EE533 Final Project: NetFPGA Hardware Accelerated Publish/Subscribe Broker

Team GotYourTrifecta: Timothy Ferrell, Sebin Kuriakose, Leo Linsky, Neeraj Nath, Nita Simon, Yao Xiao

SECTION 1: ABSTRACT

Our final project transforms the NetFPGA from a simple IP router to an efficient event broker for a publish/subscribe network paradigm. We implement custom efficient multicast protocols to allow for flexible, accelerated delivery to topic subscribers. Our system will have several novel and innovative features to distinguish our design from existing event broker implementations. First of all, we will implement custom, accelerated event matching hardware. This allows us to efficiently match any given event with the correct subscriber network. Second, we will leverage and modify the existing NetFPGA input and output queues to support more efficient multicast using multiple threads and cores. Third, this publish/subscribe network will be completely encrypted in a content-based manner. This allows publishers and subscribers to treat each topic as a type of Virtual Private Network, which is highly desirable for effective and confidential information dissemination.

SECTION 2: INTRODUCTION

Publish/subscribe middleware has become a big business, as traditional IP routing does not provide an efficient multicast system. As network applications diversify, engineers have found the customary client/server paradigm increasingly limiting. Many group messaging applications, such as Facebook messenger, have moved to a publish/subscribe architecture. In any application where mass content promulgation is advantageous, the ability to ‘push’ out notifications to a subscriber network in parallel is extremely useful. In order to accomplish this, traditional and dumb IP routers must be replaced by an intelligent network event broker with efficient multicast capabilities. Additionally, information protection and confidentiality has become more important than ever. Protecting and securing communications is of increasing concern. Both companies and individuals covet the ability to encrypt and distribute information according to content. Accordingly, we see a massive opportunity in implementing a hardware-accelerated publish/subscribe event broker for private, encrypted content-based topics. Our broker provides hardware accelerated topic matching, which allows our broker to process many more multicast packets than a more traditional software solution. Additionally, we provide a simple programming interface so the broker can be extended with extra software for a given application. Low power sensor networks could utilize the broker to do some basic content processing before forwarding messages to the subscriber networks, for example. We have created a system which greatly outperforms existing publish/subscribe systems.

SECTION 3: BACKGROUND AND RELATED WORKS

A content-based pub/sub system(Guruduth Banavar[2]) has been proposed in the literature. Due to lack of native support for multicast communication for efficient pub/sub system, this paper introduces and evaluates a novel and efficient distributed algorithm for this purpose called “link matching”. Link matching performs enough computation at each node to determine the subset of links to which an event should be forwarded. This paper also show simulations that

link matching yields higher throughput than flooding and overall CPU utilization of link matching is comparable to that of centralized matching.
An efficient event delivery(Fengyun Cao[5]) has been introduced in this literature. Existing event routing solutions such as IP multicast or application-level multicast techniques are not practical due to high volumes of communication patterns in pub/sub systems. This paper first analyzes two existing approaches: *filter-based* method, which performs content-based filtering on brokers to guide routing decisions, and *multicast-based* method, which delivers events through a few high-quality multicast groups. Then this paper presents a new routing scheme called Kyra that balance trade-offs of two methods listed above. Kyra combines the advantages of content-based filtering and event-space partitioning in the existing approaches to achieve better routing efficiency.

Content-based router using hardware accelerator(Young H.Cho[7]) has been proposed in this literature. Traditionally, we always use software to do pattern matching but with the processing speed increasing to 1Gbps or even more, we cannot afford to search a given pattern in software. This paper introduces techniques for parsing data streams using hardware to achieve a maximum throughput of 12.90 Gbps.

The paper(Baldoni[1]) proposes an architecture for implementing content-based pub/sub that harnesses and leverages the scalable message routing and adaptive self-configuration of overlay networks, i.e. the properties so much sought by content-based pub/sub schemes. As the paper suggests, the proposed architecture is the first content-based pub/sub implementation not requiring any manual configuration and management apart from the setup of an overlay network itself. The paper introduces a novel primitive for one-to-many message delivery, showing through simulation how this can improve performance of the architecture.

SECTION 4: GYT SYSTEM DESIGN

A. Network Processor Hardware Design

General Processor Design

We designed a basic 5 stage pipeline processor as the basic building block for our broker. All code can be found in gyn_processor.v and gyn_datapath.v. All source code is included as an attachment to this report. An essential part of this design is the network state machine. This determines access to the data memory, and if the instruction counter is advancing or not. This also allows dynamic The relevant code is in gyn_datapath.v, around line 350:

```
        case(state)
                    START: begin
                           thread_can_inc = 1'b0;
                           if ((in_ctrl != 1'b0) && input_grant) begin // If control bits
are not 0, then we have new coming packet.
                                  state_next = HEADER;
                                  begin_pkt_next = 1'b1;
                                  end_of_pkt_next = 1'b0;
                           end
```

```
                        else if (!modeTopLevel[2]) begin
                                state_next = REGISTER;
                                fifo_ready_next = 1'b0;
                                begin_pkt_next = 1'b0;
                                end_of_pkt_next = 1'b1;
                        end
                end
                HEADER: begin
                        begin_pkt_next = 1'b0;
                        if (in_ctrl == 1'b0) begin
                                 header_counter_next = header_counter + 1'b1;
                                if (header_counter_next == 2'b11) begin
                                        state_next = PAYLOAD;
                                end
                        end
                        if (!modeTopLevel[2]) begin
                                state_next = REGISTER;
                                state_next = REGISTER;
                                header_counter_next = 1'b0;
                                end_of_pkt_next = 1'b1;
                                fifo_ready_next = 1'b0;
                        end
                end
                PAYLOAD: begin
                        if (in_ctrl != 1'b0) begin
                                state_next = PROCESS;
                                header_counter_next = 1'b0;
                                end_of_pkt_next = 1'b1;
                                fifo_ready_next = 1'b0;
                        end
                        if(!modeTopLevel[2]) begin
                                state_next = REGISTER;
                                header_counter_next = 1'b0;
                                end_of_pkt_next = 1'b1;
                                fifo_ready_next = 1'b0;
                        end
                end
                PROCESS: begin
                        processor_control_next = 1'b1;
                        if(modeTopLevel[2]) begin //Can only leave state if we are in
execute mode
                                thread_can_inc = 1'b1;
                                if(opcode == 6'b111111) begin
                                        num_of_halt_next = num_of_halt + 1'b1;
                                end
                                if(num_of_halt_next == 3'b100) begin
                                        processor_done_next = 1'b1;
                                        thread_can_inc = 1'b0;
                                        processor_control_next = 1'b0;
                                        state_next = FORWARD;
                                end
                        end
                        else begin
                                thread_can_inc = 1'b0;
                        end
                end
                FORWARD: begin
```

```
                        thread_can_inc = 1'b0;
                        if(fifo_empty) begin //once we are done sending packet out, then
ask for next
                                mem_pointer_reset_next = 1'b1;
                                if(!modeTopLevel[2]) begin
                                        state_next = REGISTER;
                                end
                                else begin
                                        state_next = RESET;
                                end
                                processor_done_next = 1'b0; //processor not done again
                        end

                end
                REGISTER: begin
                        processor_control_next=1'b1;
                        thread_can_inc = 1'b0;
                        if(modeTopLevel[2]) begin
                                //Back to execute mode, so go back to reset then start
state
                                state_next = RESET;
                                mem_pointer_reset_next=1'b1;
                                processor_control_next = 1'b0;

                        end
                end
                RESET: begin //this is just a dummy one clock state to reset packet
pointers, and we can use it for other things later if we want
                        num_of_halt_next = 3'b0;
                        mem_pointer_reset_next = 1'b0;
                        fifo_ready_next = 1'b1;
                        state_next = START;
                        if(!modeTopLevel[2]) begin
                                fifo_ready_next = 1'b0;
                                state_next = REGISTER;
                        end
                end
        endcase
```

