NetRPC: Enabling In-Network Computation in Remote Procedure Calls

Bohan Zhao, *Tsinghua University;* Wenfei Wu, *Peking University;*
Wei Xu, *Tsinghua Univesity*

https://www.usenix.org/conference/nsdi23/presentation/zhao-bohan

This paper is included in the
Proceedings of the 20th USENIX Symposium on
Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

Open access to the Proceedings of the
20th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

NetRPC: Enabling In-Network Computation in Remote Procedure Calls

Bohan Zhao
Tsinghua University

Wenfei Wu
Peking University

Wei Xu
Tsinghua University

Abstract

People have shown that in-network computation (INC) significantly boosts performance in many application scenarios include distributed training, MapReduce, agreement, and network monitoring. However, existing INC programming is unfriendly to the normal application developers, demanding tedious network engineering details like flow control, packet organization, chip-specific programming language, and ASIC architecture with many limitations. We propose a general INC-enabled RPC system, NetRPC. NetRPC provides a set of familiar and lightweight interfaces for software developers to describe an INC application using a traditional RPC programming model. NetRPC also proposes a general-purpose INC implementation together with a set of optimization techniques to guarantee the efficiency of various types of INC applications running on a shared INC data plane. We conduct extensive experiments on different types of applications on the real testbed. Results show that using only about 5% or even fewer human-written lines of code, NetRPC can achieve performance similar to the state-of-the-art INC solutions.

1 Introduction

The recent programmable switches like Barefoot Tofino [16] and Cisco Silicon One [5] can execute user-specified stateful packet processing at line rate. The evolution has sparked a surge of proposals to offload application functions into the network. The trend is called *in-network computation (INC)*.

INC has been widely applied in various applications including distributed ML training [11, 22, 29, 31, 37], cache [19, 25], agreement [6, 18, 40], and network monitoring [12, 17, 26, 27]. The tremendous bandwidth and low latency on switches lead to huge performance gains. For example, ATP [22] accelerates distributed training throughput by 38% $\sim$ 66%; P4xos [6] reduces the end-to-end delay by more than 90%; NetCache [19] improves throughput by 3-10 times compared with a host-only software solution. However, developing INC applications involves too much arcane knowledge in networking that is far from application programmers' (we refer to them as *users* in this paper) expertise and willingness to learn.

First, the INC program centers on individual packets. Users need to handle network functions such as packet parsing, flow table installation, forwarding, routing, reliable transmission, and congestion control as part of the application.

Second, users need to learn chip-specific languages like P4 [3] and NPL [28]. Even the high-level programming models like Lyra [7] and C3 [20] still focus on packet processing and require too much network knowledge (e.g., transmission windows and protocol fields) for software engineers.

Third, users need to understand low-level chip design details and limitations. Familiar data types and operations like floating points are missing, and users have to design approximations [13, 22, 31] manually. Even harder, users need to place their program on a pipeline of *stages* with isolated memory and deal with limitations, like once-only memory access per stage and a limited number of tables and rule entries.

Last but not least, users need to *statically* decide switch memory layout, table/register arrangement, etc., as the switch hardware can only modify them at boot time. Therefore, users need to reset the switch to start/remove an INC application, causing minute-level service interruption.

As a result, existing projects use INC only as a single application accelerator instead of a shared infrastructure. Even a simple application involves thousands of lines of code on both switches and hosts (Table 4 in Section 6). The development and operation difficulties prevent wide INC adoption.

In comparison, traditional software uses two abstraction layers to decouple application code from network details: 1) a *Socket layer* providing connection/session management, resource sharing, reliable communication, and byte stream abstraction; and 2) a *remote procedure call (RPC) layer* providing high-level data types and call interfaces. In the popular gRPC framework [10], users write a language-independent *interface definition language (IDL)* (e.g., `protobuf` [9]) specifying types of parameters and return values, and the gRPC compiler generates client and server *stubs* that users can integrate into application code. The stubs automatically marshal/un-

marshal arguments and handle underlying Socket connections. RPCs prove to be a powerful interface to build modern distributed systems. Unfortunately, neither layer exists in INC, leaving tedious network details to user applications.

We propose NetRPC to add both missing layers — an *INCLayer* and an *RPCLayer* — to bridge application programming and network packet processing, allowing users to leverage INC features to develop a diverse set of distributed applications using the familiar RPC interfaces.

The RPCLayer provides a high-level RPC interface. It is built on gRPC with two extensions: *INC-enabled data types (IEDTs)* and a NetFilter. IEDTs include basic types like integers and floating points, and collections like arrays and maps. Users define RPC services using the same protobuf language, just replacing vanilla gRPC types with IEDTs to allow NetRPC to recognize and process these data fields. In addition, users provide a NetFilter to specify the computation with INC, in terms of five *reliable INC primitives (RIPs)*. RIPs implement high-level operations on IEDTs such as arithmetics, reading/writing a map/array of arbitrary size, and synchronization primitives. RIPs also guarantee reliability, i.e., under various network conditions, RIPs eventually complete as long as the client/server processes survive.

RPCLayer also provides automatic data parallelism for calls with large arguments. NetRPC breaks up a call into subtasks, executes these subtasks concurrently, and sends out multiple *concurrent flows*. We offer it as a built-in feature to save programmers from handcrafting concurrent flows or co-flows to fully utilize the 100+ Gbps links in INC switches.

Analogous to the Socket, the INCLayer handles all flows from the RPCLayer. In addition to the basic guarantees of the Socket-like connection, reliable transport and congestion control, the INCLayer implements the RIPs using a set of protocols involving both the INC switches and the end-hosts.

We build NetRPC as a *general* INC-enabled RPC system. This is different from existing INC projects that only need to find one workaround for the switch hardware limitations as they target only a single application. The first design trade-off we need to make is between *generality (i.e., how programmable the network is)* and *simplicity (i.e., how easy it is to program it)*. Instead of building yet another general INC language, NetRPC chooses to provide only the necessary set of network-independent primitives and the simple NetFilter specification. Observing INC projects in the past ten years, we find only a handful successful types (Section 3.1). We design the primitives so that users can easily develop applications of all these types and enjoy the INC performance boosts.

New challenges for NetRPC include, from low level to high level: 1) efficiently managing the switch memory and pipeline stages to support the high-level array and map types; 2) hiding the switch hardware limitations from high-level programs; 3) supporting reliable transmission for different INC scenarios; 4) running multiple INC applications concurrently on a shared data plane; and 5) allowing users to define INC operations for their applications in the familiar gRPC abstraction.

We have many innovative designs to solve the above challenges. 1) Using a fallback mechanism, the end-host agents can take over all cases that the INC switches fail to handle; 2) Using an INC-compatible transport protocol, we can correctly handle packet retransmission and congestion control, maintaining both correctness and throughput; 3) Adapting a novel memory management scheme, we map from *keys* to unified 32-bit *logical addresses* that further map to switch *physical addresses*, allowing us to optimize the switch memory management much like normal caches; 4) By providing only a limited interface NetFilter, we abstract all obscure hardware limitations into a single high-level limitation (i.e., the primitives NetFilter supports).

We implement NetRPC using a testbed with two Barefoot Tofino [16] switches and eight machines. Using four non-trivial applications (Paxos, network monitoring, distributed training, and MapReduce) as examples, we show that 1) we reduce the *line of code* (LoC) on the end host to about 1/20, using less than two dozen network-related LoC per application; 2) NetRPC code is completely the same as vanilla gRPC code; and 3) we can offer the same or even better INC speedup.

In summary, our contributions include:

1) As a programming interface, NetRPC is the first framework to integrate INC acceleration into the RPC framework, reducing the bar of INC adoption in software.

2) As an INC system, NetRPC proposes a set of INC primitives applicable to different INC application types and innovative design elements to efficiently implement them, including reliable transport, memory management, and synchronization, as well as enabling a multi-application INC data plane.

3) Using four common INC application types on a real testbed, we demonstrate that we can offer the same INC performance boost with far fewer lines of code.

2 Related Work

Most existing INC applications make a network-software co-design. Even with the "network programming languages", users still have to handle many network engineering details.

Network-software co-design of INC. People have recently demonstrated many promising INC-accelerated applications, such as NetCache [19] and distCache [25] for caching, P4xos [6], NetChain [18] and NetLock [40] for agreement, SwitchML [31], SHARP [11], and ATP [22] for distributed ML training, and ElasticSketch [38], SilkRoad [26] and Sonata [12] for network monitoring. These solutions are similarly constructed as the network-software co-design — user interfaces, customized protocols, switch programs, rule installation, and endpoint agents — to achieve full-stack optimization and higher switch resource efficiency.

Chip-specific Programming Languages. People have proposed several chip-specific programming languages [3, 28,

32, 33] to support data plane customization. Existing programming languages are tightly coupled with corresponding ASICs. For example, Trident-4 [28] only supports NPL, while P4 programs can run on Tofino and Silicon One. P4 [3], arguably the most popular one for recent INC solutions, follows a *reconfigurable match table* (RMT) architecture. P4 programs first define packet headers and corresponding parsers and then process extracted header fields in a pipeline. Programmers must specify the actions on header fields, persistent switch registers at each pipeline stage, and drive actions by match-action tables. Also, users must define a *deparser* to reconstruct the packet for forwarding.

