There can be multiple buffer rings
- No submission queue management!

```
while (true) {

    // In non-blocking mode, we need to wait for the CQE to be ready
    // before we can process it. We use a timeout to avoid blocking
    // indefinitely.
    ret = io_uring_wait_cqe_timeout(&_ring, &cqe, &to);
    if (ret == -ETIME) {
        continue; // No CQE ready yet, try again
    } else if (ret < 0) {
        throw std::runtime_error(std::string("Failed to wait for CQE: ") +
                                 strerror(-ret));
    }

    // There was a successful read, so we can process the CQE
    if (cqe->res < 0) {
        // If we got an error, we throw an exception
        throw std::runtime_error(std::string("Failed to receive data: ") +
                                 strerror(-cqe->res));
    }

    // If we got here, we received data. The res field contains the
    // number of bytes received.
    recv_len = static_cast<size_t>(cqe->res);
    // Copy the received data to the buffer
    const size_t buffer_id =
        static_cast<size_t>(cqe->flags >> IORING_CQE_BUFFER_SHIFT);

    // Your UDP payload is now in the buffer with index buffer_id!

    /* we're done with the buffer, add it back */
    io_uring_buf_ring_add(br,
                          b(buffer_id),
                          _buf_size,
                          buffer_id,
                          io_uring_buf_ring_mask(_n_bufs),
                          0);
    /* make it visible */
    io_uring_buf_ring_advance(_br, 1);
    /* CQE has been seen */
    io_uring_cqe_seen(&_ring, cqe);
}
```

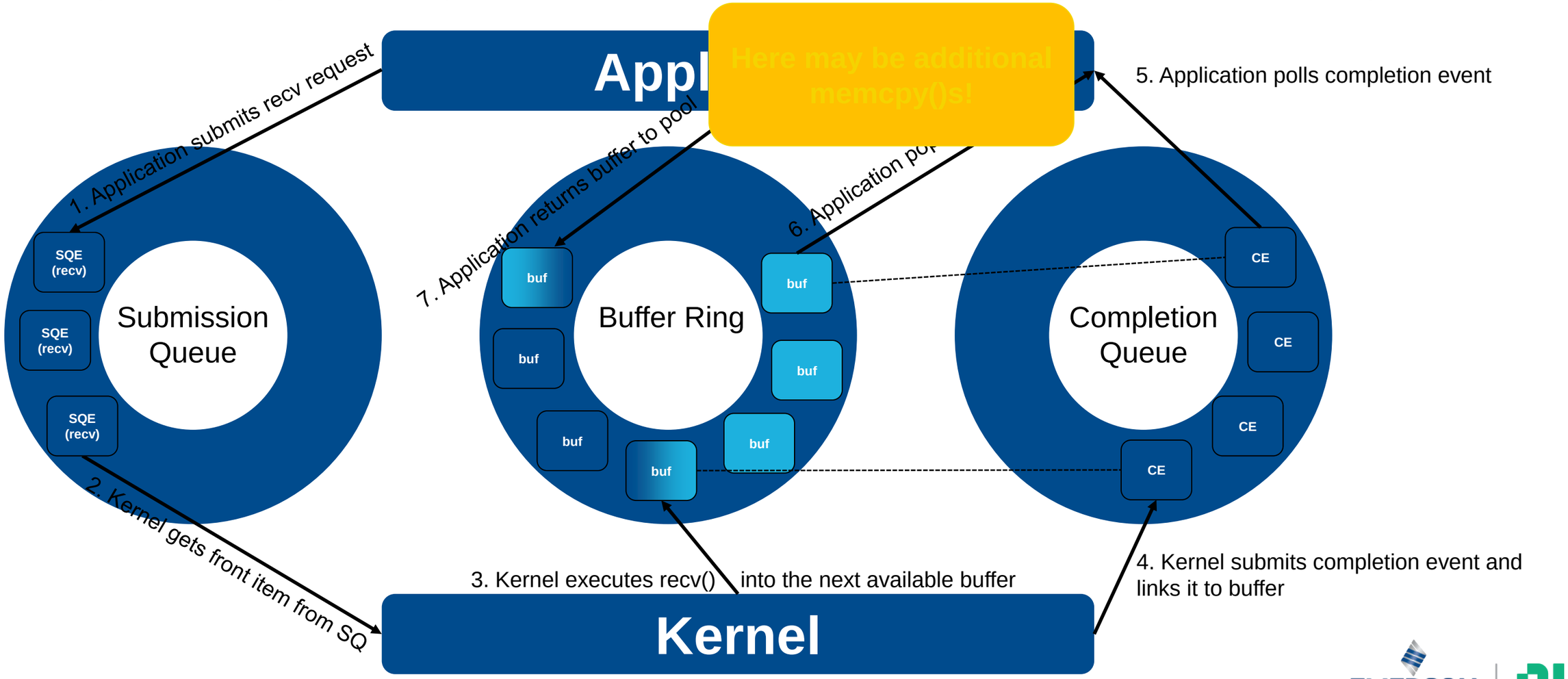
The most common side effect: Separate buffer rings/pools

