FastPass:
An Internet Architecture Resilient to Network DDoS

IXP2400 Implementation and Evaluation

George Nychis, Priya Sankaralingam, Gaurang Sardesai
Carnegie Mellon University

1 Goals

The goal of our project is to implement FastPass[1], a network architecture that is resilient to bandwidth flooding, on the IXP2400 platform for evaluation in true hardware. FastPass improves upon other works in DDoS prevention by ensuring network capabilities can be exchanged, even when the destination host is already being flooded. FastPass also does not need global cooperation of routers to perform filtering.

The architecture uses network capabilities, also found in other DDoS prevention systems such as SIFF[6] and TVA[7], to prioritize packets in flows authorized to be legitimate traffic by the destination hosts. An overview of capabilities is further discussed in section 2.1.

FastPass introduces the use of out of band tokens, signed by the destination host, to give initial packets without valid capabilities higher priority at the routers. This is important to prevent denial of capability attacks which prevent an initial capability packet from reaching the destination. This was a limitation of SIFF[6], which had no component to prevent denial of capability attacks.

The project must also incorporate other works to properly implement FastPass on the IXP2400. FastPass has only previously been evaluated in the Click modular router[8], which has access to external libraries for encryption, monitoring, and does not have the same computational and memory constraints found on the IXP2400. For these reasons we are turning to other works such as [5] for hashing functionality, [4] for parallel Bloom filters, and [3] for packet sampling and multistage filters. Further details of the implementation and integration of these works can be found in sections 2 and 3.

The architecture of our system will be introduced in section 2. This will cover the overview of how network capabilities work, how tokens obtained and used to prevent denial of capability attacks, and how monitoring prevents duplicate tokens and hosts from exceeding the rate specified by their capability. Section 3 will go through the implementation details of each of these components, as well as what layer they will reside in on the IXP2400. Section 4 provides several metrics and ideas for evaluating our implementation and the FastPass architecture in true hardware. Section 5 concludes with project management material.

2 Architecture

The FastPass architecture has three major components which will be descussed in detail seperately, but are all integrated together to provide the full functionality of the system. The first component is capabilities, which are used to prioritize packets in a flow between two end hosts. The second component, tokens, are meant to prioritize a single packet to help build capabilities. The final component is monitoring, which is used to check for duplicate tokens and used to ensure that hosts do not exceed their maximum allowed rate.

2.1 Capabilities

A capability is built and included in each packet by the network as a means of prioritizing packets to be legitimate traffic. A capability is data within a packet header which consists of non-forgeable hashes, inserted by each router as it is forwarded to the destination. Once the packet reaches the destination, if the destination wants to allow the source host to send it data, the destination sends a packet back to the source including all of the hashes marked by each router. The source can then send prioritized packets by including these hashes in the headers. The packets will be prioritized at the routers by recomputing the hashes and comparing them to those in the header. Assuming that if the destination gave the source the hashes created in the original packet, it is assumed by the routers that the destination host wants to accept the incoming traffic. If the destination did not want to allow the incoming traffic, it would not have sent the hashes back to the source host.

Packets which have valid capabilities are given priority at the routers to be placed on the fast data path since the destination host must accept the capability and send it back, showing interest in receiving the traffic from the host. Packets without valid capabilities will be placed on the slow data path and dropped when the link becomes congested. This would ensure valid traffic to be prioritized over flood traffic with no capabilities.

To prevent capabilities from being forged or shared amongst hosts, the hash is computed on the source IP, destination IP, and a rate specified in the packet. By including the source and destination IP addresses in the hash, it ensures that if the destination sends the hashes back to the source, these hashes are not valid if given to other hosts. If another host tries to use the hash to send prioritized packets, when a router recomputes the hash to verify the capability, it will fail since the source IP address has changed.

To prevent hosts from sending at any rate once a capability is acquired, a rate is included in each packet and hash computed at each router. By including the rate, a destination host can ensure that it does not allow sources to send at an aggregate rate which is greater than the bandwidth of its link. By placing the rate in the packet and hash, routers can estimate the rate of the source to the destination and drop packets if detected to be greater than the rate agreed upon in the capability. This is further discussed in the monitoring section 2.3.

The capability header is included between the IP layer and transport protocol layer in all packets. This is illustrated in figure 1. Each capability has three fields: current, next, and echo. Capability current was built by the routers during the previous packet, was echoed back to the source, and is being used to validate the current packet. For first packet between two hosts, the capability current field is blank. Capability next is filled in by each router as a packet goes through the network. Once it reaches the destination, it is then placed in the destinations outgoing packet as the capability echo. On the first capability packet, where capability current is blank, capability next is the field being filled in by each router. Capability echo is also blank on the first packet and is used on all other packets for a host to send back a capability which it has agreed upon.

The reason for a capability next field is to prevent a destination from being locked into receiving all packets from a host that seemed legitimate, but has been further concluded to be malicious. If the hashes had no chance of ever being changed, once a destination echoes back a capability, it is good for all packets taking the same path until a new router is introduced or an old router is removed to change the hash.

For this reason, routers will cycle their private key at an interval realistically set to five seconds. This will make sure that an end host does not receive unwanted traffic for more than ten seconds, two times the cycle period. Once a private key is changed by any router on the data path, a new capability must be created. If the destination concluded a host to be malicious, it may decide to not echo back a newly created capability to prevent further packets from the source being prioritized.

To properly implement this, a router must also cache its previous key to validate traffic from a host which has not yet received the new capability. If the destination has decided not to echo back the new capability,

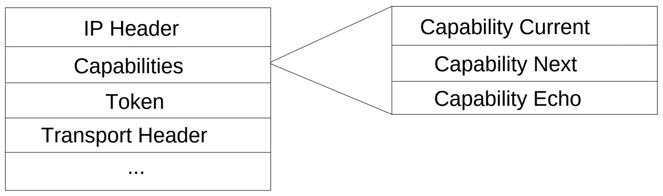

Figure 1: The token and capabilities fall between the IP header and the transport layer header. The capabilities header has three fields, capability current, capability next, and capability echo.

unwanted traffic can be received by using the previous has for up to a cycles worth of time, or a specified smaller interval. To simplify our implementation, we are forcing the cache time of previous hashes to be equal to a cycle period.