High-level network programming abstractions. There have been efforts to simplify the INC programming. E.g., Lyra [7] offers a one-big-pipeline abstraction that allows programmers to express their intent with simple statements; NCL [20] imports a window-based abstraction over packets as the basic processing units. μP4 [34] provides a lightweight logical architecture that abstracts away the structure of the underlying hardware pipelines for better program composition. NetVRM [42] allows developers to virtualize switch memory with a few modifications to existing P4 code. Chipmunk [8] adopts a domain-specific program synthesis technique to generate faster packet-processing code at the cost of longer compilation time. However, these high-level abstractions still revolve around networking details, such as (de)packetization, connection maintenance, and protocol stacks. The semantic gap between the software and network programming model is still a significant obstacle for ordinary software developers.

3 Design Overview

We design NetRPC to allow software developers to enjoy the performance benefits of INC without tedious network programming. We want NetRPC to be general enough to support typical INC application scenarios.

3.1 INC Application Types

INC accelerates applications primarily in two ways: optimizing bandwidth usage (reducing the *number of bytes* to servers) or reducing latency (removing the server from the round trip). People have proposed many INC applications. Table 3.1 summarizes the four types of applications.

The first two types handle large data sets with optimizing bandwidth as the main goal: (1) synchronous aggregation (SyncAtgr) for distributed machine learning (ML) training; (2) asynchronous aggregation (AsyncAtgr) for general MapReduce-type applications. The difference between these two types is that SyncAtgr aggregates only a fixed-sized array (e.g., the gradient updates) and works in iterations, i.e., we can proceed only *after* all clients send the updates. In contrast, AsyncAtgr aggregates over an arbitrary number of keys as they come in and allows accessing results at any time.

The other two types only use small data, with the main goal to optimize latency by avoiding sending packets to the server: (3) key-value cache (KeyValue) that require frequent queries and responses; and (4) Voting (Agreement) that involve counting votes from different clients until reaching a threshold. Unlike (1) and (2), each request is small, but the challenge is how to achieve a latency smaller than client-to-server RTT by not involving the server at all.

3.2 Challenges and Solution Overview

Providing a reliable data stream for general INC application types. Different from traditional networks, there are *side effects* when packets go through an INC switch, such as updating a map. Thus, when a packet goes through a switch twice in retransmission, the computation is no longer *idempotent*, violating the computation correctness. Prior solutions are application-specific, e.g., ATP [22] requires explicit server ACKs. It works in SyncAtgr, but not in the other three types because involving the server defeats latency optimization. We design an efficient and general retransmission mechanism that maintains the per-flow state on the switch using only a few bits in switch memory. We also design an effective flow and congestion control protocol (Section 5.1).

Making "normal path" efficient: Supporting memory-efficient arrays and maps on INC switches. Arrays and maps are core data structures in many applications, and INC significantly accelerates operations on them with parallel element processing. E.g., training applications use arrays to store the aggregated gradients, and monitoring applications keep the aggregates in a map, one key per metric. In both cases, the switch can add up all values in parallel. Existing systems either require pre-determined encoding of keys (e.g., knowing all the keys at compile time) or waste precious switch memory and packet header space to store the long keys. We leverage the host agents to generate a *two-level mapping* from keys of arbitrary lengths to a unified 32-bit *logical address space* and then map it to the switch physical memory. We also design a cache management algorithm running on the server agent to improve switch memory utilization efficiency (Section 5.2.2).

Making "corner cases" correct: Hiding switch hardware limitations from the upper-level program. We still need to handle switch hardware limitations. Our key idea is to use all *host agents* as a fallback mechanism. The host agents emulate all switch operations in software and thus can always provide correct INC results to the `RPCLayer` regardless of the switch's ability or resource. NetRPC supports two kinds of fallbacks: 1) arithmetic overflows that may happen in floating-point computation and accumulations (Section 5.2.1); and 2) insufficient memory on the switch (Section 5.2.2).

Supporting multi-application data plane. Prior arts support only a single application, and the life span of the switch program does not exceed that of the application. How-

Table 1: Four Common INC Application Scenarios and Primitives They Need

Type	Applications and Existing Systems	IEDT	Primitives
SyncAgtr	Distributed ML training (ATP [22], SHARP [11], SwitchML [31])	Array	`Map.get`, `Map.addTo`, `Map.clear`, `CntFwd`
AsyncAgtr	MapReduce (ASK [2], NetAccel [23], Cheetah [36])	Map	`Map.get`, `Map.addTo`, `Stream.modify`
KeyValue	Cache (NetCache [19], DistCache [25]), Monitoring (ElasticSketch [38])	Map	`Map.get`, `Map.addTo`
Agreement	Synchronization (P4xos [6], NetChain [18], NetLock [40])	Integer	`Map.get`, `Map.addTo`, `Map.clear`, `CntFwd`

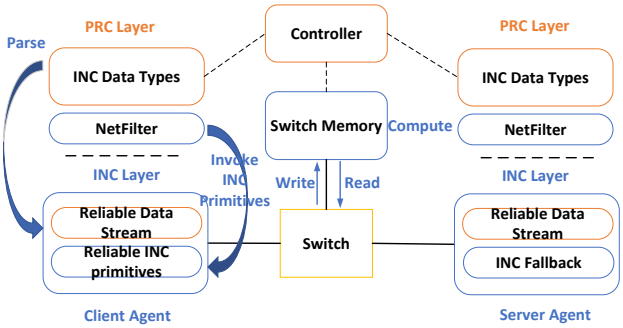


Figure 1: NetRPC system architecture.

```
1  import "netrpc.proto"
2  message NewGrad {
3    netrpc.FPArray tensor = 1;
4  }
5  message AgtrGrad {
6    netrpc.FPArray tensor = 1;
7  }
8  service Training {
9    rpc Update(NewGrad) returns (AgtrGrad)
         {} filter "agtr.nf"
10 }
```

Figure 2: Example `protobuf`: gradient updates

```
1  { //agtr.nf
2    "AppName": "DT-1",
3    "Precision": 8,
4    "get": "AgtrGrad.tensor",
5    "addTo": "NewGrad.tensor",
6    "clear": "copy",
7    "modify": "nop",
8    "CntFwd": {
9      "to": "ALL",
10     "threshold": 2,
11     "key": "ClientID",
12   },
13 }
```

Figure 3: Example `NetFilter`: gradient updates

ever, the RPC servers are long-running daemons, and server start/stop/restart events are common. It is prohibitively expensive to reset the switch on such events. We solve the problem with three designs: 1) letting all applications share the same set of RIPs; 2) sharing the same set of switch memory blocks among applications by partitioning the key spaces among them; 3) providing three choices of memory eviction behaviors to fit different applications (Section 5.2.2).

Interface `INCLayer` primitives with `RPCLayer` without breaking `protobuf` abstraction. Users need to tell NetRPC *what* to process in INC and *how* to process them. We need to add the INC specification to `protobuf` language, but we decide *not* to change the language to keep the learning curve low for users. Thus, we design the `NetFilter` as a configuration instead of a program. We only allow users to specify a fixed set of RIPs with at most one instance for each kind as a filter to process arguments and return values. The limitation simplifies NetRPC design yet still allows implementing all four common types of INC applications (Section 4).

The NetRPC contains a controller, host agents, and switch programs as in Figure 1. The system-wide *controller* is a dedicated process that handles registration and name lookups at initialization, while at runtime, it manages configurations on both switches and host agents. The *host agents* run on each client/server. Each host agent maintains a fixed number of connections (configurable) with the switch, even without running tasks. These connections are essential for the reliable communication (Section 5.1). A single *switch program* starts each INC switch at boot time and executes all primitives. The switch receives configurations from the controller to run applications without resetting the switch program (to avoid interrupting the network). If the switch fails to handle a primitive due to resource or functionality limitations, the primitive execution falls back to the server agents.

4 RPC Layer in NetRPC

In this section, we first introduce the NetRPC programming interface using gradient aggregation in the distributed training application as a concrete example. Then we briefly introduce interface implementation in the `RPCLayer`.

`protobuf` definition. Like in vanilla gRPC, users first provide a `protobuf` definition that compiles into the client and server stubs. Figure 2 shows an example `protobuf` file. The `messages` are user-defined types, and `service` is the RPC definition using `messages` as arguments and return values. The only modification to vanilla `protobuf` is the `filter` clause allowing users to provide the `NetFilter` file name (see below).