As you can see, in state START we are just waiting for the next packet from the previous module. Once we begin to receive that, we load the module and ethernet headers into data memory in HEADER state. Then we load the IP header and the rest of the packet payload into data memory. Then, we are in PROCESS mode. This is where our pipelined processor has access to the memory, and the instruction counter begins to execute. You see that if we are not in ‘execute mode’, we will not leave this state. This is done so that we can program the memories through our register interface. Read part C of this section to understand the mode structure. If we are in execute mode, though, we will execute all instructions until we read a ‘HALT’ (opcode 111111) instruction. At that point we will forward the packet, then spend one clock in RESET state to allow pointers to reset, and then go back to START state to wait for the next packet. At any point, we allow the control node to program the data memory through the register interface. This is accomplished with the REGISTER state, which can be considered an

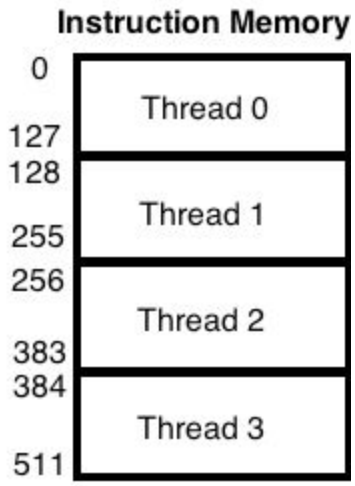
‘override’ state. Any time we receive a register write request, we immediately service that request, then return to normal packet processing. We took this basic processor as a building block, and then added hardware threads, and duplicated the cores to allow for parallel packet processing.

Implementing Multiple Threads on the Network Processor

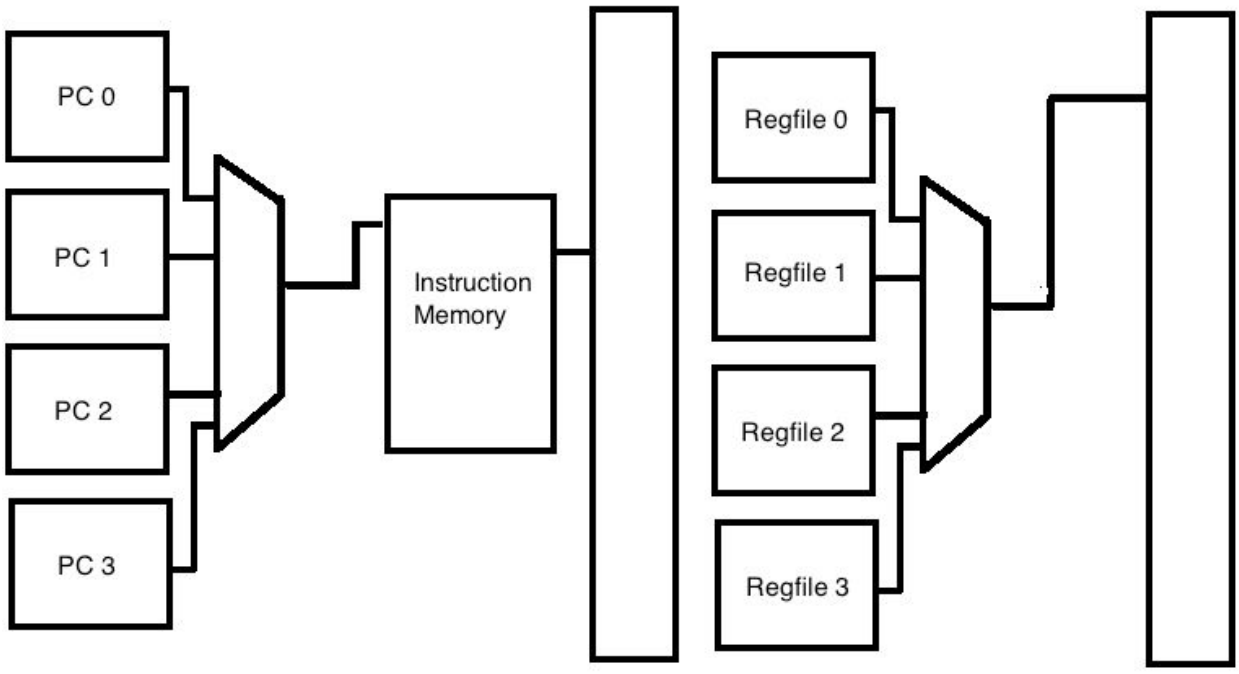
We chose to implement four hardware threads on our network processor cores. This allows us to use a simple round robin scheduling mechanism, and reach maximum efficiency without any slice-consuming forwarding or hazard detection modules. Following is a table that shows an example instruction issuing schedule among our four threads:

Clock Cycle	**Instruction Issued**
0	Instruction 0, Thread 0
1	Instruction 0, Thread 1
2	Instruction 0, Thread 2
3	Instruction 0, Thread 3
4	Instruction 1, Thread 0
5	Instruction 1, Thread 1
6	Instruction 1, Thread 2
7	Instruction 1, Thread 3
8	Instruction 2, Thread 0

As you can see, since there will not be any data dependency between threads, we are able to fully utilize the pipeline with our scheduling scheme. In order to implement this, we divide our instruction memory address space into 4 equal partitions. Thread 0 uses instruction memory addresses 0-127, Thread 1 uses instruction memory address 128-255, and so on. This diagram shows the full layout of the instruction memory:



We maintain four separate program counters. On each clock cycle, the correct program counter is selected, so we can support jump instructions and subroutines in each individual thread. Additionally, each thread is given its own register file, with 16 (OR 8?!?! IF SIZE IS AN ISSUE!!) registers. This allows each thread to operate autonomously on data, without worrying about context switches or anything. To make this design clear, let's look at what the first two stages of our pipeline look like, from the perspective of a high level block diagram:



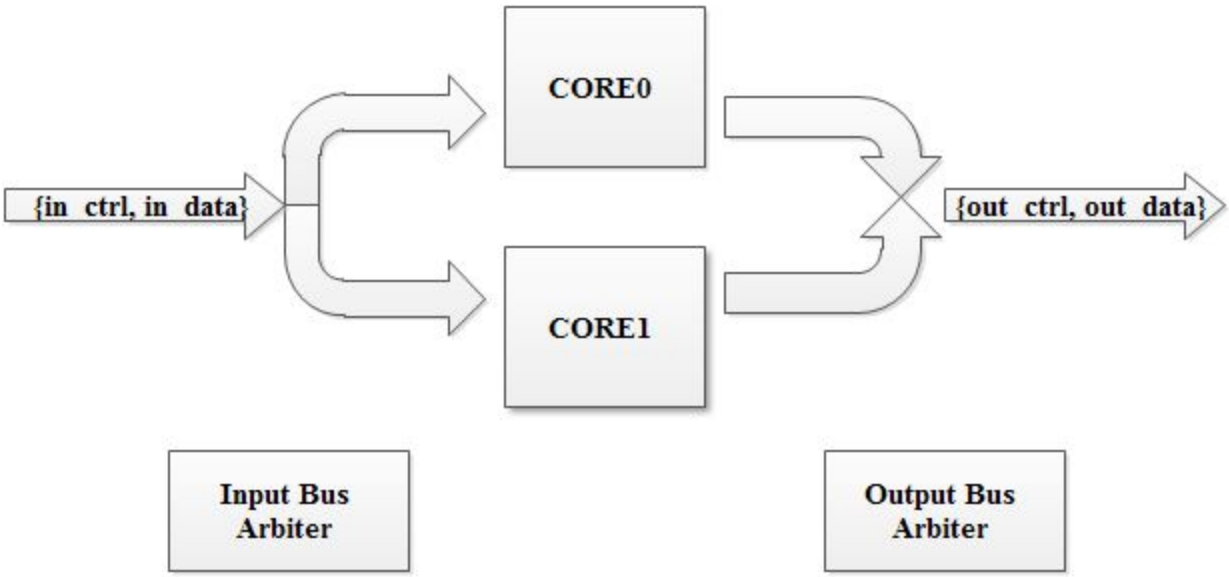
There is obviously some additional logic here, which can be examined by looking at gyn_datapath.v. From a high level though, one can envision our multi-threaded system through this design. The round robin scheduler allows us to ignore data dependency within any

individual thread, and the separate register files allow each thread to use register operands without worrying about context switches at all.

