Extending Click to Support Block Requests

ANDREW BIRCK
Brown University

---

1. INTRODUCTION

Click[1] is a software architecture for easily creating configurable routers. Routers are built from simple processing modules called elements. Each element implements a simple function like classification, queuing or scheduling. A configuration then specifies how to arrange the elements in a connected graph where the connections represent the flow of the packets through the elements. Configurations are easy to create and can be extended by adding existing elements or creating new ones. This allows users to quickly and easily build complicated routers for testing or production use.

Click was built because most routers have closed, static designs. Click easily lets you create a flexible, extensible router. Similarly hardware or software dealing with block devices are inflexible and static. Creating a new disk scheduling algorithm involves kernel hacking. There's no easy way to configure or extend an existing RAID scheme. There's little the user can do to specify how the block device system should operate based on their needs. Extending Click to handle block requests and block devices is meant to solve these problems.

I've modified Click to be able to handle block devices requests and have written 16 elements to create configurations with. I've built an example configuration that implements a one-time pad type device.

The rest of the paper describes how the Linux kernel and Click were modified to allow Click to process block requests (Section 2), an example application of a one-time pad like configuration (Section 3), a description of the block elements and how they work (Section 4), and future work (Section 5).

2. ARCHITECTURE

Click was extended to handle block requests rather than building a different architecture to handle block requests because it potentially allows block requests to interact with packets (a network file server could easily be a Click configuration) and it gives us the features already built into Click. Features like push and pull connections, flow-based router context, the Click language, and hot swapping configurations all work with the new block elements.

The modifications to Click to allow it to handle block requests currently require Click to be compiled as a kernel module on Linux. To allow Click to handle block requests (which I'll also refer to as bios because Linux represents a block request with a bio structure) changes were made to both the Linux kernel and Click itself.

2.1 Hooking Into Linux

The Linux kernel used while making modifications to Click was the 2.6.13.4 vanilla Linux kernel. The first step in getting block requests to Click was to decide what mechanism Click should use to interact with Linux. Click uses the notifier_block structure in the kernel to intercept packets and we chose to use notifier blocks to do the same with block requests. To inject packets back into Linux at the same point they were intercepted Click adds an extra argument to the function from which the packet was intercepted. When injecting a packet Click passes a special value to the function so it won't try to intercept the packet again. The same method was used to block requests.

Next it was necessary to identify appropriate places for Click to hook into the kernel. We needed to find places we could be sure all bios going to and from block devices could be intercepted, manipulated by click and then possibly sent back to the block device or to Linux.

To intercept block requests on the way towards the block device we hooked into the generic_make_request function. Whether requests are generated by user requests or by the kernel itself all block requests pass through generic_make_request. In addition the function is the last place in the kernel you're guaranteed to see a request before a block device handles it. In most cases after passing through generic_make_request a bio will enter a request queue but some drivers that don't benefit from ordering of requests (like a memory backed device) won't use a request queue and generic_make_request is the last place to catch the request for those types of devices. Because of the way stacking drivers work we also had to be careful about where we intercept packets in generic_make_request. The notifier_block is set up such that it will intercept the bio every time it is about to be resubmitted to a different block device by a stacking driver.

On the return path all bios pass through the bio_endio function. There are a couple complications with this function also. The bio_endio function maybe be called on a partially completed request as well as a fully completed request. Bios that experience errors while the block device is servicing them are returned to Linux through bio_endio as well. Since we cannot assume all bios are successfully completed bios we need to send more information than just the bio into Click. Therefore a struct containing the number of bytes just completed by the driver as well as the error code as passed in as well as the bio. Any BlockElement that first receives the bio on the return path then takes these values and puts them in annotations for use by other BlockElements as the request is passed along.

When intercepting packets in bio_endio we also have to realize we're most likely in an interrupt request so we can't immediately start processing packets. Instead we queue the requests and wait for the Click kernel thread to start at which point we forward the packets along their path. This isn't a problem in the generic_make_request function because we are never in an interrupt request when inside that function.