NetRPC data types. Users declare all variables that they want to process in INC using *INC-enabled data types (IEDTs)* defined by NetRPC. E.g., line 3 and 5 in Figure 2 defines variables (both `tensor`) as a `netrpc.FPArray` (floating point

```
shared_ptr<Channel> channel =
    CreateCustomChannel(server_ip,
    InsecureChannelCredentials());
unique_ptr<Stub> stub_(NewStub(channel));
void PushPull(double* data, int length) {
  NewGrad request;
  AgtrGrad reply;
  ClientContext context;
  request.mutable_tensor()->mutable_data()
        ->Add(data, data+length);
  Status status = stub_
        ->Update(&context, request, &reply);
  memcpy(data, reply.tensor().data(),
        length * sizeof(double))
  train(data);
}
```

Figure 4: Client program to use the RPC

Table 2: NetRPC Primitive Semantics

Primitive	Args	Semantics
`Map.addTo`	stream	map[stream.key] += stream.value
`Map.get`	stream	stream.value = map[stream.key]
`Map.clear`	empty	map[stream.key] = 0
`Stream.modify`	op,para	stream.value = op(stream.value, para)
`CntFwd`	key,th,tgt	cnt[key]++; if cnt[key] == th then forward(tgt) else drop

array) IEDT. Optionally, user can add normal gRPC data fields to the same messages, and NetRPC simply passes them to the server without processing.

Collections (`Array` and `Map`) are core data types in NetRPC. The item value can be integers or floating points, and keys can be integers or strings. NetRPC enables 1) automatically applying the user-defined `NetFilter` on *every* value in these collections and 2) accessing the *global INC map* using keys.

Life of a NetRPC call. In NetRPC, when a client initiates a call, the *client stub* marshals the arguments and sends them through one of two channels: messages with IEDT through the INC channel established by the per-host *client agent* and normal messages through the original gRPC Socket. In this paper, we only focus on the *data streams* in the INC channel. The underlying `INCLayer` processes the data stream and optionally interacts with the *INC map*. The INC map is a NetRPC abstraction of unlimited global memory addressable using keys or array indices. INC map is implemented on both switches and host agents (in Section 5.2.2). The return path is similar: the *server stub* marshals the return value and sends it through either the INC channel or the normal Socket.

The `NetFilter` and reliable INC primitives (RIPs). In addition, users need to specify their INC operations. Here, we have a choice in terms of what kind of operations NetRPC should provide. We want to find the sweet spot in the trade-off between generality and simplicity. We also want to provide a reconfigurable switch program to serve new applications. Therefore, we pick five primitives that we can compose together in a similar layout to implement existing types of INC operations (Section 3.1). Figure 5 displays this layout and its implementation on the switch. The users only need to provide configurations for these five primitives in their `NetFilter` file (Figure 3) to specify their INC operation of interest.

The `NetFilter` is a JSON configuration file. It contains a `AppName` that uniquely identifies an application, a `Precision` field that specifies the floating-point precision (number of digits after the decimal point). Lower precision allows INC to process more data without falling back to the host.

The more interesting part in `NetFilter` is the next five fields that allow users to provide arguments to RIPs, including three map-access primitives, `Map.addTo`, `Map.get`, and `Map.clear`, one data stream manipulation primitive, `Stream.modify`, and one synchronization primitive, `CntFwd`. Table 2 summarizes the parameters and semantics of these primitives.

`Map.addTo` *accumulates* data items from the stream to the map according to their keys/indices, and `Map.get` reads out the values of a specific key from the map. In Figure 3, we add the values of the `NewGrad.tensor` array to the INC map to aggregate the gradient, and on the return path, we read out the results from the INC map into the `AgtrGrad.tensor` array.

`Map.clear` defines how to clear a value from the INC map. In the example, `copy` means backing up the aggregates to the server before clearing it out to handle packet losses. We introduce other possible options in Section 5.2.2.

`Stream.modify` performs arithmetics on the stream. It only modifies the stream without accessing the INC map. In Figure 3, we set it to `nop`, as we do not modify streams. Table 8 in Appendix A lists all operations we support for `Stream.modify`.

The `CntFwd` is the most interesting primitive. It accumulates values on one or more keys (specified with `CntFwd.key`) in the INC map until the accumulator reaches the specified threshold (`CntFwd.threshold`). Then it forwards out the message to the destination(s) specified at `CntFwd.to`. The `CntFwd` primitive is essential to control both *how many* packets to forward to the clients/servers and *when* to forward them, and thus essential for SyncAgtr and Agreement applications. In this example, we set `key` to a single `ClientID`, meaning that we only need one counter for the number of unique clients who have sent gradient updates. In this case, only when exactly two unique clients have sent a stream, will the network aggregate the items and send back `AgtrGrad` to `ALL` clients.

There are other use cases for `CntFwd`. Setting the `CntFwd.threshold` to one makes the `CntFwd` behave as the `test&set` primitive in many instructions sets, useful to implement distributed mutual exclusion. Also, by providing a collection in the data stream, we can use a map of counters to track multiple votes in concurrent ballots, a widely-used functionality in distributed agreement protocols. `CntFwd` allows the switch to notify the clients only when enough votes arrive. Appendix D provides more examples of `CntFwd` primitive.

Table 1 summarizes the primitives used in each INC application type. Figure 5 illustrates a RIP pipeline running the example code in Figure 3. A SyncAgtr application pushes

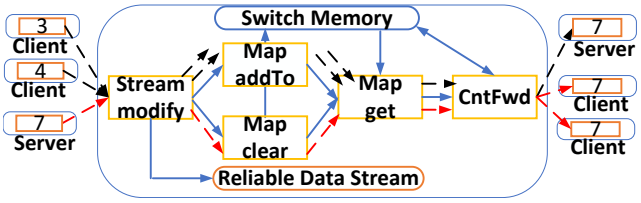


Figure 5: RIP pipeline in switches.

data into the network by its clients (black arrows) for on-switch aggregation and on-server backup. The server sends back the computation results (red arrows) to clients and clears the switch memory. The same switch program (and the RIP pipeline) completes all INC functions in the round trip without reconfiguring the switch.

Using RPC calls to build the application. With the `protobuf` and `NetFilter` definition, the remaining process is exactly the same as normal gRPC. The `protobuf` compiler generates client and server stubs, and users include stubs in their application. Figure 4 provides the client code using the RPC service defined in Figure 2. Note that the code is completely identical to vanilla gRPC, hiding INC details from the users.

Automatic data parallelism on RPCs with large arguments. There could be multiple concurrent NetRPC applications/channels and procedure calls in the runtime. The stub submits calls as tasks to the host agents. A task contains the application data (i.e., arguments or returned values, encoded as protobuf messages), the metadata (e.g., network program configurations), and the routing information.

The host agents maintain a thread pool of worker threads to process tasks. NetRPC automatically partitions the task into subtasks and dispatches them onto multiple worker threads for load balance. The worker threads serve the subtasks in their queue on a First-Come-First-Serve (FCFS) basis. The worker threads serialize the subtask's data into a sequence of packets (Appendix B.1) and send them over the user-level network stack we implement using DPDK [15].

Limitation of RIP abstraction. RIP abstraction targets simplifying programming in general INC scenarios where different INC applications regularly start and stop to share the infrastructure. It lacks logical semantics like looping and branching and thus can not implement complex algorithms (e.g., DHS [41]) or data structure (e.g., NetChain [18] queue). Adding more RIPs will extend the functionality but reduce the source available for each RIP. We leave further extensions and a customizable set of RIPs as future work.

5 INC Layer in NetRPC

`INCLayer` provides a reliable layer to support RIPs. There are two design objectives: 1) efficiently utilize INC switch resources to support a multi-application data plane with full INC performance boost; 2) provide an end-host software-based fallback mechanism to support reliable byte streams and INC primitives for the `RPCLayer`. As a result, `RPCLayer` can safely assume that the data stream is delivered reliably, and the `NetFilter` is fully executed in various network conditions.

In this section, we first introduce how we can build a general reliable data stream abstraction. Then we introduce the essential RIPs, including map access, arithmetics, and `CntFwd`. Finally, we briefly introduce the switch implementation.

5.1 Reliable Data Stream

Encoding IEDTs into sequences of packets. Client agents receive data streams containing multiple key-value pairs from the `RPCLayer`. Then the client agent encodes these pairs into separate packet headers using a user-level networking stack written in DPDK [15] and sends them out. Each packet contains a fixed number of key-value pairs (32 in the current setting), a sequence number, a *global application ID* (GAID), as well as other state information we will introduce in this section. Figure 14 in Appendix B.1 illustrates the packet header structure. The packet can optionally contain the normal payload with non-INC types for the application. NetRPC only processes the key-values encoded in the header.

Idempotent packet retransmission. In case of packet loss, traditional transport simply retransmits the lost packet. However, INC complicates the retransmission because the retransmitted packet can cause *side effects* on the switch, such as incrementing a map value again. In other words, naive retransmission is not *idempotent* and may lead to incorrect INC results. Switches need extra information to detect which packets are retransmitted. Traditional networking doctrine tells us that we shall avoid keeping states on switches. Thus, some INC designs choose to keep extra states on the sever [22] and let the server to ACK each packet. The ACK informs the switches about processed packets. This design requires the server to ACK every packet. It works on applications like gradient aggregation, where the INC is primarily used for reducing *server bandwidth* by only forwarding the results to the server to ACK while completing aggregation on the switch only. However, this design does not fit applications like KeyValue and Agreement, as server ACKs defeat the purpose of latency reduction via sub-RTT switch response.