We use a shared data memory, making no changes from previous labs. This is similar to many multi-threaded systems. A shared memory is much more efficient in terms of space, which is important to our design. Because we have not implemented any memory coherence protections, we must be a little careful when writing programs. Each thread can be assigned a memory space for program constants and data, avoiding any problems there. However, all threads use the same packet space. This means that ideally, each thread operates on a different section of the packet. At the very least, they cannot operate on the same part of the packet at the same time, and the programmer must be aware of this. To see a detailed explanation of how we developed efficient and data-safe software for this processor, please see Section 2B: Network Processor Software design.

Implementing Dual Cores on the Network Processor

We implemented a parallel multiprocessor design. This allowed us to speed up packet forwarding by providing two cores which can process data. Simple bus arbitrators direct incoming packets into the correct processor, and allow processors to forward packets to the next module when processing is finished. Our processor architecture design is show in the following high-level diagram:

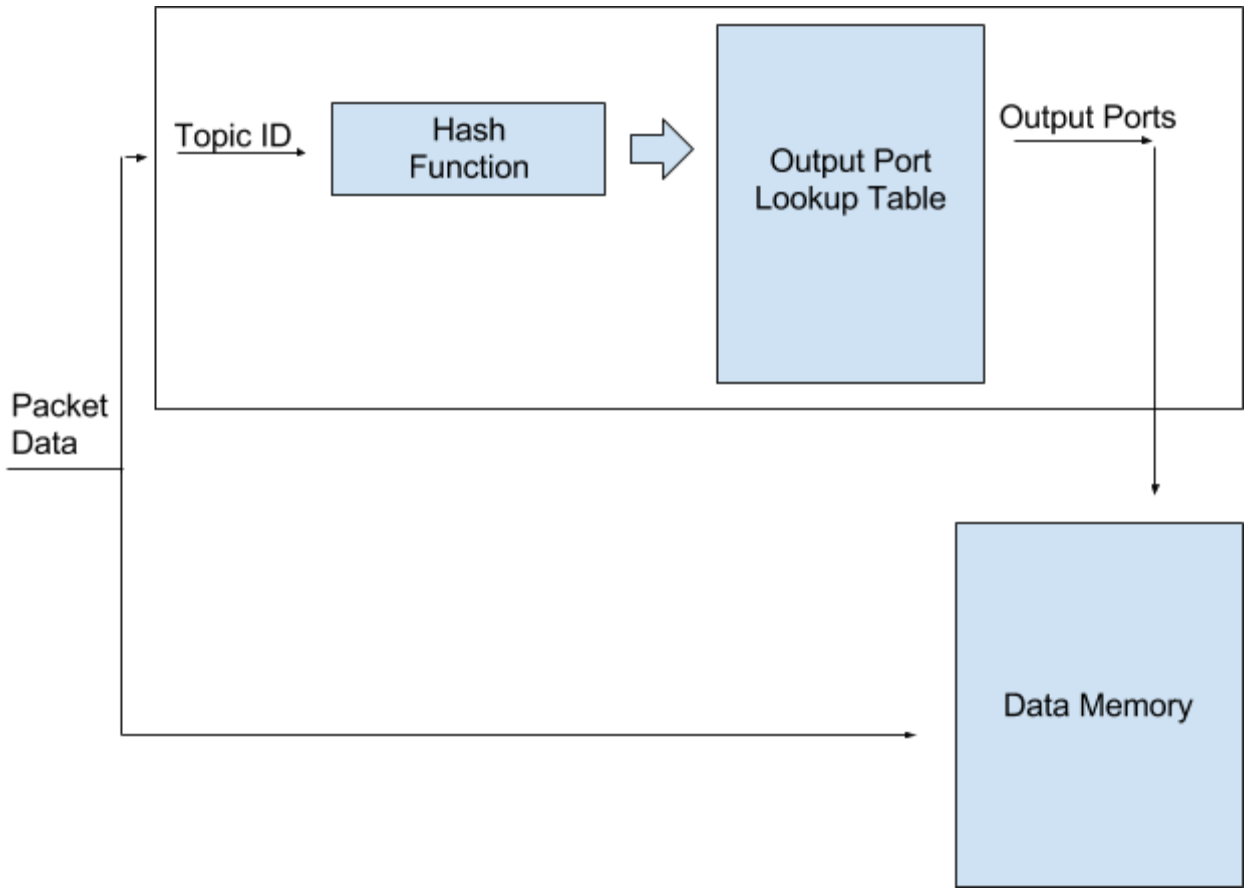


We have two identical cores and each core has four threads. Since we only have one input data and one control signal, input bus arbiter is needed to grant connections between data and cores. Our bus arbiter is round-robin, which means that if cores are idle and willing to accept data, only core0 is granted while core1 has to wait until core0 has received the whole packet. Then, core1 can be granted meanwhile core0 is processing. Then, when core0 finishes

processing, output bus arbiter grants it forwarding privileges and data is sent out. This design allows us, in theory, to process packets at nearly twice the speed of a single processor design.

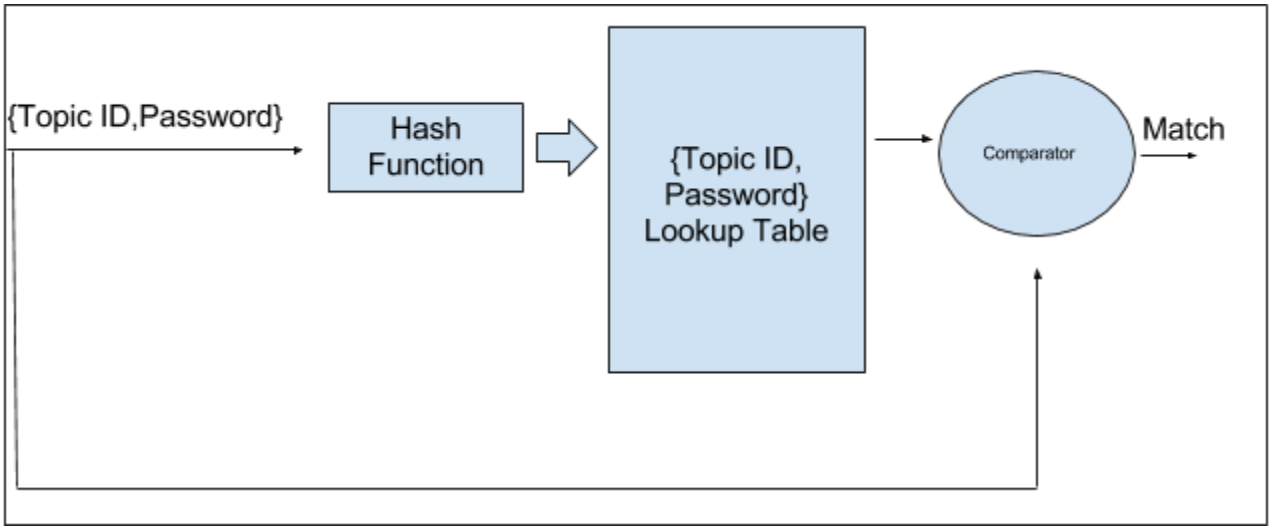
Implementing Hardware Accelerated Topic ID Matching on the Network Processor