- To enable (most) asynchronous UDP streaming applications, we need a special buffer pool
- This buffer pool needs to registered in a way that makes it accessible by the kernel. **This memory space is reserved for the async interaction!**



The most common side effect: Separate buffer rings/pools

- To enable (most) asynchronous UDP streaming applications, we need a special buffer pool
- This buffer pool needs to registered in a way that makes it accessible by the kernel. **This memory space is reserved for the async interaction!**



Benchmark 1: Capture 1.6 GB of data from active pipe

- Data source: USRP X310, freewheeling raw UDP streaming mode, 200 Msps == 800 Mbyte/s, capturing 2s worth of data

strace overhead!!

```
Allocating memory (1600000000 bytes)...
Running with receiver timeout (10s)...
Creating socket receiver on 0.0.0.0:49000...
Receiver is non-blocking.
Receive buffer size set to 2500000 bytes (requested 2500000 bytes).
Starting receiver of type non-blocking, monitor type: poll...
Received 1600000000 bytes in 9292ms.
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 94.63    1.992442           9    200402           recvfrom
  5.36    0.112773       56386         2           munmap
  0.00    0.000088          11         8           mprotect
  0.00    0.000055           7         7           write
  0.00    0.000028           3         8           close
  0.00    0.000021           0        32           mmap
  0.00    0.000018           6         3           brk
  0.00    0.000012          12         1           socket
  0.00    0.000007           0         8           fstat
  0.00    0.000006           6         1           getrandom
  0.00    0.000004           4         1           bind
  0.00    0.000004           4         1           setsockopt
  0.00    0.000004           2         2           fcntl
  0.00    0.000004           4         1           futex
  0.00    0.000004           4         1           set_robust_list
  0.00    0.000004           4         1           prlimit64
  0.00    0.000004           4         1           rseq
  0.00    0.000002           2         1           poll
  0.00    0.000002           2         1           getsockopt
  0.00    0.000000           0         6           read
  0.00    0.000000           0         2           pread64
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1           arch_prctl
  0.00    0.000000           0         1           set_tid_address
  0.00    0.000000           0        36        29 openat
  0.00    0.000000           0         6         5 newfstatat
------ ----------- ----------- --------- --------- ----------------
100.00    2.105482          10    200536        35 total
```

```
Allocating memory (1600000000 bytes)...
Running with receiver timeout (10s)...
Creating socket receiver on 0.0.0.0:49000...
Receiver is non-blocking.
Receive buffer size set to 2500000 bytes (requested 2500000 bytes).
Page size is: 4096 bytes.
Setting up buffer ring with 128 buffers of size 8192 bytes.
uring configured!
Truncating received data to remaining buffer size of 6400 bytes.
Successful peeks: 184613
Received 1600000000 bytes in 1999ms.
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 72.73    0.176920          11     15788       116 io_uring_enter
 26.57    0.064632       16157         4           munmap
  0.20    0.000484          13        36           mmap
  0.18    0.000430         430         1           execve
  0.15    0.000376          10        36        29 openat
  0.02    0.000058           5        11           write
  0.02    0.000049           8         6           read
  0.02    0.000040           5         8           fstat
  0.02    0.000039           4         8           close
  0.02    0.000037          37         1         1 access
  0.01    0.000035           4         8           mprotect
  0.01    0.000031           5         6         5 newfstatat
  0.01    0.000027          27         1           io_uring_setup
  0.01    0.000018           6         3           brk
  0.01    0.000016           8         2           io_uring_register
  0.00    0.000010          10         1           socket
  0.00    0.000008           4         2           pread64
  0.00    0.000006           6         1           getrandom
  0.00    0.000005           5         1           futex
  0.00    0.000004           4         1           bind
  0.00    0.000004           4         1           setsockopt
  0.00    0.000004           4         1           arch_prctl
  0.00    0.000004           4         1           set_robust_list
  0.00    0.000004           4         1           rseq
  0.00    0.000003           3         1           getsockopt
  0.00    0.000003           3         1           set_tid_address
  0.00    0.000000           0         1           prlimit64
------ ----------- ----------- --------- --------- ----------------
100.00    0.243247          15     15932       151 total
```