To design a general protocol, we observe that 1) we send/receive all INC flows using host-agents under NetRPC control; 2) the INC switches have a relatively large memory, and the retransmission states are almost negligible compared to the INC map. Thus, we can safely keep the per-flow state on the switch as long as we can limit the number of agent flows.

We further design our protocol to minimize per-flow switch memory usage, allowing a switch to only keep a *bit array* of size w_{max} per flow, where w_{max} is the max sending window size. The switch initializes all bits to 1. Every packet contains a sequence number (`seq`), and a `flip` bit that is set to

$(seq/w_{max})\%2$. On receiving the packet, the switch checks the $(seq\%w_{max})$-th bit in the bit array. If the bit is the same as the `flip` in the packet header, the switch considers it as a retransmitted packet and thus skips updating the INC map. Otherwise, the packet is new, and the switch sets its corresponding bit to the `flip` and processes the packet normally.

We show that this simple protocol guarantees idempotent execution, i.e., 1) a packet's first appearance flips the bit, and 2) a packet's later appearance (retransmission) equals the bit. We prove it by *induction*. For sending window 0, all `flip` bits in packets are 0s, and the switch bitmap is all 1s. Since each packet only sets the bitmap to 0 once, at the end of window 0, all bits become 0. Then assuming the two properties hold for window $t-1$, we show that they still hold for window t. Recall that the client agent sends out the i-th packet in window t (denoted as P) only after the i-th packet in window $t-1$ (denoted as P') is ACKed. Therefore, when P first appears, P' should already set the bit as P''s `flip`. As P and P' are opposite in `flip`, P's first appearance flips the bit. P's later appearance would not flip the bit, and the window controls packets out of the window not to appear (and falsely flip the bit) between P's appearances. Thus, the two properties hold in window t. By induction, it is correct for all sending windows.

We can use $N \times w_{max}$ bits of switch memory to support N concurrent reliable flows on each agent. We experimentally set $w_{max} = 256$ and find it sufficient to achieve a per-flow bandwidth of 20+ Gpbs.

Flow control and congestion control. Note that w_{max} is a fixed value. We still need to deal with the flow control and congestion control due to resource contentions on either the end hosts or switches. We use the same mechanism to handle both flow and congestion control by automatically setting a congestion window $cw \leq w_{max}$.

Traditional congestion control, like the one in TCP, relies on round-trip-time (RTT) and duplicate ACKs to adjust cw. However, in INC primitives like `CntFwd`, these signals may not reflect the real network congestion as the receiver needs to wait for the slowest sender before ACKing.

Thus, NetRPC adopts an ECN (i.e., explicit congestion notification)-based congestion control mechanism. The switches set the ECN when the ingress port length exceeds the threshold. Meanwhile, it writes the ECN information to the INC map under a special key. Thus, all retransmission packets carry ECN until cleared like other map values. This prevents ECN signal loss due to packet loss. Otherwise, the client agents adjust the cw using the same *additive increase multiplicative decrease (AIMD)* policy as priot arts [22]. Experiments show that this design allows multiple flows to achieve both high goodput and fairness in bandwidth sharing.

Other transport protocols. Several recent transport protocols have affected the design of `INCLayer`. MTP [35] proposes a message-based protocol to customize congestion control, load balancing and resource isolation for INC. However, it requires maintaining per-pathlet states on both packet headers and end hosts, importing extra overhead to hurt the system's performance. DCTCP [1] imports a more fine-grained congestion window adjustment based on the ECN proportion. This approach is inapplicable to INC scenarios because we have to count the maximum number of ECNs in a single (i.e., the most congested) path instead of the total ECN proportion due to incast. This consumes more resources on the switch and will reduce the stream goodput, so we utilize AIMD to simplify implementation and will extend the protocol in future work (Section 7).

5.2 Reliable INC Primitive Designs

5.2.1 Computation and arithmetic overflows

Floating point arithmetic by quantization. INC switches only have limited 32-bit arithmetic functionality, yet INC applications like training require floating-point (FP) arithmetic. The standard practice uses quantization to fit the FP numbers into 32-bit integers (aka, fixed-point numbers). NetRPC quantizes an FP value in the client agent by multiplying it with a scaling factor (the `precision` field in `NetFilter`) and maps the value back to FP before handing it to the `RPCLayer`.

Handling overflows. While people have shown that the precision loss might not cause problems in many applications, 32-bit fixed-point numbers do not offer enough *representable range* in many cases, and thus overflow is unavoidable. Even without FP numbers, just using the `Map.addTo` to accumulate values may also lead to overflow. Thus, we need a way to handle occasional overflows.

When the switch detects an overflow during computation, it sets the overflowed value to `MAX_INT` or `MIN_INT` and forwards the packet normally. When a host agent receives a packet with `MAX_INT`/`MIN_INT` value, it suspects there is an overflow [1] and gives up the result. Then the client agents mark and resend these overflow packets, causing the switch to skip the processing and directly forward them to the server agent. The server agent computes the correct result using 64-bit integers or FP numbers in software.

Fallback on network fabrics without INC support. A similar fallback mechanism works when there are no programmable switches or data-plane resources reach capacity. If the controller fails to assign the INC application to any switch, the server agent will execute RIPs in software using the same switch failure handling mechanism. Therefore, the application is guaranteed to derive the correct results with the transparent fallback, only losing performance benefits from INC.

[1] Strictly speaking, there is one possibility of a false positive where the result is exactly `MAX_INT`/`MIN_INT`. The false positive only slightly affects performance leading to an extra retry, but not correctness.

5.2.2 Memory: INC map-access primitives

Memory address spaces in `INCLayer`. The `RPCLayer` supports maps with arbitrary keys, while the `INCLayer` only provides a 32-bit *logical* address space per application. The client agent hashes keys with different types and lengths into the 32-bit address space. We handle all collisions by putting the colliding keys into the payload to bypass the switch INC and let the server agent to process them. We choose not to use a larger logical space as we find it sufficient to support multiple applications with acceptable collisions. A short address saves bits in packets, increasing the effective bandwidth.

`INCLayer` maps the 32-bit address space onto the *physical* address space on switches. Each physical address corresponds to a *register* on a switch. Switches may have different numbers of registers. E.g., the switch we use has about 160K registers available per pipeline stage, and we use eight stages to support map-related primitives.

It is not trivial to map the logical address to a physical switch address. The above hashing approach does not work here because switch registers are a valuable resource we want to make full use of, but when the utilization is high, the collision rate increases fast, causing many fallbacks to servers. In fact, we need to pack the physical memory tightly. Also, we want to avoid keeping the logical-physical address mapping on switches; otherwise, it wastes switch memory.

In some applications, such as distributed training (as in ATP [22]) in SyncAgtr, it is simple as every client has the same set of keys. Each of them only needs to sort the keys and give each key a sequence number. However, it does not work in general cases, such as AsyncAtgr, where each client might have a different set of keys.

Multiple clients of a single application. We solve the problem by letting the server agent, shared by multiple clients, decide and maintain the mapping for all its clients. The first time a client uses a new logical address, it sends packets to the server without INC. If there is switch memory available, the server agent will piggyback a mapping for this address on the returning ACK. Then the client can send subsequent packets with the physical address set in the packet for the switch. If the switch memory is full, the server will not return the mapping, and thus clients keep sending subsequent packets to the server without INC. With the method, we ensure all clients calling the same server use a consistent mapping.

Handling multiple applications. According to the applications' requests, the controller reserves switch memory at application registration time. When an application gets no switch memory, they fall back to using server agents. We use a simple FCFS policy for the static allocation among different applications and leave advanced memory scheduling as future work. Note that although the controller reserves memory at registration time, the actual allocation only happens when the clients plan to send out data streams. Thus we can avoid holding memory unnecessarily.

Cache replacement policies. The switch memory serves as a cache for certain keys, and we need a replacement policy at the server agent. We take an approximation to the *least-recently-used* (LRU) policy. Each client agent counts the uses of each logical address within a *cache update window*, and at the end of the window, they send the counter to the server, allowing the server to compute the most-used keys in the last window. Then in the next period, the server evicts less used values. We also evaluate other popular cache replacement policies in Section 6, and we show that this periodic counting-based LRU policy works well.

Optimization for synchronous aggregation. In addition to the general logical-physical mapping, we realize that the SyncAgtr (i.e., distributed training) applications like SwitchML [31] only require access to large continuous arrays. It is more memory-efficient to be able to allocate such arrays in a few *circular buffers* instead of many individual addresses. NetRPC supports such buffers of a fixed size of 256 keys.

Preventing switch memory leaks on host failures. Unlike existing INC designs that serve only a single application, NetRPC is a shared infrastructure supporting many applications. Thus, we need to take care of potential switch memory leaks resulting from the crashing of user programs or host machines before they can explicitly release the memory. We address this issue with a *two-level timeout* mechanism.