An example of creating a capability is given in figure 2. In the figure, S is the source, D is the destination, *R1-R3* are routers, and K_{R_n} represents the private key of router *n*. As the packet is forwarded by each router, the router computes a hash based on the source IP, destination IP, and the rate and appends it to the capability next field. The capability current in the example would be blank if it were the first packet, and in the case of all other packets, as long as the current packet is taking the same path as the last packet, and the routers have not changed their key, it would be: "*&#". The reason for this is that the destination would have sent back "*&#", given the same conditions, and the current packet would be using it to validate itself.

Capability echo includes a capability that one end of the two communication has accepted. In the example, once "*&#" is verified by the destination, the next packet that the destination sends out will include "*&#" as its capability echo, so the source can use it as its next packet's capability current.

By creating these capabilities, end hosts can validate traffic to be legitimate and refuse to echo back capabilities of non-legitimate traffic to place it on the slow data path. The main issue with this design is before a capability is ever created, the first packet has no prioritization. This means that if the link is already being flooded, a first packet will likely not make it through, and a capability can never be created. This is referred to as a denial of capability attack. FastPass addresses this issue through the use of tokens.

2.2 Tokens

As explained in section 2.1, once the capabilities are established, traffic prioritization can take place in a streamlined manner. While this helps prevent denial of service, there is still another threat that the system is vulnerable to, denial of capability. Since the initial packet has not been marked by all the routers and verified by the destination, the routers cannot prioritize it. The initial packet is thus unprotected. This can be a problem when the initial capability packet is not able to get to the destination because other malicious hosts flood the links, causing them to drop the initial packets. In such a scenario, the capability packet will never get to the destination, and capabilities and traffic rates cannot be verified. In order to prevent this, the concept of tokens is introduced by the FastPass system.

Token are essentially a way of enabling hosts to prioritize a single packet for the use of setting up capabilities with a destination. These tokens, if valid, tell routers that they should prioritize the packet. At this point, the routers and destination do not know if this host is malicious, and do not need to know. The reason for this is that the token is only valid to be used once, and is prevented from being used again within a certain window of time. This will limit the rate at which a host can send prioritized packets, which will be explained in further detail within the monitoring section. The important aspect is that the token will validate a single packet to allow for the possibility of a capability being set up even during a denial of capability attempt.

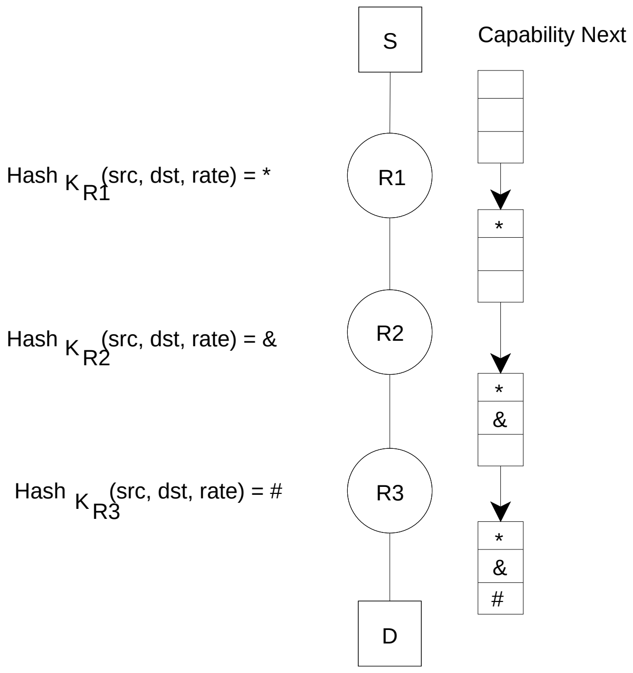
S
Capability Next
Hash K_{R1}(src, dst, rate) = *
R1
*
Hash K_{R2}(src, dst, rate) = &
R2
*
&
Hash K_{R3}(src, dst, rate) = #
R3
*
&
#
D

Figure 2: Capability headers are built using a secret key at each router to create a hash based on the source, destination, and rate for the flow. As the packet traverses the network, each router places its own special marking which will be echoed back and used on the next packet to give it priority at each router.

A token is a signed piece of data, used in the header of packets, with the destination's key. This could either be the destination's private key, if asymmetric key cryptography is use, or a shared key if symmetric key cryptography is used. Tokens have an explicit expiration time and an ID to ensure that tokens are not reused within specific windows of time and that they cannot be used forever.

2.2.1 Token Distribution

The destination's public key is distributed to the routers via routing updates. Every router that has a path to the destination knows its key. Note that in case of symmetric key cryptography, we would use a hash chain instead. This is explained in 3, due to the lack of cryptographic support in the IXP2400.

The destination hands out its token to other hosts which want to communicate with it ahead of time, completely out of band of the current communication. This distribution can take place as transactions, if the end user has an account at a web site for an example, or the requester could just obtain it through out of band mechanisms like email. Content delivery systems like Akamai could be used as well. The benefit of using Akamai is that the communication paths between Akamai servers and end hosts are over provisioned, ensuring that a token can always be obtained. The point to note here is that the token originator does not necessarily know who has its tokens and how many are out there. This is why duplication detection, covered in section 2.3.1 is important, ensuring that malicious hosts can not gather and share tokens to create large numbers of prioritized packets.

2.2.2 Token Duration

Unlike capability packets which expire within seconds, tokens can last on the order of days. End users will likely not want to be bothered with obtaining tokens at frequent intervals. Tokens thus need to have an explicit expiration time. This is checked for every token packet by the router for validity. At the same time however, reuse of the same token by other hosts should be prevented, and this duplicate suppression is addressed in section 2.3.1.

The ID is unique for every token issued by a destination that has not expired. Once the router has verified that the ID and expire time are valid, they can verify the key and prioritize the packet. This functionality is provided by the monitoring component of the FastPass system.

2.3 Monitoring

The monitoring functionality of our system has two major components: token duplication detection and rate policing. Token duplication detection will ensure that the same token cannot be used twice within a time span. Rate policing samples packets from hosts with capabilities to detect hosts trying to send at a rate significantly greater than the rate permitted by their capability.