Benchmark 1: Capture 1.6 GB of data from active pipe

- Data source: USRP X310, freewheeling raw UDP streaming mode, 200 Msps == 800 Mbyte/s, capturing 2s worth of data

```
.../gr-netring/build  mbr0wn/wip ● /usr/bin/time ./examples/rx_to_mem --tech uring --recv-buff-size 2500000 -d 10 -m poll -b 800000000 --mtu 8000
Allocating memory (800000000 bytes)...
Running with receiver timeout (10s)...
Creating socket receiver on 0.0.0.0:49000...
Receiver is non-blocking.
Receive buffer size set to 2500000 bytes (requested 2500000 bytes).
Page size is: 4096 bytes.
Setting up buffer ring with 128 buffers of size 8192 bytes.
uring configured!
Truncating received data to remaining buffer size of 3200 bytes.
Successful peeks: 91221
Received 800000000 bytes in 999ms.
0.15user 0.48system 0:01.39elapsed 46%CPU (0avgtext+0avgdata 786540maxresident)k
0inputs+0outputs (0major+195747minor)pagefaults 0swaps
.../gr-netring/build  mbr0wn/wip ● /usr/bin/time ./examples/rx_to_mem --tech sockets --recv-buff-size 2500000 -d 10 -m poll -b 800000000 --mtu 8000
Allocating memory (800000000 bytes)...
Running with receiver timeout (10s)...
Creating socket receiver on 0.0.0.0:49000...
Receiver is non-blocking.
Receive buffer size set to 2500000 bytes (requested 2500000 bytes).
Starting receiver of type non-blocking, monitor type: poll...
Received 800000000 bytes in 1000ms.
0.17user 0.79system 0:01.44elapsed 66%CPU (0avgtext+0avgdata 785384maxresident)k
0inputs+0outputs (0major+195487minor)pagefaults 0swaps
```

CPU Overhead!

```
Allocating memory (1600000000 bytes)...
Running with receiver timeout (10s)...
Creating socket receiver on 0.0.0.0:49000...
Receiver is non-blocking.
Receive buffer size set to 2500000 bytes (requested 2500000 bytes).
```

0.00	0.000002	2	1		getsockopt
0.00	0.000000	0	6		read
0.00	0.000000	0	2		pread64
0.00	0.000000	0	1	1	access
0.00	0.000000	0	1		execve
0.00	0.000000	0	1		arch_prctl
0.00	0.000000	0	1		set_tid_address
0.00	0.000000	0	36	29	openat
0.00	0.000000	0	6	5	newfstatat
100.00	2.105482	10	200536	35	total

0.00	0.000005	5	1		futex
0.00	0.000004	4	1		bind
0.00	0.000004	4	1		setsockopt
0.00	0.000004	4	1		arch_prctl
0.00	0.000004	4	1		set_robust_list
0.00	0.000004	4	1		rseq
0.00	0.000003	3	1		getsockopt
0.00	0.000003	3	1		set_tid_address
0.00	0.000000	0	1		prlimit64
100.00	0.243247	15	15932	151	total

Benchmark 2: Capture nothing (there is no data)

This is an idle test:

- epoll is a slight improvement over poll
- io_uring basically same ballpark (this is expected: We need to rely on the kernel to tell us when there is IO to be had)

```
Allocating memory (800000000 bytes)...
Running with receiver timeout (2s)...
Creating socket receiver on 0.0.0.0:49000...
Receiver is non-blocking.
Receive buffer size set to 2500000 bytes (requested 2500000 bytes).
Page size is: 4096 bytes.
Setting up buffer ring with 128 buffers of size 8192 bytes.
uring configured!
Successful peeks: 0
Received 0 bytes in 2003ms.
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 97.68    0.082411       20602         4           munmap
  0.84    0.000707          33        21        20 io_uring_enter
  0.57    0.000483         483         1           execve
  0.27    0.000225           6        36           mmap
  0.21    0.000173         173         1         1 access
  0.11    0.000097           2        36        29 openat
  0.09    0.000078           7        10           write
```

```
Allocating memory (800000000 bytes)...
Running with receiver timeout (2s)...
Creating socket receiver on 0.0.0.0:49000...
Receiver is non-blocking.
Receive buffer size set to 2500000 bytes (requested 2500000 bytes).
Starting receiver of type non-blocking, monitor type: epoll...
Received 0 bytes in 2006ms.
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 97.74    0.085911       42955         2           munmap
  0.63    0.000557          27        20           epoll_wait
  0.55    0.000484         484         1           execve
  0.40    0.000352          11        32           mmap
  0.27    0.000240           6        36        29 openat
```