After completing a dual-core, quad-threaded general network processor design, we ran performance tests and found roughly a performance speedup of 10x over a popular publish/subscribe broker, Mosquitto. This was mainly due to our efficient assembly software. We set out to improve this performance further by adding some hardware acceleration. We identified two major areas for hardware acceleration: topic ID lookup and password authentication. First, let's examine our design for topic ID lookup. The only messages our NetFPGA is concerned with are Publish messages, using IP protocol 253. All other control messages are handled by our software, as we have separated the control and data plane to facilitate faster message forwarding. Our Topic ID lookup uses a hash-based lookup, and allows us to associate the topic ID with the correct output port module header in real time, as the packet is being loaded into the data memory. In this way, we save time as the Network processor already has the correct module header by the time the packet is loaded into data memory. The following diagram illustrates our Topic ID hardware:

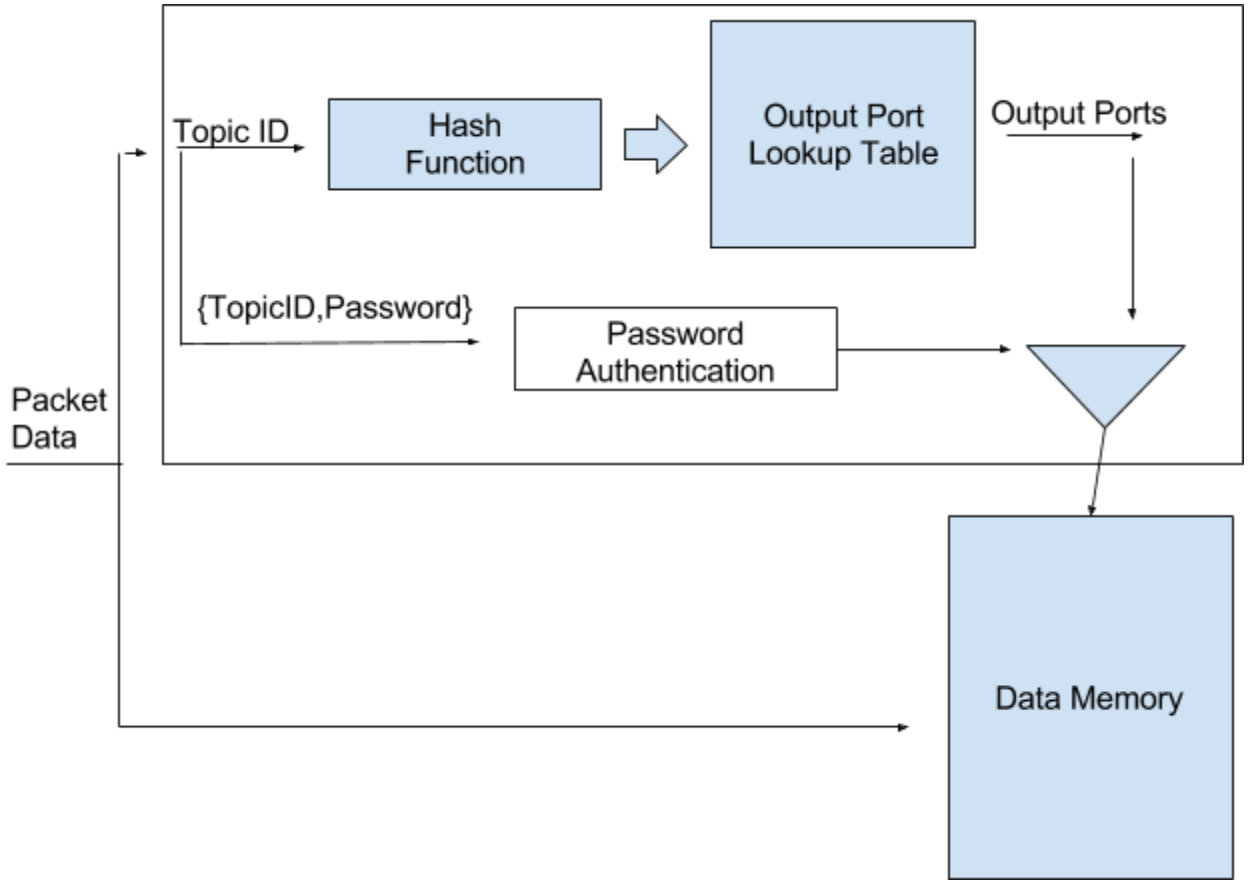


As you can see, packet data is simultaneously loaded into the data memory, as well as fed through the output port lookup table. We use our network state machine to only enable the hash function and output port lookup when the topic ID data is in our hardware accelerator, with all the other data harmlessly flowing through the module. In this way, by the time the packet is fully loaded into the data memory, we have already performed the output port lookup calculation. Since the topic ID is in the custom GYT packet header, there is most likely a significant amount of the packet coming through the data queue after the topic ID, so we get this speedup at zero cost at all. By the time the packet has fully buffered, we already have the output port matching available, reducing processing time.

Another important function of the broker is preventing an arbitrary attacker from publishing to any topic it wants. Our protocol has built in topic ID/password associations to prevent this, but the NetFPGA must have some software or hardware implementation to enforce this policy. We implemented a similar password hash-based lookup table, with the output of a match signal, to accelerate this process. This way, the NetFPGA does not have to do any additional software checks, and simply drops the packet if it is not sent with a valid password. The following diagram shows our



As you can see, again this strategy allows us to determine if a multicast message or valid or not before even the entire packet is buffered, allowing for fast discarding of invalid messages. So, our hardware acceleration module in entirety looks like this:



So, viewing the entire system, you can see that while the packet data is coming in through the input queues, our hardware is working to lookup the correct output ports for the multicast, as well as perform password validation. This allows to quickly drop malicious or erroneous multicasts, as well as quickly forward correctly formatted messages.

B. Programming the Network processor

Instruction Set Architecture

We based our instruction set mostly on MIPS. We chose to implement a 32 bit ISA, 64 bit data, and up to 8 processor registers for each hardware thread.

Rd is always destination, Ra and Rb are always operands.
R-type: [opcode 31:26] [Rd 25:22] [Ra 21:18] [Rb 17:14] [shift 13:8] [reserved 7:0]
I-type: [opcode 31:26] [Rd 25:22] [Ra 21:18] [Rb 17:14] [Immediate 13:0]
J-type: [opcode 31:26] [Immediate 25:0]

Rd, Ra, Rb have static positions, and Rd is always a destination, and Ra and Rb are always operands. With load/store instructions, Ra is the address register. and Rb is the value register.

Immediate values are represented as two's complement numbers. Immediate values will be sign extended/truncated as the operation requires.

R0 is the zero register. R15 is the special register. JC and JMP all jump to the location specified by register Ra, and for JC, it uses R15(Rb) to determine whether it should jump.