2.2 Supporting Block Requests in Click

Click handles Linux's skb structures (the packet equivalent of a bio) by casting it into a class called Packet. A Packet is a class with no virtual methods and no member variables so the cast is just an easy way to associate a set of functions with the skb. Elements then pass Packets to each other and when it comes time to send the Packet back to Linux it's

cast to a skb again.

This trick works well but also has its limitations in that we can't use polymorphism to generalize any similarities between Packets and our block request class because we can't have a virtual method table. Instead we've simply adopted the same trick of casting a bio to a class with no virtual methods or member variables that's called a BioRequest. The BioRequest then has methods to access data from the bio, set annotations, create a clone and more. When passing the BioRequest to the next element it is temporarily cast to a Packet so that it can be sent along Ports which only work with Packets. It is then cast back to a BioRequest at the next element. The casting is handled by the BlockElement superclass so this is transparent to the elements themselves.

The downside of using the existing Ports class to create connections and move BioRequests between elements is that this provides no type checking. An element that uses BioRequests will happily cast any real Packets it receives to BioRequests and an element that uses Packets will accept BioRequests since they're passed as Packets. Implementing a scheme to enforce type checking should be done and is future work.

3. ONE TIME PAD EXAMPLE

This section presents a Click configuration that shows how the block elements can be combined to do something interesting. One-time pad encryption is the only theoretically unbreakable encryption method so lets suppose two parties wanted to exchange data by using a disk in a one-time pad type manner.

The sender and receiver both have a block device filled with the same bits and this block device is at least as long as the data being exchanged. We'll call this disk /dev/hda1. They then wish to exchange data by sending an encrypted floppy in the mail. We'll call the block device that should contain the encrypted information /dev/fd0. The unencrypted message is the data on /dev/fd0 xored with the data on /dev/hda1. The figure 1 shows a Click configuration that allows both parties to transparently read and write the encrypted message. All ports are push ports in this configuration so the marks for push/pull ports have been omitted.

The configuration creates a fake device called /dev/clickfake0 that is used to create and read the encrypted message. When Linux wants to write to the clickfake block device Click intercepts the packet. Once in Click the configuration filters the request based on whether it is a read or a write since they have to be handled differently. Full descriptions of the elements are contained in section 4 but I'll describe what's happening on both paths to illustrate how the configuration works.

The read path starts by forking the request into two different requests because we need to perform two reads. The BlockFork element creates copies of the BioRequest fed to it as input. Each copied request will be the same in terms of read/write status, block device and sectors requested but the BlockFork element will provide each BioRequest copy with a new memory page to back the request. This ensures that if the memory backing one BioRequest changes (perhaps due to a read) the data pointed to by the other requests will not change. In this case it lets us perform two reads without either read interfering with the other. Once we've set the correct device on each request the BlockDev elements will complete a bio by sending it to the block device and intercept it on its way back to Linux. The BlockShadow element is necessary because of the way