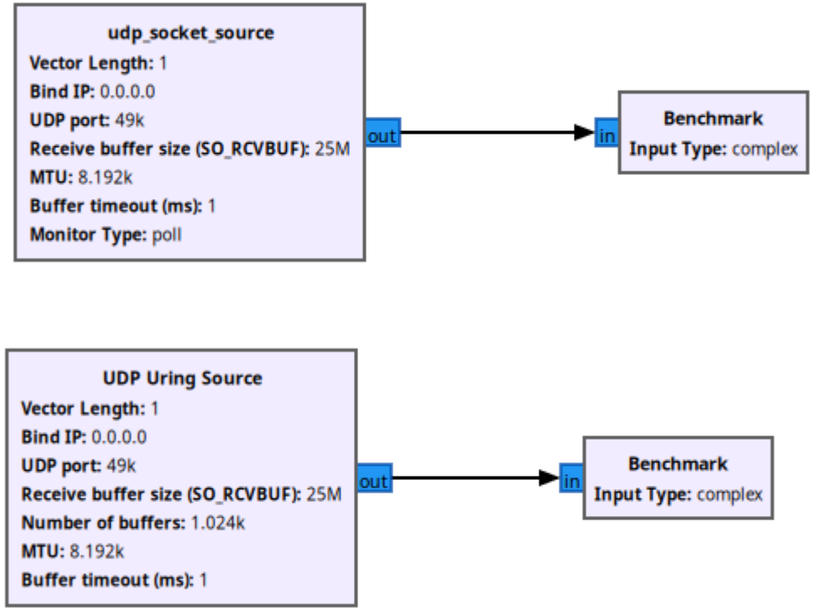
```
Allocating memory (800000000 bytes)...
Running with receiver timeout (2s)...
Creating socket receiver on 0.0.0.0:49000...
Receiver is non-blocking.
Receive buffer size set to 2500000 bytes (requested 2500000 bytes).
Starting receiver of type non-blocking, monitor type: poll...
Received 0 bytes in 2005ms.
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 99.05    0.083780       41890         2           munmap
  0.81    0.000683          34        20           poll
  0.06    0.000052           7         7           write
  0.03    0.000027           3         8           close
  0.02    0.000016          16         1           socket
  0.01    0.000007           7         1           setsockopt
```

Putting it into action

Well, does it work?

New blocks: All experimental

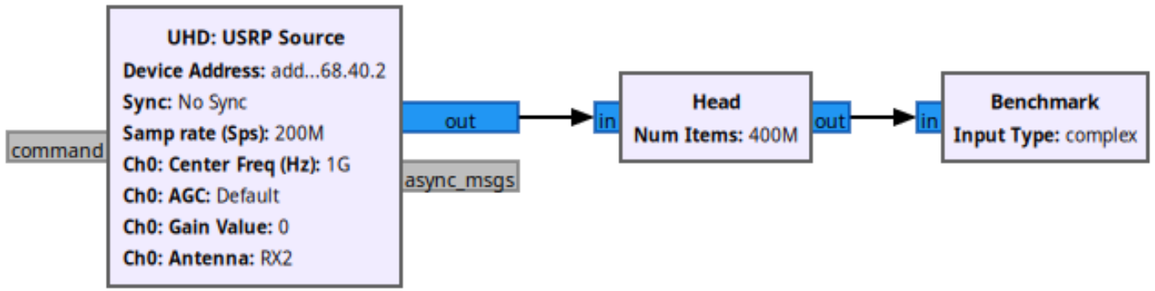
- Socket source:
  - Maybe helpful
  - Ideally, merge with portability from in-tree (Boost-y) UDP Source
- Uring source:
  - Not yet proven better (actually, seems to be worse)
  - io_uring could be great for file sink or UDP sink (reduce syscalls)
- Currently not implemented / tested:
  - DPDK blocks: This would have to operate in tandem with core pinning
  - AF_XDP / eBPF: Honestly, I don’t know
  - RDMA-UC: RDMA is not strictly UDP, but this could be a useful tool to offload data movement



Existing Code: Good ol’ USRP source

- USRP source: Works fine!
- UHD uses (some) Boost for portability, and poll on Linux for non-blockingness
- On top of UDP streaming, this does type conversion and flow control! And still, we lose less data.

- In a scenario of medium-range rates and erratic thread activity, flow control has its benefits



Quo Vadis, o streamer?

- Is it worth pursuing other options than directly talking to sockets (with Boost, with direct system calls, or however)?
- Can io_uring do anything for us?
  - Multishot operation is probably not the right solution for a UDP source, so how well would an implementation with an actively managed submission queue work?
- Which other technologies (DPDK, RDMA) are most promising for GNU Radio network streaming?
- Which metrics are most appropriate for evaluating the performance of streaming code? (Number of received packets, CPU load under various streaming circumstances, …)?
- How good can a non-flow controlled UDP streamer actually be in combination with GNU Radio’s scheduler?