NetRPC processes a packet with an *admission rule* that checks the GAID. We keep a timestamp of the last time the rule runs for each GAID. The controller periodically polls the switch for these timestamps. If it finds a stale timestamp, it triggers the *first-level timeout* by notifying the server agent to retrieve the application's INC map. After a longer period, the server agent triggers the *second-level timeout*, sending the saved data items to the user-defined stub or deleting them if the stub no longer exists. As switch memory is small and precious, we want to reclaim it quickly with a small *first-level timeout*. However, the small timeout unavoidably introduces false positives, hurting the correctness of programs with low communication frequency, such as monitoring infrequent events. In fact, these applications will benefit little from INC anyways, and the timeout mechanism allows them to run just like normal applications. Servers have much larger memory and thus can keep user maps longer, providing the correctness of such programs similar to software.

The `Map.clear` primitive. The switch memory only supports `Map.addTo` instead of directly overwriting the value. Thus, to start a new accumulation (e.g., a new iteration of training, restarting a vote, etc.), the user program needs to execute three steps: 1) `Map.get` the accumulator value to the hosts, and 2) `Map.clear` the memory and 3) start to `Map.addTo` new values. However, there is a risk that the packets get dropped *en-routing* to the host. In this case, the memory is already cleared, so the value is permanently lost.

NetRPC provides different methods to prevent this loss, as

there is a latency-throughput tradeoff. We decide to allow users to choose from three clear policies in `NetFilter`.

1) [Copy]: The client-call stream first carries the map's value to the server, and then the return stream from the server will `Map.get` and `Map.clear` the values. Thus we guarantee the server has a backup in case the return packet is lost. This policy requires no extra switch memory at the cost of forwarding more data to the server and thus higher latency.

2) [Shadow]: The switches double memory allocation. The data stream uses two memory segments alternatively: `Map.get` from one and `Map.clear` the other. This approach reduces latency at the cost of doubling memory usage and thus is only suitable for latency-sensitive applications with few data items.

3) [Lazy]: The `Map.clear` primitive only lets the host agents to save the current value and let the switch to keep accumulating without clearing. The host agent subtracts the saved value to compute the accumulated value since the last clear. When the accumulator eventually overflows, we fall back to the server agent using the same overflow logic and clear the switch memory. If the application (e.g., voting) has a slow-increasing counter, lazy policy involves little overhead.

The multiple `clear` policies allow users to better customize their INC applications according to their SLA requirements and workload features. We compare the performance of the three policies in Section 6.

Implementation on the switch. We allow 32 key-value pairs per packet. We use four register groups per stage and 8 out of the 12 stages on the switch to implement the INC map access. This design fits the switch hardware limitation: a packet can only access each group of registers in the switch once per trip. For the same reason, we arrange `Map.get`/`Map.addTo` and `Map.clear` to execute in the opposite direction of a packet round trip. These primitives are organized in a flow chart on the switch pipeline (Figure 15 in Appendix C). Appendix D displays a number of example settings of `NetFilter` in different application types.

5.2.3 Forwarding: the CntFwd primitive

The `CntFwd` primitive requires two extra pieces of logic in the switch. First, the switch needs to recognize the packet is a `CntFwd` packet, and then the packet goes through the normal map-access pipeline to increase and read the values in the accumulator. We implement different computation logic for the accumulator (`test&set` or accumulate) by applying different match-action tables according to the `CntFwd.threshold`. Finally the packet enters the last stage on the switch that decides whether to `drop`, `send`, or `multicast` the packet.

6 Evaluation

In this section, we show that NetRPC achieves the following desirable properties: 1) NetRPC supports four kinds of INC applications; 2) NetRPC significantly reduces the amount of application code; 3) NetRPC achieves the same performance as handcrafted INC applications; 4) NetRPC handles situations like packet loss, congestion, etc. In addition, we evaluate the effects of policy settings (clear and caching).

Table 3: Workload and Baseline in Experiments

App Type	App	INC Baselines	Dataset
SyncAgtr	Distributed Training	ATP [22] SwitchML [31]	ImageNet [14]
AsyncAgtr	WordCount	ASK [2]	Yelp [39]
KeyValue	Network Monitoring	ElasticSketch [38]	CAIDA Anonymized Internet Trace [4]
Agreement	Paxos	P4xos [6]	Synthatic workload

6.1 Experiment Settings

NetRPC implementation. We implement NetRPC switch logic on a 12-stage programmable switch. The NetRPC switch pipeline contains 32 read-write memory segments corresponding to the 32 key-value pairs in the NetRPC packet. Each memory segment contains 40k 32-bit units to restore INC states or the INC map. Depending on the service configuration, we vary packet lengths from 192 to 320 bytes.

NetRPC includes four modules: $\sim 4K$ lines of P4 code for the switch logic, $\sim 2K$ lines of Python code for the remote controller, $\sim 2K$ lines of C++ code as the plugin of gRPC++ [10], and $\sim 3K$ lines of C++ code for the NetRPC end-host agents using DPDK. We also implement four types of INC applications with only 200 $\sim$ 500 lines of code each.

Testbed. We run NetRPC on a testbed of 8 GPU machines and two programmable switches. The devices form a dumbbell topology: two connected switches, each with four machines. In the experiment, we use "X-to-Y" to denote a topology with X clients and Y servers. The switch contains a Barefoot Tofino chip and provides 32 $\times$ 100 Gbps ports. Each machine has a Mellanox ConnectX-5 dual-port 100 Gbps NIC. Each machine is equipped with two NVIDIA GeForce RTX 2080Ti GPUs, 56 CPU cores at 2.20GHz, and 192GB RAM. The machines install NVIDIA driver 430.34, CUDA 10.0, Mellanox driver OFED 4.7-1.0.0.1, and Ubuntu 18.04.

Workloads and baselines. Table 3 shows the workloads and baselines we use. We run various typical models (VGG, ResNet, AlexNet) for SyncAgtr. We also implement each application's pure software version as baselines using DPDK.

6.2 Reducing User Code Complexity

We compare the user-written lines of code (LoC) of NetRPC applications with existing INC arts. Table 4 shows that NetRPC reduces the overall human-written code by over 97% in all four application types. To enable INC in an RPC, the application developers only need to configure the `NetFilter` to enable/disable RIPs on the switch without writing any switch

Table 4: LoC Comparisons: NetRPC vs. Prior INC Arts

	NetRPC		Prior INC Arts	
	Endhost	Switch	Endhost	Switch
SyncAggr	173	13	3394	5329
AsyncAggr	166	26	3278	4258
KeyValue	162	26	898	2360
Agreement	1453	26	5441	931

Table 5: Microbenchmark on Basic INC Functions

Metrics	NetRPC	Prior Arts	DPDK
SyncAgtr Goodput(Gbps)	50.55	46.44 (ATP)	40.11
AsyncAgtr Goodput(Gbps)	72.31	73.96 (ASK)	45.88
Voting Delay(μs)	20	22 (P4xos)	92
Monitor Delay(ms)	3.52	3.26 (ElasticSketch)	4.05
Packet Processing Capacity(Mpps)	>1000	>1000	83.47

code. `NetFilter` results in a huge LoC reduction (12-21 LoCs in NetRPC v.s. 931-5329 in prior arts). On the host, NetRPC also reduces the LoC of host programs by 95%, 95%, 73%, and 82% for the four applications compared with existing INC applications, as NetRPC users only write code to process data-stream as call arguments, avoiding the tedious network functions like (de)packetization, reliability, etc.

6.3 End-to-end Application Performance

Distributed ML training. We set up eight worker machines for this evaluation. We use two existing INC frameworks, SwitchML [31] and ATP [22], and a pure software solution, BytePS, as baselines. We implement the NetRPC version on BytePS with only 500 LoC modifications. All INC versions use a single parameter server (PS), while the software version uses eight to provide enough throughput.

Figure 6 shows the average training speed per worker. We have the following observations: 1) INC solutions outperform non-INC ones for most models because they avoid incast to the PS. NetRPC, ATP, and SwitchML are 42%, 42%, and 11% faster than BytePS in VGG16; 2) For all models, NetRPC performs similar to ATP (97% to 100% of ATP), and at most 28% faster than SwitchML; 3) the training speeds on ResNet are similar because they are computation-intensive, and communication does not affect the overall performance much.

We believe the performance gain in NetRPC over existing systems is from the automatic parallel streams. As a side benefit, NetRPC uses only a single port (or one pipeline) instead of recirculation like ATP or SwitchML. Using fewer ports is essential for the multi-application data plane. SwitchML-RDMA [30] uses even more pipelines by chaining four pipelines together to achieve a performance gain over ATP. We do not adopt the design because resource efficiency is one of our key considerations.

Paxos. We use NetRPC to implement a Paxos [21] consensus system, offloading the leader and vote counting functions to switches. The implementation only contains about 700 LoC changes. We use an INC baseline, P4xos [6], and two software ones, `libpaxos` [24] and DPDK Paxos [6]. We run two proposers, two acceptors, and three learners in all cases.