JMP and JC are implemented like the following:
JMP: [opcode 31:26] [Rd 25:22 (don't care] [Ra 21:18] [Don't Care]
JC [opcode 31:26] [Rd 25:22 (don't care] [Ra 21:18] [Rb(R15) 17:14] [Don't Care]

Following is a table with all of our instructions and Opcodes:

Instruction	**Type**	**Opcode (31:26)**	**Description**
NOOP	-	00	Do nothing
STORE Rb, (Ra)	R	01	Store value in Rb into the address (Ra) in dataMEM DMEM[Ra] = Rb;
STORE Rb, #i	I	02	Store value in Rb into the address i in dataMEM; DMEM[i] = Rb;
STORE Rb, #i(Ra)	I	03	Store value in Rb into the address Ra plus i in dataMEM. DMEM[Ra+i] = Rb;
LOAD Rd, (Ra)	R	04	Store value in address Ra of dataMEM into the register Rd; Rd = DMEM[Ra];
LOAD Rd, #i	I	05	Store value in address i of dataMEM into the Rd; Rd = DMEM[i];
LOAD Rd, #i(Ra)	I	06	Store value in address i plus Ra of dataMEM into Rd; Rd = DMEM[i + Ra];
JMP Ra	I	08	Unconditionally jump to the address specified by Ra;
JC Ra	I	09	Conditionally jump to the address specified by Ra; if (R15 == 1), jump Ra;

ADD Rd, Ra, Rb	R	14	Add values in Ra and Rb into register Rd; Rd = Ra + Rb;
ADDI Rd, Ra, #i	I	15	Add values in Ra and i into register Rd; Rd = Ra + i;
MOVE Rd, Ra	R	16	Move value in Ra into Rd; Rd = Ra; we can think of it as I-type: Rd = Ra + 0;
SUB Rd, Ra, Rb	R	17	Subtract value in Rb from value in Ra and put into Rd; Rd = Ra - Rb;
SUBI Rd, Ra, #i	I	18	Subtract i from value in Ra and put into Rd; Rd = Ra - i;
AND Rd, Ra, Rb	R	19	Bit-AND values in Ra, Rb and put into Rd; Rd = Ra & Rb;
OR Rd, Ra, Rb	R	20	Bit-OR values in Ra, Rb and put into Rd; Rd = Ra | Rb;
XOR Rd, Ra, Rb	R	21	Bit-XOR values in Ra, Rb and put into Rd; Rd = Ra ^ Rb;
XNOR Rd, Ra, Rb	R	22	Bit-XNOR values in Ra, Rb and put into Rd; Rd = Ra ^~ Rb;
LShift Rd, Ra, #shift	R	23	Left shift Ra by value in shift and put into Rd; Rd = Ra << #i;
RShift Rd, Ra, #shift	R	24	Right shift Ra by value in shift and put into Rd; Rd = Ra >> #i;
ANDI Rd, Ra, #i	I	25	Bit-AND values in Ra, i and put into Rd; Rd = Ra & i;
ORI Rd, Ra, #i	I	26	Bit-OR values in Ra, i and put into Rd;

			Rd = Ra | i;
XORI Rd, Ra, #i	I	27	Bit-XOR values in Ra, i and put into Rd; Rd = Ra ^ i;
XNORI Rd, Ra, #i	I	28	Bit-XNOR values in Ra, i and put into Rd; Rd = Ra ^~ i;
SLT Rd, Ra, Rb	R	29	Set Rd if Ra is less than Rb Rd = (Ra < Rb)
SLE Rd, Ra, Rb	R	30	Set Rd if Ra is less or equal to Rb Rd = (Ra <= Rb)
SGT Rd, Ra, Rb	R	31	Set Rd if Ra is greater than Rb; Rd = (Ra > Rb)
SGE Rd, Ra, Rb	R	32	Set Rd if Ra is equal or greater than Rb; Rd = (Ra >= Rb)
SEQ Rd, Ra, Rb	R	33	Set Rd if Ra is equal to Rb; Rd = (Ra == Rb)
SNE Rd,Rb,Ra	R	34	Set Rd if Ra is not equal to Rb Rd = (Ra != Rb)
EXPAND Rd, Ra, Rb	R	57	Lets you set all of Rd to the selected bit in Ra. Rd=64{Ra[Rb]}
BIT Rd, Ra, Rb	R	58	Lets you select a bit in Ra. Rd=Ra[Rb]
JAL Ra	I	59	Jump to address Ra, with return address put in Rd, which is always R14
HALT (special instruction)	-	63	Signals the end of a program, allows state machine to begin sending out the packet

As you can see here, we have a special opcode of 111111 for the Halt instruction. This signifies that the program is done, and the state machine can move away from the process state. See description of the Network state machine in section A for further details.

We also have a working C compiler which converts MIPS ISA to our custom instruction set. We also have automated the process of converting ASCII assembly programs to instruction memory binaries. We have two scripts that allows us to generate scripts to program instruction memory. So for example, if you write a sample assembly code, sampleAssembly, with one instruction per line, then you would run the following two commands in that folder:

```
./Parser.pl sampleAssembly
./memInterface.pl <output file>
```

The result will be a ready to run script which leverages our register interface to program the instruction memory. All you have to do manually is add any data memory programming statements that you need. The Parser script converts the Assembly to a binary string representing the instruction. The memInterface script converts that binary string to a shell command using our ./multicoreRegInterface.pl script to program the instruction memory sequentially. It automatically adds a HALT instruction, so it is not necessary to manually code that. The Register Interface will be covered in the following section.

C. Register Interface Design and Modes

The register interface is very important - it's how we get the data we need into instruction and data memory. We use a script "multicoreRegInterface.pl" to do that. It is very straightforward to use. The syntax looks like this:

```
./multicoreRegInterface.pl <command> <arguments>
```

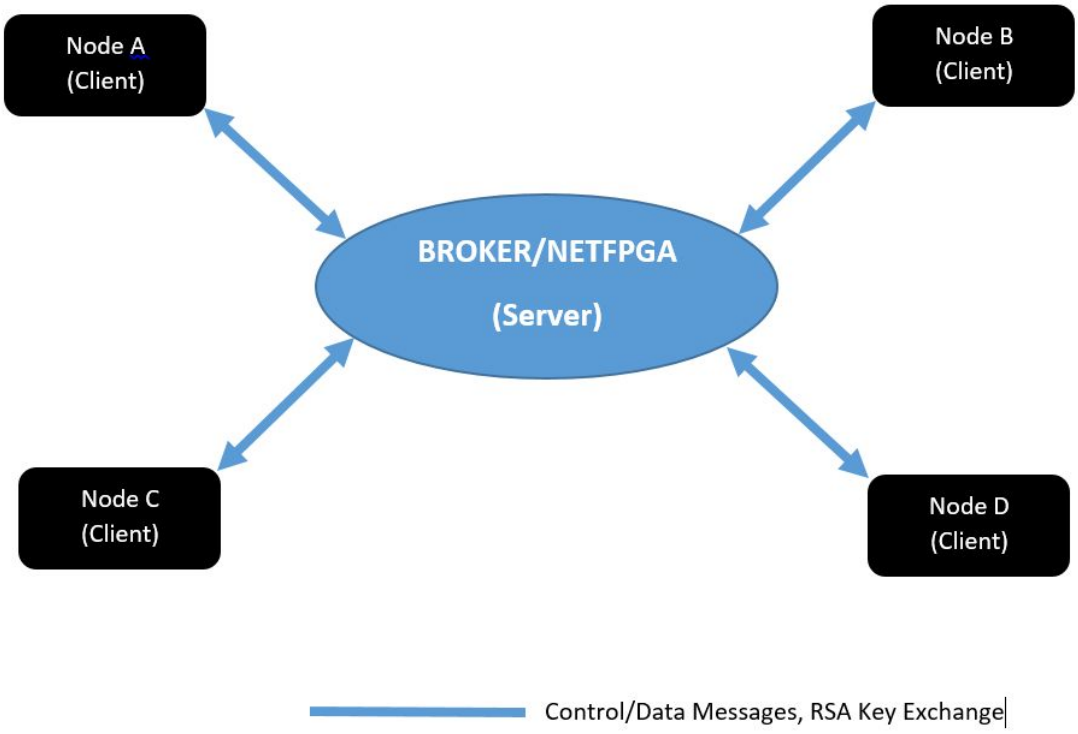
The `<command>` argument tells the script what function you want it to perform. Possible functions include read_data, write_data, read_instr, write_instr, mode_execute, mode_data_read, and others. Later in this section, we will fully detail all the different "modes", and what they mean. A full list of functions and their descriptions and syntax can be obtained by running "`./multicoreRegInterface.pl`" without any arguments. All data and addresses should be passed into the script as hex numbers. So, for example, say you wanted to write value 63 into data memory 10. You would issue the following command:

```
./multicoreRegInterface.pl write_data 000000000000003f a
```

The second part of the register interface is very important. The processor has several 'modes'. The full list of modes is execute, data_read, data_write, instr_read, and instr_write. You can look at the verilog files gyn_processor.v and gyn_datapath.v to see exactly what the behavior is under these modes. Essentially, when the processor is in 'execute' mode, it just executes instructions normally. When it is in any other mode, the address inputs to instruction memory/data memory are multiplexed so that they get the values in our software registers, so we can read and write to the memory. Data_read or instr_read modes are considered safe. I would not recommend manually putting the processor in data_write or instr_write mode, as you could unintentionally overwrite things. A call to `write_data` will put the processor in data_write mode, then back into data_read mode, so that we have control over the inputs at all times.