2.3.1 Token Duplication Detection

Denial of capabilities, as explained in section 2.2, can prevent an initial capability setup packet from reaching the destination. Without a capability, a packet must be placed on the slow path and has a chance of being prevented from reaching the destination. Tokens prevent denial of capability attacks, prioritizing initial packets which do not yet have a capability to reach the destination. This gives two hosts a chance to setup a capability in the midst of a denial of capability attack.

Tokens are given out of band to hosts without the originator of the token knowing nothing about the host who obtained it. The host who obtained the token may be malicious, but the originator cannot determine this. These tokens are used to give packets higher priority and have the possibility of being sent from malicious hosts. One could then imagine malicious hosts cooperating in a distributed denial of service attack, each obtaining a token and then sharing it to create a large number of prioritized packets with no capabilities to the destination.

The first major component of monitoring, token duplication detection, must keep track of which tokens were seen within a span of time to prevent this kind of attack from occurring. Monitoring must ensure that malicious hosts cannot share tokens to gather large numbers of prioritized packets with no capabilities, to create a distributed denial of service attack at the destination.

To detect duplicate tokens, the router must keep a table of tokens already seen and lookup a newly received token against this table. If a router finds the token already in the table, it will place the packet in the slow path, not giving it a higher priority. Not all tokens seen by the router can be stored for lookup due to a lack of infinite resources. This forces the router to use and maintain a time window worth of tokens already seen. This table must be updated as new tokens are seen and old tokens expire.

This window defines the fastest attack that can be done per host using a single token and the amount of memory needed for the cache. A trade off must be made between the two. If the duplication detection window of time is five seconds, the maximum rate a single host can attack a destination with a single token is one packet every five seconds. A duplication detection window of five seconds also means that the upper bound on memory can be determined assuming a unique token is received with every packet and all ports are receiving their maximum capacity of packets. This would be the size of a token, times the maximum number of packets that can be processed by the router in five seconds.

2.3.2 Rate Policing

When a destination agrees upon a capability requested by a source, it also agrees upon a rate that the source can send at. This rate agreement helps a destination prevent hosts that it thought was legitimate from beginning to maliciously send at a faster rate than its connection can handle. By handing out rates with capabilities, the destination can ensure that it does not hand out a greater aggregate bandwidth than its link can handle. The routers in the path between the source and destination can also do packet sampling to estimate the source's rate, dropping packets sent over this rate.

To do rate policing efficiently in a router, a method of sampling must be used rather than classifying each packet into a flow and keeping counters per flow. This would require too much memory and computation to for the router to keep up with line speeds. An exact rate need not be determined. If the router can estimate the rate with packet sampling within a reasonable deviation from the actual rate, a threshold can be used to drop packets based on the estimation.

Estan et al[3] propose sampling of packets and the use of multistage filters to find heavy hitters amongst traffic flows. Their work fits closely to the requirements needed to do rate policing in our project. The technique proposed has a constant amount of memory references per packet and needs a small amount of memory to estimate the rate of a flow. The motivation of these requirements, as stated in the paper, is the increase of link speeds and number of flows which are too expensive to track with counters per flow in SRAM and too slow to track in DRAM.

Using the work proposed by Estan et al will allow for scalable rate policing in the scope of our project to prevent hosts from exceeding some threshold of their permitted rate. Once a host is detected to be sending at a rate greater than the permitted rate specified by the capability, packets will be dropped in the stream.

Specifying a rate will be done in the capability packet using rate classifications. Ten different rates will be permitted for the scope of this project, which we have not yet defined. These classifications will simplify rate classifications to be defined by single numbers sent in packets, one through ten. Routers can then check the rate markings against the estimated rates calculated from packet sampling to drop packets from hosts if needed.

3 Design

The design of FastPass places functionality in the IXP2400 on the data plane, control plane, and the management plane. Much of FastPass's functionality requires modifications or computations with every packet, through the capability header and sampling. Part of the functionality of monitoring will also reside in the management level, to allow an administrator to check the functionality of its policing and gather statistics about the system.

While reading through the design setions reference Figure 3 which shows the flow of a packet through the FastPass architecture. The state diagram shows which components read from the packet and how all of the components interact with each other.

3.1 Hashing

Before we introduce the design decisions of each component, a major functionality needed by each component is hashing. Routers use hashes to verify the authenticity of tokens and the parameters of a capability header. These hashes can become the computational bottleneck of the architecture, possibly limiting the line speed.

The IXP2400 has a hardware hash unit, SHaC (scratch, hash), which is capable of generating 48-bit, 64-bit, and 128-bit hashes. Generating a hash with SHaC will place the hash in the scratch pad and takes 16 clock cycles. The hash lengths we will use for each component are further discussed in their respective sections.

In the original FastPass document, asymmetric cryptography was proposed for distribution and verification of tokens. Due to the lack of cryptographic support in the IXP2400, symmetric key cryptography will be used. Private keys will be used for hashing and these keys will be shared for verifications.

Other hash functions for hardware have been proposed by by Ramikrishna et al [5]. Several of these hashes are suggested in the sampling and multistage filter work[3] to be optimal. Analysis of other hardware hash functions could be done in conjunction with this work, however SHaC will be the only hash function used given time constraints.

3.2 Capabilities

Capabilities must be done in the data plane of the IXP2400 and implemented in the micro engine so the marking of capability bits into the header can be performed at line speed. This functionality is done on each packet and would take a considerable performance hit to send all capability packets into the control plane and XScale core. The capability component does not require any special data structures, it only needs to read information from the packet, compute a hash, and then write back to the packet.

The capability is first read from the data packet which is stored by the IXP2400 in SRAM. The read of the capability header is done at an offset of the end of the IP header. The capability header has three major fields of interest: *pclass, crouter, hashes.* The *pclass* specifies the maximum rate the source can send at. This is monitored by the rate policing in the monitoring component. The *crouter* is the current offset to use in the list of hashes within the capability header. Each capability header has a hash for every hop. A *crouter* field is read and then incremented, which is an offset to the current spot in the capability header which represents the current routers hash. This will be further discussed. The *hashes* field is a multiple entry field, which has one entry per hop and represents that hops hash computed on the packet header information.