Figure 7 summarizes the results on both throughput and 99th-percentile latency to achieve one consensus. Key findings include: 1) NetRPC achieves a maximum throughput of 503K messages/second, 12% higher than P4xos, and 7.86× and 4.93× higher than the two software solutions. INC solutions are much faster because they offload packet processing to the switch to alleviate the CPU bottleneck on servers. NetRPC has higher throughput than P4xos because it only sends the final results to the learners, reducing the workload on servers and saving the traffic on learner links. 2) The 99th-percentile latency of NetRPC is 311 ms and 96 ms shorter than software but 42 ms higher than P4xos. This is because we choose not to run the acceptors on switches like P4xos and thus need an extra round trip to the software acceptor. We believe the location and replication flexibility of the acceptor is a worthwhile tradeoff for the extra latency, given that it is still much faster than pure software.

6.4 Micro-benchmarks

To better understand NetRPC performance impact, we conduct a series of micro-benchmarks, focusing on INC-related functions only. We also use both prior INC arts and pure software DPDK implementation as comparison baselines.

Throughput. We perform SyncAtgr and AsyncAtgr on a 2-to-1 testbed and measure the *sender goodput*, using ATP and ASK as INC baselines.

The first row in Table 5 shows the result. NetRPC offers 9% higher throughput than ATP. The reason is that NetRPC does not apply recirculation (we use `copy` policy in this experiment) as ATP and SwitchML, which costs extra ports or pipelines on the switch . Instead, it relies on the parallel message sending (Section 4) to increase the goodput. Not surprisingly, both INC solutions outperform software solutions, e.g., NetRPC offers 26% higher goodput than pure DPDK. In fact, the end-to-end training results (42% faster, see Section 6.3) show an even larger improvement than the micro-benchmark, as in SyncAtgr, the shorter latency also improves GPU utilization as we spend less time waiting for the aggregation results.

The second row shows the goodput in AsyncAtgr. NetRPC achieves a similarly high throughput as ASK (about 73 Gbps). Unlike SyncAtgr, the keys count as part of a valid payload in this case, and thus the goodput is higher. Both INC solutions have 37% higher throughput than the pure DPDK.

Latency. We measure the average latency for the two latency-sensitive applications: Agreement and KeyValue, using P4xos voting and ElasticSketch [38] (monitoring) as baselines. The third row in Table 5 shows the average voting latency. Both NetRPC and P4xos outperform DPDK with a 76% latency

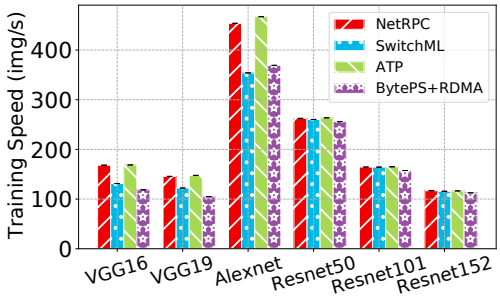
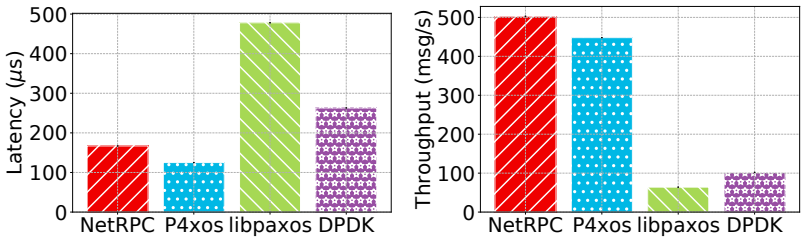


Figure 6: Deep Learning Training Speed



(a) 99th-Percentile Latency (b) Throughput

Figure 7: End-to-end Performance of Paxos Systems.

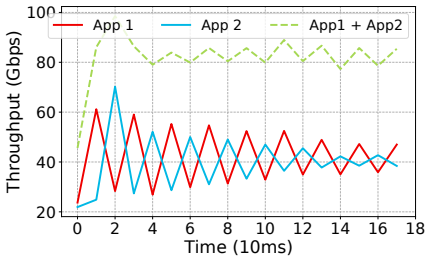


Figure 8: Congestion Control: Fairness

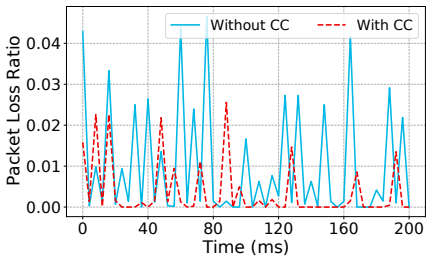


Figure 9: Congestion Control: Packet Loss

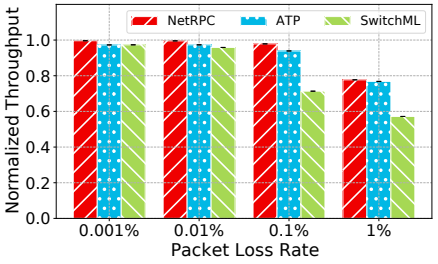


Figure 10: Packet Loss Rates vs. Throughput

reduction. NetRPC and P4xos offer similar latency, showing that NetRPC abstraction layers do not add extra latency.

The last two rows of Table 5 compares performance for KeyValue types, specifically in *flow counting*. Both NetRPC and ElasticSketch have lower latency than DPDK, by 13% and 20%, respectively. Notably, the last row of Table 5 also shows a 10× packet processing capacity increase from DPDK. NetRPC is about 0.26 ms (9%) slower than ElasticSketch because we do not have the same application-specific optimization that avoids modifying packets. We believe a less-than-10% latency increase is a reasonable price to pay for the general programming model by omitting the optimization.

Congestion control performance. To evaluate the effects of congestion control in NetRPC, we concurrently run two applications: a SyncAggr and an AsyncAggr on the same data plane (i.e., the same switch, host, and links), each having two clients and one server. Figure 8 shows the throughput over a short time period. We observe that the throughput quickly converges within 200 ms, and the combined bandwidth reaches 77% to 89% of the 100Gbps link. Also, the two application fairly shares the available bandwidth. Figure 9 shows the packet loss ratio over a short time period with/without congestion control. We can see that our ECN-based congestion and flow control reduces packet loss by about 63%, as it automatically adjusts the sending window to avoid overwhelming both the link and the server agent (Section 5.1).

Reliability mechanisms. To evaluate how NetRPC handles packet losses, we inject packet losses at different rates to emulate unreliable network. We run three INC applications NetRPC, ATP, and SwitchML and verified that all three correctly handles packet loss. Figure 10 shows the normalized throughput. NetRPC performs retransmission correctly under packet loss, using on-switch states only. At a high loss rate, NetRPC has a more graceful performance degradation. Compared with the no-loss case, NetRPC, ATP, and SwitchML's throughput decrease by 22%, 23%, and 43%, respectively. With 1% loss, NetRPC shows significantly less performance degradation than SwitchML because it adopts out-of-order ACKs and thus learns and reacts to packet loss faster.

Handling overflows. We run SyncAggr under synthetic workload varying overflow ratios from 0.001% to 1%. Figure 11 plots the throughput vs. overflow ratios. In all experiments, we check the computation results to ensure that NetRPC detects and corrects the overflow as we expect. When the overflow ratio exceeds 0.1%, we notice throughput degradation due to the software fallback. NetRPC still achieves about 65 Gbps throughput at 1% overflows. Note that the overflow ratio in real workload is far less than 1% with a reasonable quantization scaling factor for floating-point numbers. In contrast, the pure software solution only achieves a max of 40 Gbps.

Performance of `clear` policies. NetRPC offers three ways to handle `Map.clear` in `NetFilter` (Section 5.2.2). We measure the performance of a 2-to-1 SyncAggr using three `Map.clear` policies, and Table 6 summarizes the results. `Lazy` policy performance depends on the ratio of arithmetic overflow, and we use three ratios of 0%, 1%, and 10%. `Copy` policy achieves the highest throughput without extra memory cost but also has the highest latency because it relies on servers to backup

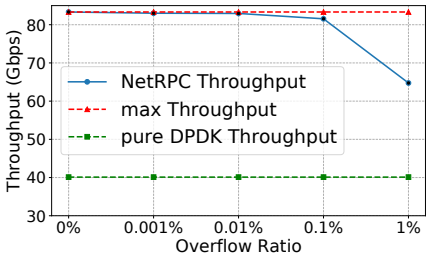


Figure 11: Overflow Ratio vs. Throughput

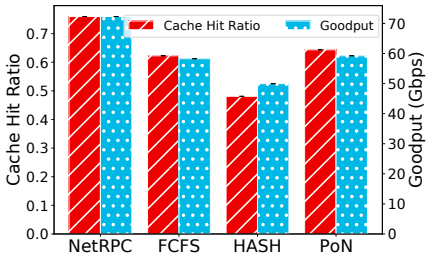


Figure 12: Caching Policy Comparison

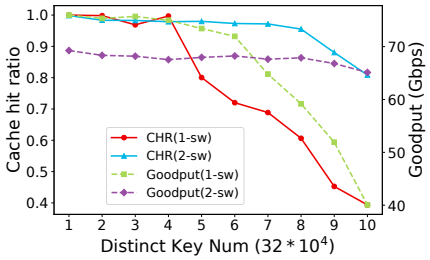


Figure 13: NetRPC on Two Switches

Table 6: Clear Policy Impact on Performance

	Latency	Memory	Throughput
copy	74μs	1x	83.11Gbps
shadow	24μs	2x	50.41Gbps
lazy (0%)	22μs	1x	83.31Gbps
lazy (1%)	23μs	1x	64.75Gbps
lazy (10%)	30μs	1x	34.82Gbps

Table 7: Concurrent Application Throughput and Latency

Metrics	1APP	4APP	4APP×5
Sync Goodput (Gbps)	50.55	24.88	24.84
Async Goodput (Gbps)	72.31	36.01	36.60
Goodput Sum (Gbps)	N/A	60.89	61.44
KeyValue Delay (ms)	3.52	3.56	3.85
Agreement Delay (μs)	20	21	24

the cleared states for reliability. `Shadow` policy offers a good latency of 24μs but doubles memory usage and has the lowest throughput because it needs to recirculate the packet and keep an extra copy. `Lazy` policy achieves both the highest throughput and lowest latency with no overflows. But as the overflow ratio increases, both metrics degrade. The actual accumulator overflow ratio depends on the data. Thus, we leave them as a user configuration in the `NetFilter`.