SECTION 5: GYT SOFTWARE AND PROTOCOL DESIGN

A. Software Implementation Overview.



The software implementation is two fold: one running on the ControlNode/Broker and the other running on the end nodes. The software running on the control node basically coordinates multicast broker functions among the end nodes(connected to the output ports of the NetFPGA). Currently two experimental protocols are used for communication between the broker and the nodes, one for the exchange of control messages and the other for the exchange of data respectively.

The broker itself is an OpenFlow style NetFPGA router acting as accelerated multicast forwarder. It maintains state in software and every time there is a change through control messages, it reprograms the NetFPGA. It has hardcoded nodeID/port associations. The software running on the end nodes (clients) enables them to perform various multicast functions which includes and is not limited to creating/joining a multicast group and publishing data to a multicast group.

RSA key exchange runs on all nodes and the broker(Server). This establishes a PEM file with an AES64 key for communication of control messages between the broker and clients. Once key exchange has been established, sensitive data namely the the topic ID, multicast group

name and password including the published data is encrypted using the symmetric key(different for each node) and sent out on the network.

B. Network Protocol Information

As mentioned earlier, the system employs two experimental protocols namely the IP protocol 254 for the exchange of control messages and the IP protocol 253 for the exchange of data messages between the broker(NetFPGA) and the end nodes. The IP headers are unchanged whereas we use custom headers for the control messages.

Custom Header Structure:

Byte	**Field**	**Description**
0 – 1	Source	Source ID of the sending node.
2 – 3	Packet Size	Size of the packet.
4 – 5	Packet Type	Control Message Type
6 – 7	Unique ID	Unique ID for each Multicast Node in the system.
8 – 261	Data	Data
262 – 263	Password	Password to create/join a Multicast group.
264 – 271	Multicast ID	Multicast ID to be created or join.

Control Messages are mainly sub-divided into 3 types:

1. **Request Multicast groups available(Node > Broker)** (Control Packet Type: 20)
   An end node can request the broker to list all active multicast groups, their multicast names and IDs.
   Response (Broker > Node): Broker provides a list of all Multicast IDs and corresponding Multicast names to the Node ID that requested it.
2. **Create a Multicast group (Node > Broker)** (Control packet Type:21)
   Control message sent by a node to create a Multicast group with the requested Multicast Name and ID followed by password. The group is created if a group with the same name or ID currently doesn't already exist.
   Response(Broker to Node): Success or Fail acknowledgement is sent out by the broker to the receiver. If successful, Multicast key is also sent out in the response.
3. **Join a Multicast Group (Node > Broker)** (Control Packet Type: 25)

Control messages with which nodes can request to join a Multicast group and if approved by the broker is added to the group.
Response(Broker to Node): Acknowledgement Success or Failure is sent out to the NodeID from the broker.

Structure of Control Response Messages:

Multicast List Response

Byte	**Field**	**Description**
0	Request Service Acknowledgement	Returns Success or Failure
1	Multicast Group Names	Lists the Multicast group name
2	Multicast Hosts	Lists all Multicast groups hosted by the Node ID

Multicast Join Response

Byte	**Field**	**Description**
0	Request Service Acknowledgement	Returns Success or Failure
1	Private Key	Symmetric Private Key
2	Multicast Group ID	Returns the Group ID to which the node Subscribes

Data Messages:
The data that is finally exchanged between a Publisher and a Subscriber is sent to the broker by the publisher. These messages are processed by the broker and published to all subscribers of a Multicast Group. The data messages are sent across the experiment IP protocol 253.

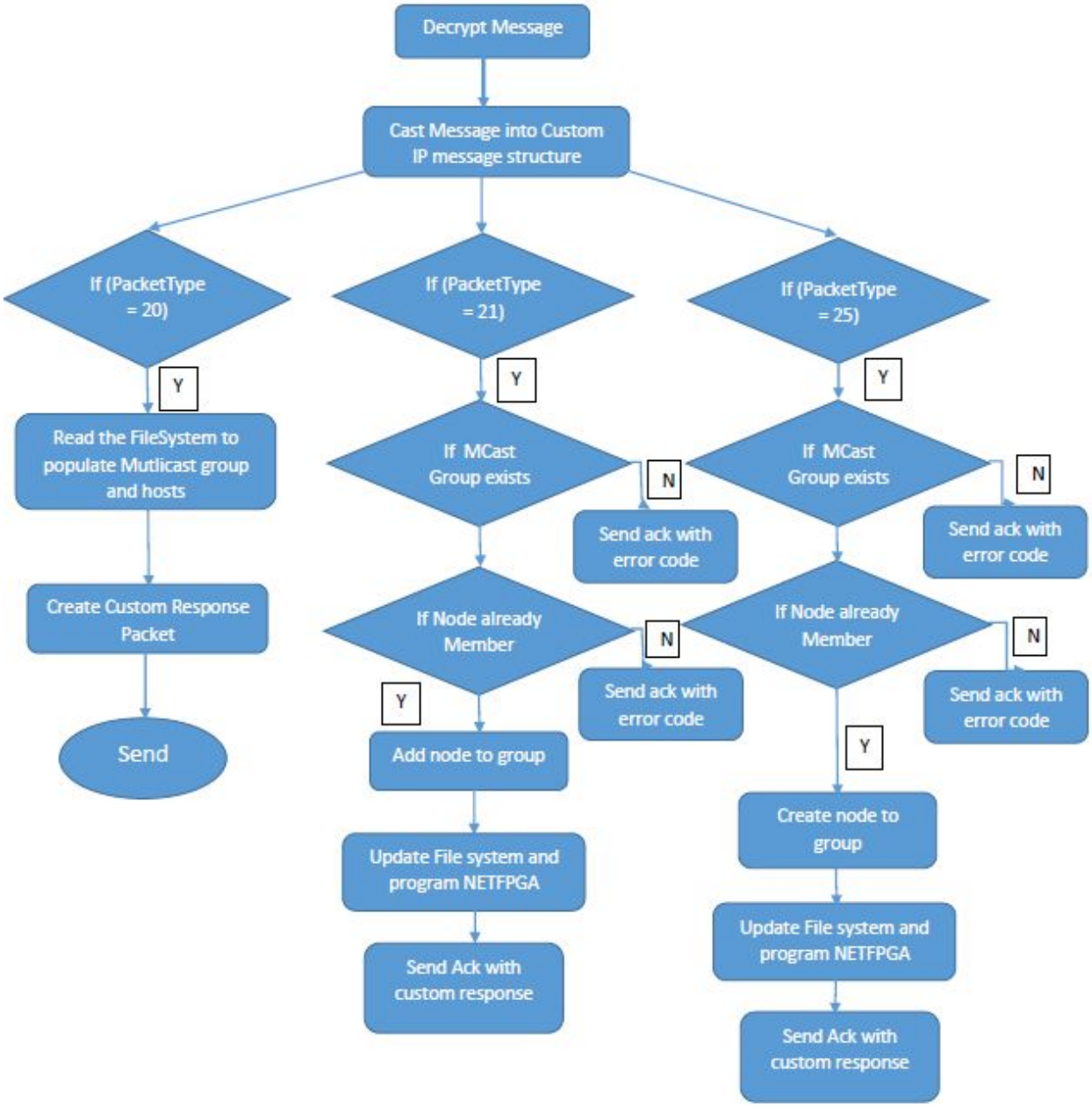
Data Messages Custom Packet Structure

Byte	Field	Description
0 – 1	Source	Source ID of the sending node.
2 – 3	Packet Size	Size of the packet.
4 – 5	Packet Type	Data Packet Type
6	Unused	--
7 – 8	Unique ID	Unique ID for each Multicast Node in the system.
9 – 264	Data	Data