The interaction between the capability field and the microengines are as follows. The capability header is stored within the packet which is stored in SRAM. The first thing done, after it has been determined a capability header exists, is *source IP* and *destination IP* being read into general purpose registers. The *rate classification* is also read from the capability header into a register. These three fields are used to compute all capabilities hashes.

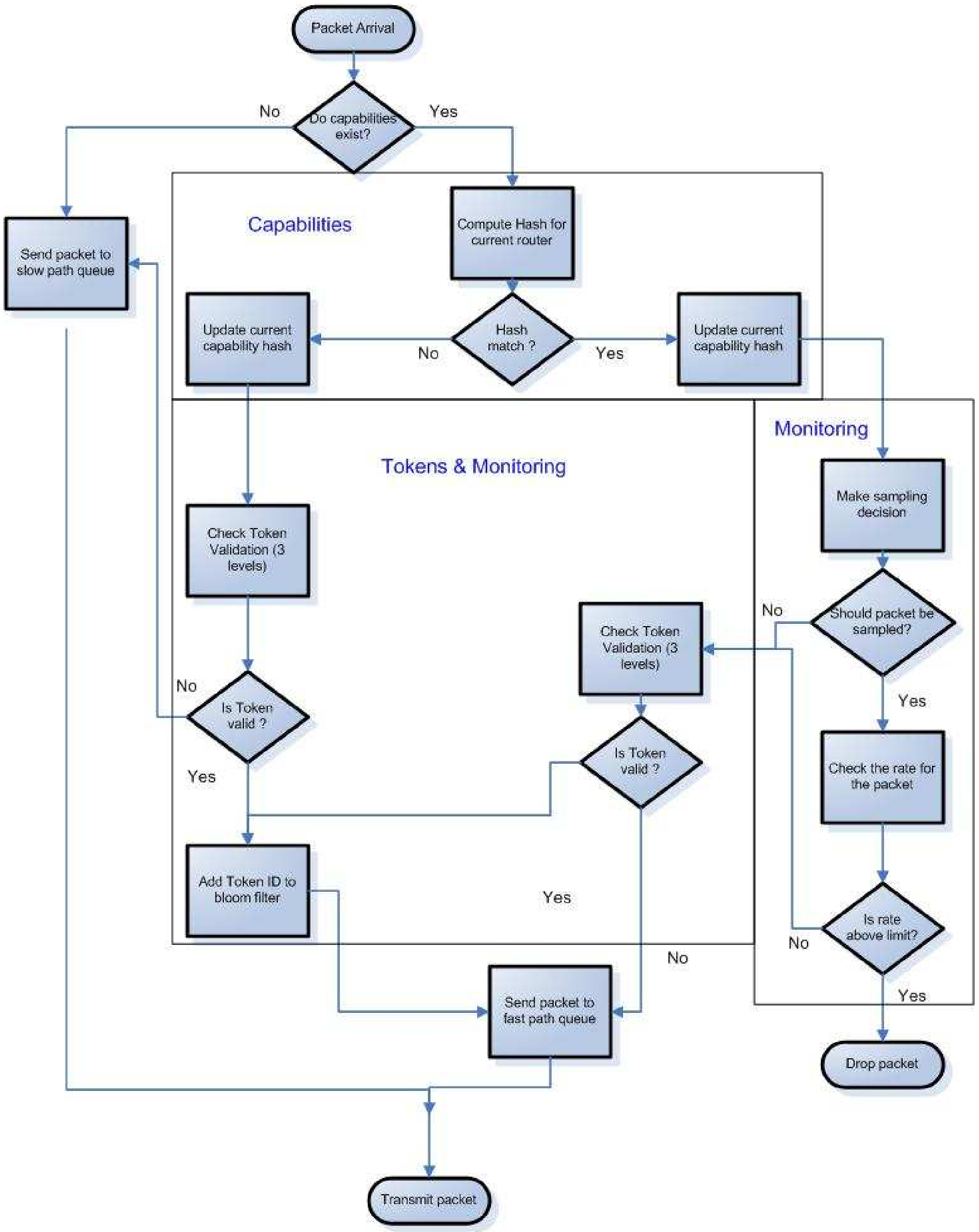
Packet Arrival
Do capabilities exist?
No
Yes
Send packet to slow path queue
Capabilities
Compute Hash for current router
Hash match ?
No
Yes
Update current capability hash
Update current capability hash
Tokens & Monitoring
Monitoring
Check Token Validation (3 levels)
Make sampling decision
Should packet be sampled?
No
Yes
Check Token Validation (3 levels)
Is Token valid ?
No
Yes
Is Token valid ?
Check the rate for the packet
Add Token ID to bloom filter
Yes
Is rate above limit?
No
Yes
No
Send packet to fast path queue
Drop packet
Transmit packet

Figure 3: The flow of a packet through the FastPass architecture, which components read from the packet, and how all of the components interact.

The next thing done is the reading of the actual current capability hash from the header. With a hash length of 8-bits, which we will be using for our project, the current hash is located at an offset of $64 + 8(crouter)$ bits from the start of the capability header. This hash is read into a general purpose register and will be used to compare to a recomputed hash.

Hashing is now done using the *SHaC* hash unit on the three fields mentioned: *sourceIP, destinationIP, rateClass*. To perform the hash, the values used must be placed into transfer registers. Therefore the next step is getting these three fields into transfer registers. Once the fields are in transfer registers, a *SHaC* hash using the routers private key is computed. This hash should match the one found at the *crouter* offset in the capabilities header.

If the hash fails, the hash on the old private key needs to be recomputed. Since we are allowing a windows worth of old capabilities to be used, we must compute this hash and also compare it to the one found at the *crouter* offset. This is again done by placing the data into transfer registers and passing them to the *SHaC* hash. In all cases, the *SHaC* hash overwrites the old values in the transfer registers. After doing this, the header has been changed, the checksum will also need to be updated.

If the hash for *crouter* in the header does not match old computed hash also, the packet must be placed in to the slow path queue. If one match occured between the two hashes then the packet will be placed on to the fast path queue.