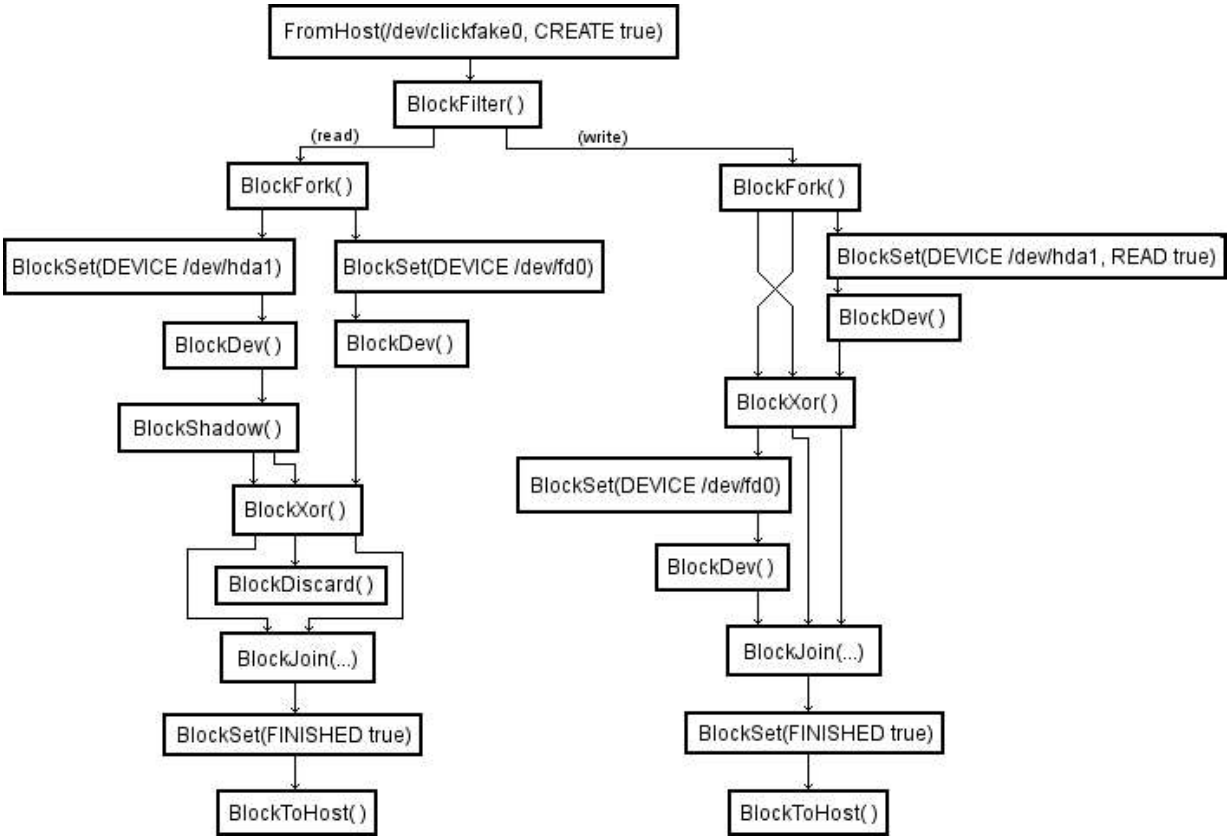

Figure 1

BlockXor works; BlockXor takes a source and two destination BioRequests. Because in the read case we want the source to be the same thing as one of the destinations we create a shadow copy so the BioRequest can be given to both input ports. This works because unlike BlockFork the BlockShadow copy points to the same memory. Once the data has passed through the BlockXor we've got what we want but there's a little clean up to do before we can hand the data back to Linux. BlockDiscard gets rid of the shadow copy we created and the BlockJoin waits for all BioRequest copies and the original BioRequest creating them to arrive. Once that has happened it outputs the original BioRequest which Linux is waiting on. The last BlockSet sets some annotations so when BlockToHost gives the bio back to Linux it will report success.

The write path makes use of many of the same elements but is one subtly involved with a write. When we're asked to write unencrypted data in memory to the disk we need to encrypt it first but we can't encrypt the memory in place because Linux is potentially still using that memory. To Linux it will appear the memory is corrupted. To solve this problem we have the BlockXor write its result into a BioRequest with a different memory page backing it. The outputs of BlockFork are crossed before entering BlockXor because the first copy leaves BlockFork on output 1 but needs to be input 0 on BlockXor to be the destination BioRequest. Once the xor has taken place the encrypted result is sent to be written to disk and the original BioRequest is sent back to Linux as a completed request as soon as it completes.