C. Network Processor Software Design

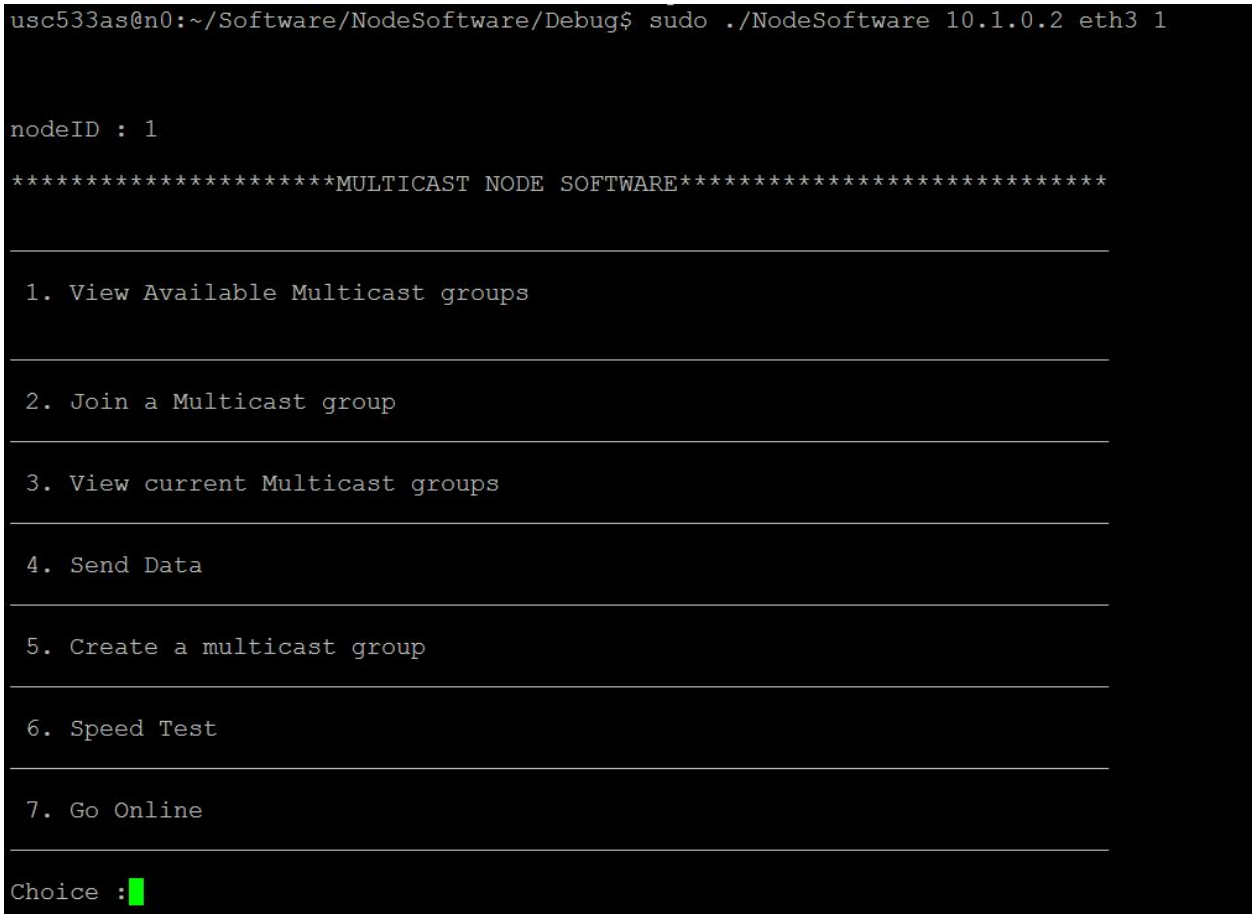
We carefully designed our network processor software to work efficiently with our quad-threaded design. We currently utilize two out of our four threads. One thread handles encryption and decryption of multicast packets. That allows future developers to make content-based changes to published messages on the broker. Another thread is the main forwarding thread. It filters out IP protocol 253, our multicast messages, and updates the module header on those packets. All other packets, it simply forwards. That way, our router still functions as a fast IP router, with additional broker support built on top of it. This software is still extremely lightweight, as all of the intensive work, like topic ID matching for module headers, is done by the hardware accelerated modules. The other two threads are currently unused. This provides an interface for future developers to add extremely fast software and functionality on top of the basic broker usage. We provide an API for developers to use the other two threads for their own purposes. All of our NetFPGA software can be found as attached source code.

D. Broker Controller Software Design



E. End Node Software Design

The below figure gives an overview of how the Node Software can be invoked and the options available for a node to create/join a multicast and so on.



Description of Node Software Options:

1. View Available Multicast Groups
   This option lets the user view the active and available multicast groups.

2. Join a Multicast group
   This option allows the node(user) to join an already existing group.
   It prompts the user to enter the multicast ID to join and password to authenticate the transaction. The software waits till the broker responds with the acknowledgement.

3. View Current Multicast groups
   This option allows the node to view all the current Multicast groups hosted by the node Itself. It reads the file system list out the subscribed topics.

4. Send data
   Prompts the user to enter the Multicast ID and associated password followed by the actual data that it needs to publish to the Multicast group.

5. Create a Multicast Group
   This enables the node to establish a multicast group by passing the group name, ID and password. The node software waits for the response from the broker to get the topic id and updates the file system.

6. Speed Test
   This options enables any user to test the speed of the Multicast system by sending multiple streams of data across multiple subscribers.

7. Go online.
   This option lets the node go into listener mode wherein it can snoop onto the network for receiving data from other nodes/subscribers in the network.