Regardless of whether the capability header is valid or not, the current hash computed on (*sourceIP, destinationIP, and rateClass*) tuple is written directly into the write side of SRAM back into the packet. It does not matter if this value has changed or not, writing it every time will prevent a check from occuring as to whether it has changed, and the newest value needs to be kept in the header regardless. By keeping the newest value we ensure that even if a capability expires through a router key cycling, the destination at least gets the new value and has the chance to determine whether it is going to echo the value back.

The memory layout for the capabilities are as follows, note the duplicates in SRAM transfer registers are for the *SHaC* hash unit:

Memory	Data in the memory
SRAM	packet, private_rkey, prev_private_rkey
Registers	sourceIP, destinationIP, rateClass, crouterHash, crouterNewHash, checksum
SRAM Transfer	sourceIP, destinationIP, rateClass

3.3 Tokens

Since tokens are used to prevent denial of capability, it is imperative that this first step should be as resilient to failure as possible. The token is a piece of data consisting of the ID of the host and the expire time of that token, signed by the destination's private key. A token is of the form $T_1 = K_{D^{-1}}(expire\_time, ID)$. This token in then placed as an option in the IP header. When a router receives a packet, it checks to see if it has already seen this token for the same source destination pair before. It already has the public key for the destination, so it can verify whether the token is authentic or not.

Since the IXP2400 does not support native cryptography, using asymmetric keys for signing will be prohibitively expensive. Hence, it would make more sense to use symmetric keys. In order to implement this, though, a hash chain configuration would be required, where each node has a shared secret set of keys with its immediate neighbors, i.e. the source S shares a set of keys with router R_1, R_1 with R_2 and so on, until the destination. This is effectively a sequence of hashes. In the real word, this would be implemented at an AS level, and all the routers within an AS trust each other. For our project however, we will implement it at the router level. Thus the token is authorized at each step along the way.

Tokens are typically only present in the first packet. Once a capability has been established, capabilities are embedded in packets henceforth. Tokens involve lookups for validation of expiration date and IDs. Also,

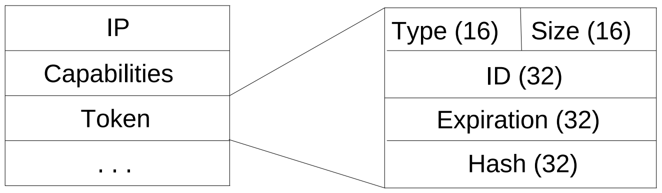

Figure 4: The token header comes underneath the IP and the capability headers. The total size of the header is 16 bytes and consists of a type, size, ID, expiration, and hash field.

if the expire time is valid and the ID is valid, the key of the destination still needs to be verified. These lookups and validations are time intensive. Also, since they do not need to be done for every packet, sending the token packets up to the XScale core for the operations is a logical choice. The XScale is more suited to perform the lookups and perform hash verifications. This leaves the microengines free to perform fast path operations for the capability packets.

We chose to implement tokens in the data plane because the functionality to perform verifications is already present in the data plane. In addition, tokens interact with the duplication detection part of the monitoring module, and it is imperative that these verifications and lookups be performed as efficiently as possible.

As can be seen from Figure 4 above, tokens lie immediately after the IP and the Capabilities header. Since the capabilities header is always a fixed size of 16 bytes, this makes it easy to get the offset to retrieve the various fields of the token header. The token header consists of 5 fields and has a total size of 16 bytes.

- **Type: (16 bits)** This indicates the type of signature used by the destination to sign the token. Various options are Public key cryptography, Symmetric cryptography using hash chaining etc. Since the IXP2400 does not contain support for hardware cryptography, using such schemes to validate the tokens will be computationally expensive. The IXP already has a ShaC unit which generates hashes. We plan to use hashing instead of keys for token signing. Hence the type field will be a fixed value for the sake of this project which is decided by the end hosts, and which is meant to let the router know which scheme is being used.

- **Size: (16 bits)** This field indicates the total size of the header. Since this is always fixed, this also carries a fixed value. This is used to ensure that no fields of the token header are missing or have been truncated.

- **ID: (32 bits)** This field carries an ID which is unique for every source destination pair. This ID is used to ensure that the token is a genuine token. Since tokens do not carry a source address, it is necessary to check for duplicates. This functionality is carried out by the token duplication detecting macro of the monitoring module.

- **Expiration Date: (32 bits)** This field carries the expiration date for the token. While tokens carry a much larger expiration time than capabilities, it is necessary to prevent tokens from being used long after they have been invalidated. It is necessary to check if the tokens have expired already, and if so, the packet should be pushed to the slow path.

- **Hash: (32 bits)** This field contains the 32 bit hash from the source. This hash is then verified to see if it is valid. If it is not, the packet is pushed to the slow path.

The tokens go through many steps before they are consider valid and the packet can placed into the fast path queue. Figure 5 also illustrates these steps through a state diagram. The following steps are considered the path of tokens:

1. **Field extraction:** Since the token consists of 5 values we need to extract these values. Type and size are used for cursory verification, so theyre not really important. So both these values can be stored in the same register. ID, Expiration date, and Hash are extracted into separate SRAM registers.

2. **Preliminary verifications:** Preliminary verification can be carried out by masking the upper/lower two bytes of the type/size register respectively and comparing with fixed, stored values, since these values do not change.

3. **ID verification:** The ID needs to be checked for duplicate detection. The ID is passed to the duplication detection macro in the monitoring block which return a Boolean value indicating if the token is valid or not. Tokens with duplicate IDs are pushed to the slow path.

4. **Expiration Time verification:** The expiration time is checked to see whether it is valid or not. This is a simplistic calculation wherein the time is compared with the present time, implemented in the form of a timer/counter. If the expiration time from the token is lesser than the current time, this means that the token has already expired, and so the packet is pushed to the slow path.