4. ELEMENT MANUAL

Format of the element manual:

- **ElementName(ARGUMENTS). PROCESSING PORTS.** Description
  - **ElementName**: Name of the element.
  - **ARGUMENTS**: Lists the order of the arguments to be supplied to the element. Types and a description of the arguments are included in the element description. Optional arguments are included in brackets. If KEYWORDS is listed then the element takes certain arguments in the form of "KeywordName Value". Valid keywords are also listed in the element description
  - **PROCESSING:** Either PUSH, PULL or AGNOSTIC like all Click elements.
  - **PORTS:** Format is: NumberInputs / NumberOutputs. A hyphen following a number means that number or more (e.g. "1/2-" means one input and two or more outputs).
  - Description: Plain text description of the element including arguments.

Alphabetical list of elements:

- **BlockDev( ). PUSH 1/1.** Takes a BioRequest, annotates it so that it may be found later and then sends it to the device specified by the bio. After the device has completed the request the element intercepts the bio on its way back and pushes it to the next element.
- **BlockDiscard( ). PUSH 1/0.** Discards any BioRequests sent to it.
- **BlockFilter( ). PUSH 1/2.** Currently only a read/write filter. Reads are sent on output 0 and writes sent on output 1. The idea is to expand this into a much more general element by allowing the user to specify the criteria to filter on.
- **BlockFork([ARMS]). PUSH 1/2-.** Will fork a BioRequest into several BioRequests. The original bio is output on port 0. The new BioRequests are identical to the original but have their own, newly allocated pages backing them and annotations marking them as copied and containing allocated pages. The pages backing the BioRequest contain the data to read or write. This allows you to fork a read BioRequest and safely send each of them to different devices as they will not try to read into to the same address in memory.

  BioRequests generated by a BlockFork element must all, at some point, be sent as inputs to a BlockJoin element that is associated with said BlockFork. It is safe to fork an already forked BioRequest so long as everything is properly nested (i.e. the inner fork in joined before the outer fork is joined).

  Optional argument ARMS is an unsigned integer that specifies the number of outputs the BlockFork should have. If the actual number of outputs differs from the specified number an error is thrown an configuration time.
- **BlockFromDev(DEVICE). PUSH 0/1.** Will examine completed (successfully or not) bios returning from block devices. If the device listed in the bio matches the device specified by BlockFromDev click will take the bio process it. Once the bio enters click it is no longer being processed by Linux. If the bio needs to be processes by Linux it can sent to a BlockToHost element.

DEVICE argument is a string specifying the block device to use. Note that this is the actual device and not a partition or RAID device. Therefore you must specify /dev/hda instead of /dev/hda1 for example.

- **BlockFromHost(DEVICE [, KEYWORDS]). PUSH 0/1.** Will examine bios that are on their way to a block device. If the block device in the bio matches the block device specified then Click will intercept the packet and stop Linux from processing it. When done with the bio it can be returned to Linux via BlockToDev or BlockToHost.

  DEVICE is a string specifying the block device whose bios we would like to intercept. This is done before partition remapping so here DEVICE should include partition numbers if appropriate (i.e. /dev/hda1). In many cases DEVICE will be a non-existent device which will be created by BlockFromHost for you if used with the CREATE keyword.

  Keyword arguments are:

  CREATE Boolean specifying whether DEVICE should be a fake device created by Click. Currently the device created has a hard coded name independent of the DEVICE argument.

- **BlockJoin(BLOCKFORK). PUSH 2-/1.** Joins BioRequests that were previously forked. When a BioRequest has been split into multiple requests by BlockFork the BlockJoin will wait for all of those requests before emitting output based on those requests. Once all BioRequests are present BlockJoin will output the original request given to BlockFork and discard the others.

  BLOCKFORK is the name of the BlockFork element whose BioRequests we are now joining.