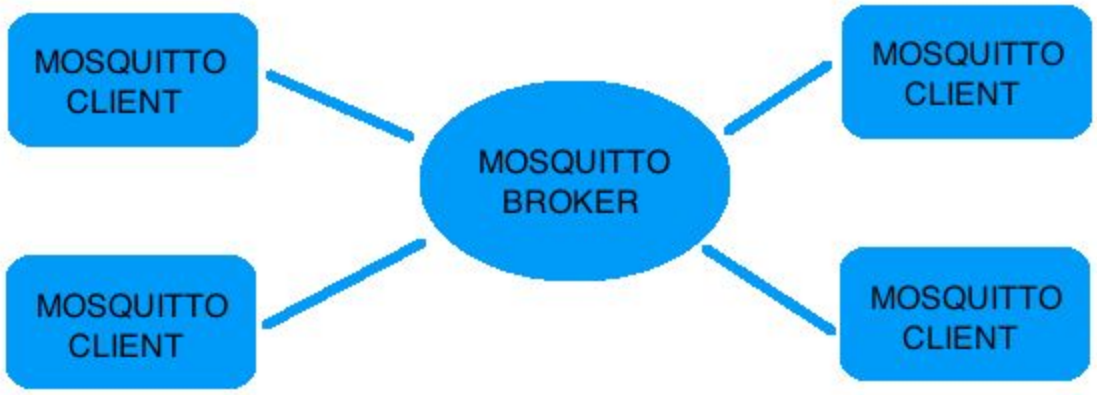
SECTION 6: GYT TESTING AND RESULTS

A. Existing Methods Comparison

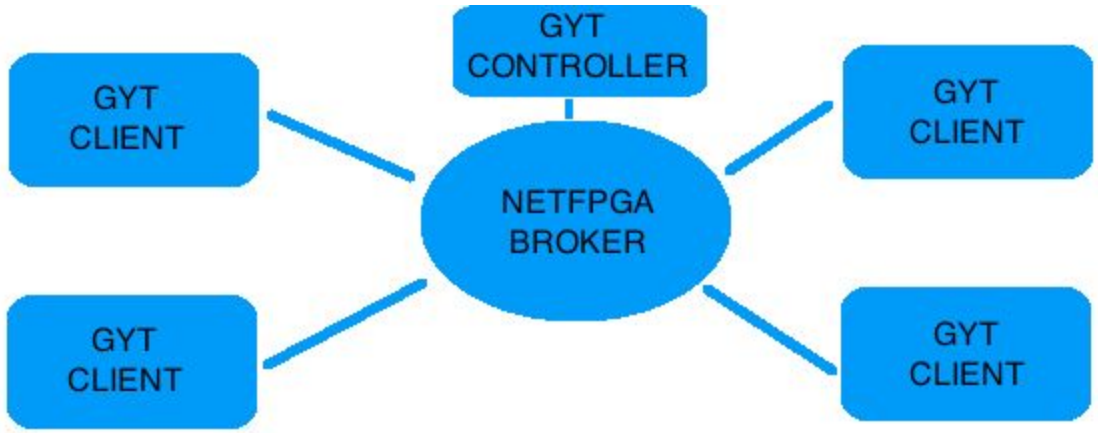
There are various existing implementations of public/subscribe protocols,and many are built on IP, just as ours is. With our design we aimed to implement a client and broker pub/sub architecture, prioritizing speed and security. Our hardware accelerated broker, which maintains topic and routing tables, provides fast multicast capabilities. Our custom IP protocol is implemented for publish/subscribe client applications, providing a lightweight API. This essentially mimics open-source MQTT broker and client libraries such as Mosquitto: http://mosquitto.org/

Mosquitto is a popular, open-source MQTT implementation. It provides source code for a broker you can run on a Linux operating system, which aims to be fast and lightweight. Additionally, it provides a client API to perform public/subscribe actions, which can be called from python or C programs, just as our design does. Additionally, since MQTT is used by messaging and multicast platforms, it closely parallels our design. While testing, we set up a Mosquitto network and GYT system in the following experimental configuration:

Mosquitto Network:



GYT Network:



In order to gauge our solution, we considered it important to test both speed and scalability. We wanted to consider both top performance with both one and several subscribers to a topic, as well as top performance with varying number of simultaneous cross-traffic streams. Following are results for both the Mosquitto network and the GYT network. Each of these numbers is an average result over 10 trials:

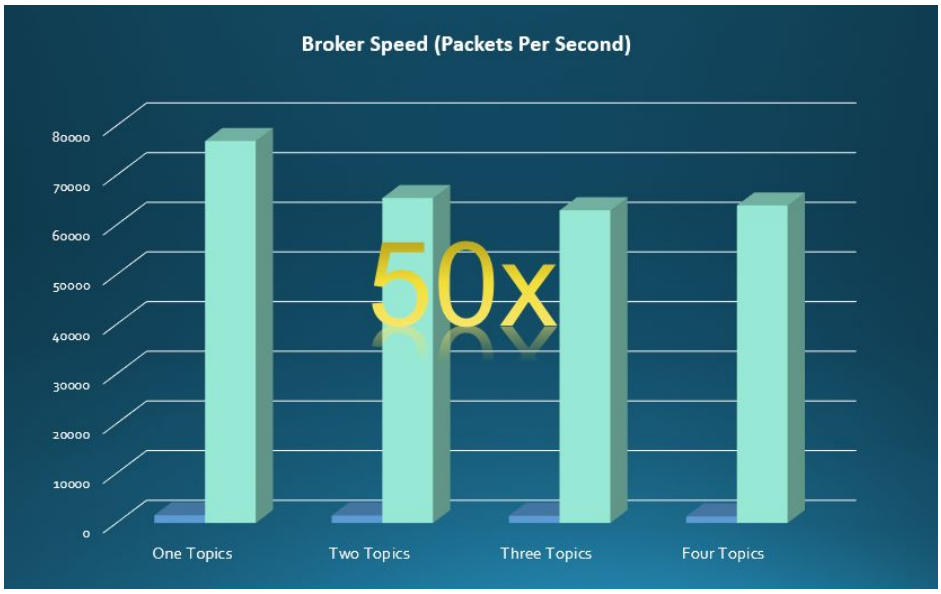
Mosquitto Network Results (messages per second):

	One Stream	**Two Streams**	**Three Streams**	**Four Streams**
One Subscriber	1578	1504	1452	1331
Two Subscribers	1568	1367	1328	1267

Three Subscribers	1521	1326	1269	1139
Four Subscribers	1356	1222	1149	977

GYT Network Results (messages per second):

	One Stream	**Two Streams**	**Three Streams**	**Four Streams**
One Subscriber	76923	65433	62976	63943
Two Subscribers	71428	63221	61134	60976
Three Subscribers	56024	54019	52082	53536
Four Subscribers	54281	52858	53912	51193



GYT Broker vs Mosquitto MQTT system

SECTION 7: ANALYSIS

The performance improvement for GYT broker for multiple data stream and multiple subscriber over Mosquitto is nearly 50 times. For multiple topics we can see that the performance of our system is almost constant unlike Mosquitto, where we saw significant drop in the performance as we increased the number of topics subscribed by a node.

SECTION 8: CONCLUSION

With this project, we have demonstrated that it is possible to build a hardware accelerated broker for a custom publish/subscribe protocol, that will not interrupt the normal flow of IP packets through the router. This allows for a high degree of flexibility in network application development. Additionally, we have demonstrated the limitations of current software publish/subscribe protocol implementations, such as the Mosquitto MQTT implementation. Because these implementations are purely software based, they cannot process high numbers of packets, posing a problem for scalable pub/sub networks. Our broker implementation successfully separates the control plane from the data plane, using an Openflow-style controller, which allows for extremely fast multicast message processing. We leave open for further research application development on top of this platform, as future researchers can utilize currently idle hardware threads to build additional functionality.

SECTION 9: REFERENCES

[1] Baldoni, R., Marchetti, C., Virgillito, A., & Vitenberg, R. (n.d.). Content-Based Publish-Subscribe over Structured Overlay Networks. *25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*. doi:10.1109/icdcs.2005.19

[2] Banavar, G., Chandra, T., Mukherjee, B., Nagarajarao, J., Strom, R., & Sturman, D. (n.d.). An efficient multicast protocol for content-based publish-subscribe systems. *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)*. doi:10.1109/icdcs.1999.776528

[3] Campailla, A., Chaki, S., Clarke, E., Jha, S., & Veith, H. (n.d.). Efficient filtering in publish-subscribe systems using binary decision diagrams. *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*. doi:10.1109/icse.2001.919117

[4] Chenxi Wang, Carzaniga, A., Evans, D., & Wolf, A. (n.d.). Security issues and requirements for Internet-scale publish-subscribe systems. *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*. doi:10.1109/hicss.2002.994531

[5] Fengyun Cao, & Singh, J. (n.d.). Efficient event routing in content-based publish-subscribe service networks. *IEEE INFOCOM 2004*. doi:10.1109/infcom.2004.1356980

[6] Kalla, R., Sinharoy, B., & Tendler, J. (2004). IBM power5 chip: a dual-core multithreaded processor. *IEEE Micro*, *24*(2), 40-47. doi:10.1109/mm.2004.1289290

[7] Moscola, J., Lockwood, J. W., & Cho, Y. H. (2008). Reconfigurable content-based router using hardware-accelerated language parser. *ACM Transactions on Design Automation of Electronic Systems*, *13*(2), 1-25. doi:10.1145/1344418.1344424