5. **Hash Verification:** The hash present in the hash filed needs to be verified to check for authenticity. This hash value is 32 bits. However, the minimum value of hash that can be computed by the ShaC is 48 bits. The hash value stored in the hash field of the token header is computed over the ID and expiration time values, since these are the two fields that really matter. Since the ID and Expiration fields are each 32 bits, this is 64 bits in total. Hence the hash generated by the ShaC will be 64 bits in length. On the surface it may seem to not make sense to compare a 64 bit hash with a 32 bit hash. However, we need to keep in mind that the assumption here is that hash present in the token header is generated using the same mechanism that we generate our hashes in the IXP. Hence it is perfectly fine to truncate the 64 bit hash to 32 bits and compare this new value with the hash in the token header, since this is also how the token header hash will have been generated anyway. The IXP2400 Hardware Reference Manual states for a given set of data, the lower order bits of the Hash Argument are fairly unique, and so we choose to ignore the higher 32 bits. We are assured to a certain extent that collisions will be rare in this scenario. The programmable hash multiplier may be modified to tweak the polynomial equation used in the hash calculations. We consider a one one mapping between a private key for a destination and a polynomial equation. Each unique key for each destination corresponds to a unique polynomial. Using this polynomial, a unique hash will be generated. Using a different multiplier for each destination, we are assured that each destination will have a different polynomial. If there are a large number of destinations, each will require its own unique equation. This search is best obtained through a hash table, where the destination address is the key and the hash multiplier is the value. Hash tables have the advantage of O (1) searches, which are essential if we wish to implement the tokens in the data plane. In short then, hash verification proceeds in the following manner:

   (a) The ID and Expiration Time are extracted and placed in GPRs
   (b) The destination address is extracted and the corresponding hash multiplier is looked up from the hash table
   (c) This hash multiplier is used to call the polynomial function on the transfer registers
   (d) The result of the hash is stored in the same transfer registers.
   (e) This is compared with the hash value extracted from the token header. If they do not match, the packet is pushed to the slow path.

   Hash verification is the first operation to be performed. If this is successful, expiration time is checked, and then is this passes as well, IDs are checked. ID verification is an expensive process, and we only want to do it if we have to.

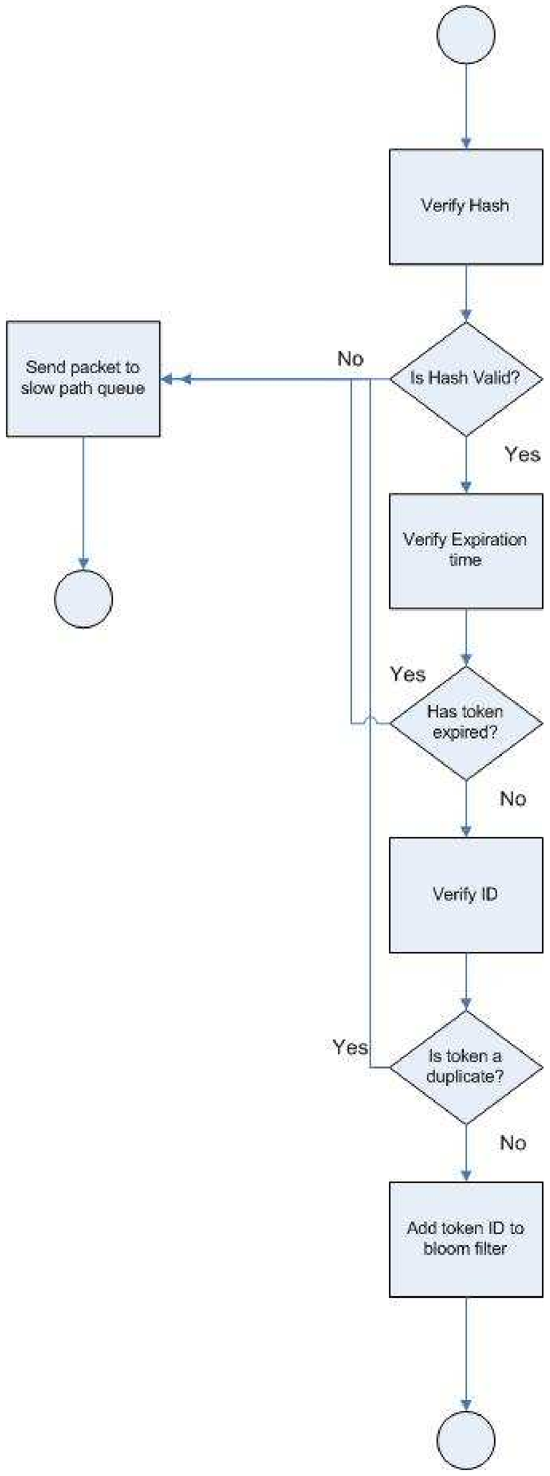

Figure 5: The following figure is a state diagram which represents the steps a token goes through.

3.4 Monitoring

The implementation of monitoring will exist in the microengines at the data plane and in the XScale at the management plane. The functionalities in the microengines are token duplication detection and rate policing, as overviewed in sections 2.3.1 and 2.3.2 respectively. The functionality in the XScale allows an administrator to connect to the IXP and receive statistics. These statistics can be estimated current flow rates, counts for packets dropped by hosts exceeding their rates, how many hosts have attempted to exceed their rate within some window of time, and tokens seen within some window of time also.

3.4.1 Token Duplication

The token duplication detection aspect of monitoring must interact and provide functionality to the token component. As incoming packets are inspected, if a token is detected in the header of the packet, the monitoring system must be able to validate or reject the token. This will be done through the work on parallel Bloom filters proposed in [4], as suggested in the original FastPass document. Bloom filters provide the functionality needed and can be implemented at line speeds in the data plane. The actual filters will be placed in SRAM due to memory requirements which will be explained in further detail.

To ensure that a denial of service attack cannot be accomplished through the use of tokens and sharing of tokens amongst malicious hosts, token duplication detection is needed. Requirements of this duplication detection system are that it produce no false negatives, few false positives, requires little system memory, and can keep up with gigabit speeds. Bloom filters meet all of these requirements. Zero false negatives using Bloom filters ensure that tokens cannot be used for a distributed denial of service attack. Bloom filters, given our parameters which we will discuss, produce such a low number of false positives that we do not need to detect and handle them. With four seperate gigabit links supported by the IXP2400, approximately 6MB of memory is needed to detect duplicates within a five second window.