Cache policy. As we discuss in Section 5.2.2, a good cache policy alleviates traffic incast at the server and improves performance. We evaluate multiple cache policies. The experiment uses $32 \times 4K$ switch memory with 2-to-1 traffic. Comparison baselines are FCFS, hash-based caching (HASH), and Power of N (PoN). HASH policy uses the hash key as the index to address the switch memory (like ASK [2] and ATP [22]) and falls back to the server agent on hash collisions. PoN is a classic policy in sketches [38]: it only caches the hot keys whose hit number exceeds a threshold N and gives up caching when the switch memory is full. We tune the hyper-parameter N to maximize the performance experimentally.

Figure 12 shows the result. First, the CHR is positively correlated with the goodput, indicating the need for cache policy optimization. NetRPC's periodic cache update outperforms other cache policies by 18% $\sim$ 57% on cache hit ratio (CHR) and 22% $\sim$ 44% on goodput. HASH performs the worst because it ignores the locality of keys in the same packet: if some keys are cached, but their adjacent keys in the same stream are not due to hash collision, the entire packet will never hit the cache. PoN and FCFS behave similarly as they stop caching new hot keys if the cache has been fully filled. Compared with these baselines, NetRPC catches up the locality better and adapts to high-skewed key distribution better because it always caches the recent hot keys and periodically updates the switch cache to make up space for newer ones.

6.5 Multiple Concurrent Applications

An important goal of NetRPC is to support a multi-application data plane without switch rebooting. To evaluate the performance, we run multiple instances of all four application types in a 2-to-1 topology. We evaluate using three concurrency settings: 1) running a single application instance ("1APP "); 2) running one instance per type ("4APP"); and 3) running five instances per type ("4APP ×5") . Table 7 shows the total goodput and average latency. In all cases, we measure and report the throughput of SyncAgtr and AsyncAgtr and the latency of KeyValue and Agreement. In the 4APP ×5 case, we take the average of all instances of the measured type.

When concurrent applications increase from 4 to 20, we observe that the total bandwidth of SyncAgtr and AsyncAgtr stays roughly the same. Although KeyValue and Agreement do not use much bandwidth, they do contend for switch PPS (packets per second) and queue up in sending threads. The experiments show that small applications have little impact on bandwidth-heavy ones. We observe only a 20% latency increase compared to the 1APP case. These results demonstrate the successful resource sharing ability of NetRPC.

6.6 Running on Multiple Switches

Limited by available hardware, we only validate NetRPC's cross-switch capability with two-switches. We chain the two switches into a longer pipeline, and thus a packet can carry more key-value pairs. The NetRPC server agent decides which

key to put on which switch. We compare the performance of running 2-to-1 MapReduce on the testbed with one / two switches. We loop through the distinct keys multiple times, and thus a cache smaller than the number of distinct keys will suffer cache misses. Then we measure the CHR and the goodput varying with the number of distinct keys as an indicator of how well NetRPC is using memory on both switches.

Figure 13 shows the result. Each switch stores $M = 32 \times 40K$ values with distinct keys. We confirm that the goodput starts to drop at M using one switch, but $2M$ with two. The peak goodput decreases slightly with more switches (from 75 Gbps to 69 Gbps), mainly because of the increased host workload to encode more keys into the packet. Beyond the switch memory capacity, the goodput first decreases slightly (5.3% of peak throughput with $1.5M$ keys for one switch) and then dramatically (22% with $2M$ keys). This is because offering a 75 Gbps workload, there is little hope that the server CPU can handle many cache misses. Nevertheless, the two-switch setting shows a $1.63\times$ improvement over the one-switch case when handling $2.5M$ distinct keys, showing that NetRPC can efficiently utilize memory on multiple switches.

7 Conclusion and Future Work

In-network computation (INC) comes from software-defined networking (SDN), but INC is fundamentally different from SDN because it mainly provides *computation* service instead of *communication*. Thus, we need a new programming model for INC to *better describe computation*. We need high-level data structures, collections, memory, and procedure calls that center around end-hosts instead of packets, headers, tables, and pipelines that center around switches. On the other hand, we recognize that the INC data plane is still a shared network infrastructure, not an application-specific accelerator. Thus, both generality and multi-application support are essential.

NetRPC, to our knowledge, is the first framework that integrates INC into the familiar RPC programming model. NetRPC allows users to implement different types of INC applications using the familiar gRPC framework and run them on a single shared INC data plane. NetRPC achieves 97% of LoC deduction for INC applications and offers similar or better performance boosts than handcrafted systems.

Current NetRPC mainly focuses on *mechanisms* of INC + RPC integration. In future work, we will focus on *policies*, such as scheduling among different applications, efficient sharing between INC workload and other SDN or traditional network traffic, efficient end-host CPU, GPU, and INC co-scheduling. We will also explore NetRPC on more complex topologies, especially those with oversubscribed links. We will extend NetRPC congestion control with more fine-grained window adjustment. We will open source NetRPC on the publication of this paper to benefit the INC community.

References

[1] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.

[2] Anonymous. ASK: In-network aggregation service for key-value streams, 2022. https://anonymous.4open.science/r/ASK-80BF.

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[4] Anonymized Internet traces, 2008. https://www.caida.org/data/passive/passive_dataset.xml.

[5] Cisco. One silicon, one experience, multiple roles, 2019. https://blogs.cisco.com/sp/one-silicon-one-experience-multiple-roles.

[6] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*, 28(4):1726–1738, 2020.

[7] Jiaqi Gao, Ennan Zhai, Hongqiang Harry Liu, Rui Miao, Yu Zhou, Bingchuan Tian, Chen Sun, Dennis Cai, Ming Zhang, and Minlan Yu. Lyra: A cross-platform language and compiler for data plane programming on heterogeneous asics. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 435–450, 2020.

[8] Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, and Srinivas Narayana. Autogenerating fast packet-processing code using program synthesis. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, pages 150–160, 2019.

[9] Google. Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data, 2008. https://developers.google.com/protocol-buffers.

[10] Google. gRPC: A high performance, open source universal rpc framework, 2020. https://grpc.io/.

[11] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. Scalable hierarchical aggregation protocol SHArP: a hardware architecture for efficient data reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 1–10. IEEE, 2016.

[12] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 conference of the ACM special interest group on data communication*, pages 357–371, 2018.

[13] Ian Horrocks, Peter F Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, Mike Dean, et al. Swrl: A semantic web rule language combining owl and ruleml. *W3C Member submission*, 21(79):1–31, 2004.

[14] ImageNet. Imagenet is an image database organized according to the wordnet hierarchy, 2022. https://www.image-net.org/.

[15] Intel. DPDK is the data plane development kit that consists of libraries to accelerate packet processing workloads running on a wide variety of cpu architectures., 2013. https://www.dpdk.org/.

[16] Intel. Barefoot tofino, 2020. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.

[17] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. Life in the fast lane: A line-rate linear road. In *Proceedings of the Symposium on SDN Research*, pages 1–7, 2018.

[18] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, 2018.

[19] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 121–136, 2017.

[20] George Karlos, Henri Bal, and Lin Wang. Don't you worry'bout a packet: Unified programming for in-network computing. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, pages 99–107, 2021.

[21] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[22] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. ATP: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761. USENIX Association, April 2021.

[23] Alberto Lerner, Rana Hussein, Philippe Cudre-Mauroux, and U eXascale Infolab. The case for network accelerated query processing. In *CIDR*, 2019.

[24] General purpose Paxos library, 2013. https://bitbucket.org/sciascid/libpaxos.

[25] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable load balancing for Large-Scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 143–157, 2019.

[26] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 15–28, 2017.

[27] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 85–98, 2017.

[28] NPL. Open, high-level language for developing feature-rich solutions for programmable networking platforms, 2021. https://nplang.org/.

[29] Amedeo Sapio, Ibrahim Abdelaziz, Abdulla Aldilaijan, Marco Canini, and Panos Kalnis. In-network computation is a dumb idea whose time has come. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 150–156, 2017.