- **BlockPrint(LABEL). AGNOSTIC 1/1.** Prints out information about the BioRequest itself. Does not print out the data being processed by the request. The information printed: label (below), address of the bio request in memory, read/write status, number of bio_vec in the bio, current vector index, first sector of the request, number of sectors left, number of current sectors, annotations related to return status and the reference count of the bio.

  LABEL is a string that precedes the BioRequest information.

- **BlockPrintShort(LABEL). AGNOSTIC 1/1.** Similar to BlockPrint but only uses one line to output the label, read/write status and first sector of each bio.

  LABEL is a string that precedes the BioRequest information.

- **BlockRaid0Concat(DEVICE [, DEVICE2, DEVICE3, etc...]). PUSH 1/1.** Creates a block device that simply concats all the block devices provided as arguments together to provide a large virtual block device.

  The DEVICE arguments are strings representing the block devices to use in the RAID configuration.

- **BlockSet([KEYWORDS]). AGNOSTIC 1/1.** Allows user to set various properties of the BioRequests. Currently allows for setting of read/write status, setting of the bio's block device and annotations regarding the completion status of the request.

  Keyword arguments:

DEVICE String representing the block device to which the bio will be sent

FINISHED Boolean that if true sets annotations in the BioRequest so that it can be sent to BlockToHost and Linux will recognize the bio as a completed request.

READ Boolean that if true will set the bio to be a read request.

WRITE Boolean that if true will set the bio to be a write request.

Note that "WRITE false" is not the same as "READ true". Setting any of the keywords to false will have a null effect.

- **BlockShadow( ). PUSH 1/2-.** Makes an exact copy of the BioRequest. Data pointed to is still the exact same as the original BioRequest as this function does not allocate new pages to back the copies. BioRequest created can simply be discarded when done with; there is no join equivalent for shadow copies.
- **BlockToDev( ). PUSH 1/0.** Accepts a BioRequest and sends the bio on its way toward the block device to be serviced. BlockToDev will not intercept the request on its way back like BlockDev.
- **BlockToHost( ). PUSH 1/0.** Sends a bio to Linux as if a device had just finished servicing the request. If the bio has never passed through a block device which would set it as completed it may be necessary to pass the request through BlockSet to mark it as completed before passing to BlockToHost.
- **BlockToPacket( ). AGNOSTIC 1/1.** Takes a BioRequest and outputs a UDP packet. The format of the UDP packet is a long followed by an unsigned int. The long is the first sector requested by the bio and the unsigned int is the number of sectors request. Meant as a proof of concept element.
- **BlockXor( ). PUSH 3/3.** Xors the contents of two BioRequests and stores the result in the third. The BioRequests must all have the same fork annotation to determine which requests belong together. That means the destination and source BioRequests must have all been produced by the same BlockFork or be BlockShadow copies of BioRequests generated by the BlockFork.

  Input 0 is the destination BioRequest and inputs 1 and 2 are the source BioRequests. All three BioRequests are then passed along on their respective outputs. The inputs are unchanged after passing through BlockXor.

5. OPEN ISSUES

- There is nothing in Click to prevent an element from sending a BioRequest to an element that expects a Packet or visa versa. A type checking system should be added to Click to minimize the possibility of creating bad configurations.
- BlockFromHost element has the ability to create a fake block device. BlockFromHost also needs to set a capacity for that block device but has no way of knowing what the capacity should be. Currently it's hard coded by the programmer on a per-configuration basis. Instead BlockFromHost should be able to query elements further down configuration to determine the correct capacity.
- Click should be able to implement various queuing policies for the block devices it uses however Linux maintains its own request queue for each device. To make its own

queuing decisions Click needs to either incorporate Linux's queue into its configuration or bypass it entirely.

- If requests were annotated with information from high levels (like the file system layer) it could allow for more intelligent and interesting decision making at lower levels. Things like file based priority queuing become possible.

REFERENCES

[1] E. Kohler et al. The Click Modular Router. ACM Transactions on Computer Systems. 18(3), pg 263-297, August 2000.