A Bloom filter is a bit array of size m bits, all initialized to 0, which k hash functions will map keys to. With k hash functions, each key will be represented by k bits in the filter. Each key will be passed to each hash function, and the associated bits will be set to 1 for insertion. To query the filter for a key, the key is again passed to the k hash functions, and each bit value associated with the hashes is checked. Only if all k bits are set to 1, will the bloom filter return a positive. This functionality will provide the insertion and querying of token IDs, using the ID as the key for all k hash functions.

The caveat to Bloom filters is that you cannot explicitly remove a single key. Since multiple keys map to the same bits, if you flipped the bits associated with a single key you would also be removing an unknown number of other keys from the filter. This would create false negatives which is a constraint of the FastPass system. Another related caveat is the more keys that map to a filter, the greater the probability of false positives. Inserting all keys within an infinite window of time to the filter would not increase the memory needed, but would increase the number of false positives. Since keys cannot be removed from filters and we want to bound the probability of false positives, the filters must be periodically reset to keep false positives to a low rate.

Every time a filter is cleared, all of the token IDs that were represented by bits in it can be reused. Token reuse must be kept at a low enough rate to prevent distributed denial of service attacks through the use of tokens. Given that n tokens can be obtained by a set of malicious hosts, with token duplication detection, the maximum rate hosts can flood a bottleneck link near the end host is k/t packets per second.

A Bloom filter is considered optimal when approximately half of the bits are set and appear random. This allows additional bits to be set while maintaining low false positives. The number of hashes, k, and the size in bits of the filter, m, determine how optimal the filter is in terms of false positives. The optimal value for k that minimizes the probability of false positives, given m and n, is $k = ln2(\frac{m}{n})$. This is approximated by $(0.6185)^{\frac{m}{n}}$.

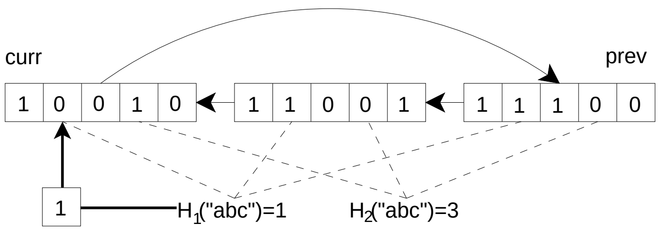

Figure 6: The circular Bloom filter illustrated has parameters of $m = 5$, $k = 2$, and $t = 3$. The key "abc" is used in computing two hashes which give the bit indexes of the key into the bloom filters. The two bit indexes must be queried in each filter, shown by the dotted lines, to check for the existance of the key in any of the filters. Since no individual filter has both bits set, the key does not exist in the circular Bloom filters, and is inserted by setting the bit of index 1 in the current filter to 1. The insertion is shown by the solid array coming from hash 1. The bit index computed by hash 2 need not be changed. After the current cycle period of an arbitrary amount of time, the *curr* Bloom filter becomes the *prev* Bloom filter, as illustrated by the arrows.

With the IXP2400's four gigabit links, and 10% of capacity being allocated to token requests, the router can process up to 32,000 tokens per second. Given a filter holds 1 second worth of token traffic: $n = 320000$. With an approximation for the optimal value of k where the probability of false positives is minimized, given a maximum false positive probability desired, a value of m can be determined. Using the approximation of the optimal value for k that minimizes the probability of false positives, and a desired maximum false positive probability of $\frac{1}{10^6}$, the optimal size of the Bloom filter becomes approximately 1.15 megabytes. The optimal number of hashes would then become: $k = ln2(\frac{9700000}{320000}) = 4$.

Wanting to keep a bigger windows worth of keys would not necessarily imply the use of a larger Bloom filter in the case of token duplication detection. It must be ensured that once a token is seen, it cannot be used again for approximately t seconds. Using a large Bloom filter cannot guarantee this. For instance, a single filter large enough to keep some false positive probability, but hold five seconds worth of tokens, can easily cycle out newly received tokens on filter clears. A token that arrives a microsecond before the filter clears can be reused twice within a small amount of time. A circular buffer of t bloom filters, one seconds worth of tokens each, can bound this.

Instead of a single Bloom filter, t circular Bloom filters can provide the same probability of false positives but ensure a more stable token duplication detection. This functionality is illustrated in figure 6. In the example, the parameters used are: $m = 5$, $t = 3$, and $k = 2$. The cycle period and the maximum number of tokens mapped to a filter are arbitrary in the example. With circular Bloom filters, insertions only go into the current filter, labeled as *curr*, whereas queries happen at all filters. After a cycle period, the circular Bloom filters rotate and a new current filter is selected. The bits in the new current filter are zeroed and then all insertions for the current cycle period go into this filter. This process repeats, removing keys that are approximately t seconds old, while ensuring that keys cannot be reused at extremely variable times. When using circular buffers, the false positive probability per packet becomes $t * rate$, or $\frac{t}{10^6}$ in the example.

A second advantage to using circular Bloom filters is the ability to access the SRAM on different channels for performance issues. The IXP2400 has four seperate SRAM channels to allow the splitting up of the circular Bloom filters. If one large Bloom filter was placed onto a single SRAM channel, all microengines would be accessing this same channel with k hash functions and could create a bottleneck for reading and writing to the Bloom filters. Since there are four seperate SRAM channels we will be using $t = 4$, which is four circular Bloom filters.

When the memory is allocated at boot time for the IXP, the *malloc* function is passed a parameter specifying which channel of SRAM to allocate the memory in. We will dynamically choose this channel based on the current filter we are allocating memory for. The first Bloom filter will be allocated in SRAM channel 1, the second Bloom filter will be allocated in SRAM channel 2, and so on.

3.4.2 Rate Policing

The work of Estan et al on sampling of packets and the use of multistage filters must be done on the data plane to keep up with line speeds. Given a probability of sampling a byte p, the probability of sampling a packet of size s becomes p_s which is equal to 1 - (1 - p)^s. This computation must be done on each packet which would place this functionality in the data plane to keep up with line speeds. The details of what is done with the packet once it is selected to be sampled are still being worked out as we read further into the document.

Monitoring must interact with the capability component of FastPass. When a capability is read, it stores the current fields in to general purpose registers. The monitoring component will be called after the capabilities have been verified and compute the probability of sampling the data. If the packet is going to be sampled, statistics are kept in DRAM about the hosts and rates. We have chosen to keep these statistics in DRAM since they are not accessed on every packet, and are accessed at a low probability. Furthermore, if rates are kept on all flows with capabilities, the size needed may be too large to store in SRAM.

The details of the data structures and full functionality of rate policing is still unknown. This is one of the research sides of this project, since there are many different rate policing policies that can be used, we hope to find at least two implementations for the IXP to use for our evaluation. This could compare different rate policing schemes to the amount of computations they require, the amount of memory they require, and their performance in estimating the hosts rates.

3.4.3 Management Statistics

The final component of the monitoring system is the management plane functionality. This functionality will provide the ability to connect to the IXP2400, receive reports of the hosts being monitoring with capabilities, the rates they using, and the estimated rates at which they are sending. This would not only provide functionality for a system administrator, but will also be helpful functionality to evaluate and test the accuracy of the system.

Through the use of the management level of the monitoring system, the estimated rates can be checked against actual known rates being sent by a host. This can be used to verify estimated rates calculated by the packet sampling and multistage filter.

The implementation of this aspect of the monitoring system is straight forward. At the boot time of the router, memory will be allocated in the SRAM for the counters. This will be done in two steps. The first step is to loop through the SRAM memory channels to find one with enough memory for the counter, this will be done with *ix_rm_mem_info* which returns the current channels information into a structure. The second step is actually allocating the memory with *ix_rm_mem_alloc*, which the current memory type (SRAM) and current channel number are passed to. We do not believe the memory needs to be patched since rate handling is handled by the XScale.

4 Evaluation

As this is a project aimed at demonstrating resilience to denial of service attacks, we would be able to best achieve optimum effect by utilizing a testbed containing more than one IXP2400 routers and more than two end hosts. We are uncertain yet as to our final number of machines, but we envisage a number that will help us effectively demonstrate our system's performance during a DDoS attack.

The primary goal is to evaluate whether the idea of FastPass translates to the same level of efficiency on hardware as it does in simulated routers. Ideally, FastPass should enable the network infrastructure to function in the face of DDoS attacks without sacrificing performance.

We aim to evaluate our system against a baseline architecture that does not have any resilience built in. Metrics we will use for comparison are end-to-end latency, packet loss, packet prioritization under various traffic patterns like light and heavy load, and DDoS attacks. End to End benchmarks will evaluate latency, packet loss, etc. Micro benchmarks will observe line speeds at each router, resource utilization like memory consumption etc. We can also determine the time for cryptographic computations and lookups, and the cost of monitoring rate detection and token duplication at each router.

Besides comparisons with the baseline architecture, we would also like to evaluate the performance of FastPass with different design configurations. We could compare different policing schemes and measure performance vis a vis resource utilization for the microengines and memory. Another interesting comparison, time permitting, would be that of public key versus symmetric cryptography.

We will also be able to use the end host implementations from the FastPass Click implementation to reduce the amount of complexity in the project for evaluating the system. The only implementation we need to take care of is that in the IXP2400 to be able to perform a full evaluation. The Click implementation can be run on end machines which will send data through our network.

5 Implementation

The team will build the system by each member taking one of the three major components and implementing it. Since the components interact with each other, we must first define a concrete API with a set of macros or functions with parameters and return values determined. This would allow progress to be made in each separate component, and assuming each component functions as designed and follows the API, they can be integrated seamlessly once complete.

We have broken up the project as follows:
Priya Sankaralingam - Capabilities
Gaurang Sardesai - Tokens
George Nychis - Monitoring

We will all be involved in the evaluation of the project as well. Since we have multiple potential evaluation scenarios, we will all work with each other during the evaluation phase.

We have broken down our project into these milestones:

Date	Milestone
10/06	Final implementation details decided upon
10/16	Capabilities implemented
10/19	**Status Report 1**
10/23	Tokens implemented
10/29	Capabilities and tokens debugged
10/30	Monitoring implemented
10/31	**Status Presentation**
11/05	Components fully integrated, code freeze for debugging of integrated components
11/15	Evaluation of the system begins
11/10	**Status Report 2**
11/30	**Project Demos**
12/05	Evaluation results must be finalized
12/07	**Final Presentation**
12/19	**Final Report Due**

The major risks associated with our project come from integrating the three components together properly. API's must be *strictly* defined such that each component provides the proper feedback to the component using it. If we implemented our components separately, then pull them together and each one gives different feedback than expected, our system will fail and we will likely not have time to fix it and perform as detailed of an evaluation as we would like.

Therefore the most crucial aspect is defining the macros and functions used by the microengines or XScale to integrate our components. I think each component can be completely reasonably, however it is the integration that is the major risk.

The other major risk comes in the debugging phases. Each component should be tested independently rather than being built, then only debugged when integrated with the other components. If debugging is held off until final integration, when bugs occur, finding which component is responsible will become difficult. Therefore proper debugging of each component independently must be done.

References

[1] D. Wendlandt, D. Andersen, and A. Perrig. *Bypassing Network Flooding Attacks using FastPass.*

[2] FastPass: *http://www.cs.cmu.edu/ dwendlan/fastpass/*

[3] C. Estan, S. Savage, and G. Varghese. *Automatically inferring patterns of resource consumption in network traffic.* Technical report CS2003-0746, UCSD.

[4] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood. *Deep Packet Inspection using Parallel Bloom Filters.* Technical report, WUSTL.

[5] M. Ramikrishna, E. Fu, and E. Bahcekapili. *A Performance Study of Hashing Functions for Hardware Applications.* In Proc. of Int. Conf. on Computing and Information, pages 1621-1636, 1994.

[6] A. Yaar, A. Perrig, and D. Song. *SIFF: A Stateless Internet Flow Filter to Mitigate DDoS Flooding Attacks.* In Proceedings of the IEEE Security and Privacy Symposium, May 2004.

[7] X. Yang, D. Wetherall, and T. Anderson. *A DoS-limiting network architecture.* In SIGCOMM, Aug. 2005.

[8] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. *The Click modular router.* ACM Transactions on Computer Systems, 18 (3):263297, Aug. 2000.