[30] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan Ports, and Peter Richtarik. Scaling distributed machine learning with In-Network aggregation. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 785–808. USENIX Association, April 2021.

[31] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan RK Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701*, 2019.

[32] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 15–28, 2016.

[33] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 127–132, 2013.

[34] Hardik Soni, Myriana Rifai, Praveen Kumar, Ryan Doenges, and Nate Foster. Composing dataplane programs with μP4. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 329–343, 2020.

[35] Brent E Stephens, Darius Grassi, Hamidreza Almasi, Tao Ji, Balajee Vamanan, and Aditya Akella. Tcp is harmful to in-network computing: Designing a message transport protocol (mtp). In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, pages 61–68, 2021.

[36] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. Cheetah: Accelerating database queries with switch pruning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2407–2422, 2020.

[37] Raajay Viswanathan and Aditya Akella. Network-accelerated distributed machine learning using mlfabric. *arXiv preprint arXiv:1907.00434*, 2019.

[38] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 561–575, 2018.

[39] Yelp. An all-purpose dataset for learning, 2022. https://www.yelp.com/dataset.

[40] Zhuolong Yu, Yiwen Zhang, Vladimir Braverman, Mosharaf Chowdhury, and Xin Jin. Netlock: Fast, centralized lock management using programmable switches. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 126–138, 2020.

[41] Bohan Zhao, Xiang Li, Boyu Tian, Zhiyu Mei, and Wenfei Wu. Dhs: Adaptive memory layout organization of sketch slots for fast and accurate data stream processing. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 2285–2293, 2021.

[42] Hang Zhu, Tao Wang, Yi Hong, Dan RK Ports, Anirudh Sivaraman, and Xin Jin. NetVRM: Virtual register memory for programmable networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 155–170, 2022.

Table 8: Arithmetic Operations in `Stream.modify`

OP	Semantics
MAX	stream.value = max(stream.value, para)
MIN	stream.value = min(stream.value, para)
ADD	stream.value += para
ASSIGN	stream.value = para
SHIFTL	stream.value <<= para
SHIFTR	stream.value >>= para
BAND	stream.value &= para
BOR	stream.value |= para
BNOT	stream.value = ∼stream.value
BXOR	stream.value ^= para

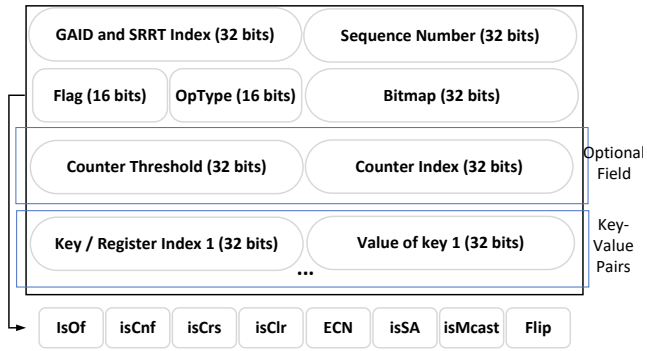


Figure 14: NetRPC Packet Format

A Arithmetic in NetRPC

We list the arithmetic operators of `Stream.modify` and their semantics supported by NetRPC in Table 8.

B NetRPC Protocol

B.1 Packet Format

The packet contains three kinds of fields. The key-value pairs encode the application data with the format of an array of <key/index, value> tuples; computation control fields encode the `NetFilter` configurations and guide the switch program for the computation; transport control fields maintain the channel connection.

Key-value pairs. Each NetRPC packet carries 32 key-value pairs. These pairs are either processed on the switch or the server agent by the selected primitives. The computation results are also carried back in the same position.

Computation Control Fields. The control flag bits contain the basic information about primitives selection. Current bits in use can indicate the following choices: whether any overflow happens (`isOf`); whether to use `CntFwd` (`isCnf`); whether to clear the target memory (`isClr`).

`OpType` indicates the type of arithmetic operation on key-value pairs. NetRPC supports various line-rate on-packet computation as we discuss in Appendix A. In `bitmap` field, the *i*-th bit in the bitmap indicates whether the switch should process the *i*-th key-value pair. The `CntFwd` fields only come into effect when the `isCnf` flag is set. `counter index` tells the switch which counter (register) to increase; when the register value equals to the `counter threshold`, the switch should forward the packet instead of dropping it.

Transmission Control Fields. Concurrent NetRPC connections (de)multiplex the network, and NetRPC distinguishes the flows by the `GAID`. On hosts, received packets are classified to the applications; on the switch, the `GAID` is also used for admission control. In NetRPC, each sending thread maintains a short-term connection to serve applications' calls/tasks and thus assigns a sequence number (starting from zero) for each packet. In addition, the reliability control requires sending threads to maintain a long-term connection (cross the tasks) with the switch. The field State Register of Reliable Transmission `SRRT` is the switch memory address to store the state, and the `flip bit` is the reliable state to store. Some bits in the `Control Flag` also controls the routing: whether the packet should cross the switch to the server agent (`isCross`); `ECN` indicates whether the switch is experiencing congestion (queue buildup); whether the packet comes from the server agent (`isSA`); whether to multicast the packet (`isMcast`).

Optimization. Some optional fields will be removed if unnecessary in the computation to improve the network bandwidth efficiency and the goodput. (1) If we address the key-value or value stream linearly to the switch memory, we can eliminate the key fields and indicate the starting index of the memory segment by the `counter index` field. (2) If the computation does not need `CntFwd`, we can eliminate the `CntFwd` fields.

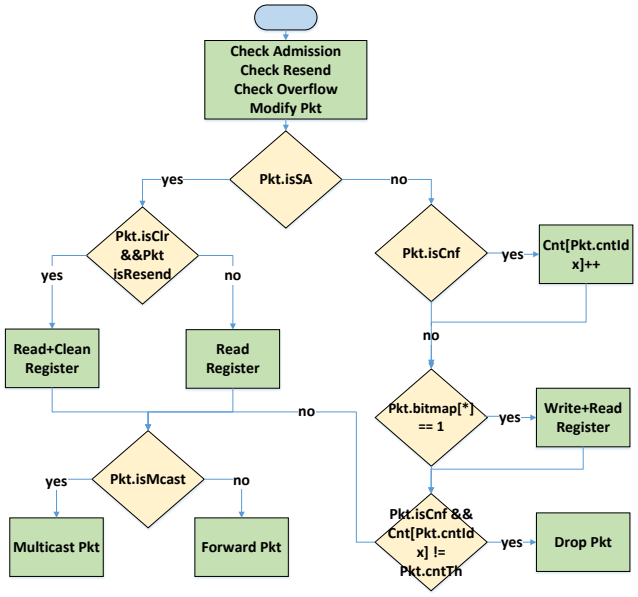


Figure 15: NetRPC Switch Logic

C Switch pipeline details

There is a 12-stage pipeline in our switch, and we use 8 to implement the map access primitives. The remaining four stages handle the reliable transmission, flow and congestion control, as well as `Stream.modify` and the `CntFwd` primitive. Figure 15 illustrate the flowchart for switch logic.

When the switch receives a NetRPC packet, it will first check whether the corresponding application (GAID) has registered. Unregistered packets will be forwarded as normal ones. Moreover, the switch checks whether it receives the packet for the first time. Otherwise, it avoids `Map.addTo`/`Map.clear` primitives on the switch memory but still `Map.get` values from registers into the packets. An overflow packet will be forwarded directly for fallback without on-switch processing.

For packets to the server, the switch first executes `Stream.modify` and `CntFwd` if required, then processes key-value pairs in the packet: `Map.addTo` the switch registers and `Map.get` the computation results back to replace the value. The switch drops those packets that enable `CntFwd` but do not reach the threshold and forwarded/multicast the rest packets.

For packets from the server, the switch first `Map.get` register values into the packet and then decides whether to clear the corresponding registers. The switch will forward/multicast the packets according to control flags and routing rules.

D NetRPC Implementation Examples

We enumerate some NetRPC implementation of classic INC applications: MapReduce, lock server, and network monitoring in Figure 16 to 24.

```
import "netrpc.proto"
message ReduceRequest {
  netrpc.STRINTMap kvs = 1;
}
message ReduceReply {
  string msg = 1;
}
message QueryRequest {
  string msg = 1;
}
message QueryReply {
  netrpc.STRINTMap kvs = 1;
}
service MapReduce {
  rpc ReduceByKey (ReduceRequest) returns
      (ReduceReply) {} filter "reduce.nf"
  rpc Query (QueryRequest) returns (
      QueryReply) {} filter "query.nf"
}
```

Figure 16: RPC Service Definition of Distributed MapReduce

```
{ //reduce.conf
  "AppName": "MR-1",
  "Precision": 0,
  "get": "nop",
  "addTo": "ReduceRequest.kvs",
  "clear": "nop",
  "modify": "nop",
  "CntFwd": {
    "to": "SRC",
    "threshold": 0,
    "key": "NULL",
  },
}
{ //query.conf
  "AppName": "MR-1",
  "Precision": 0,
  "get": "QueryReply.kvs",
  "addTo": "nop",
  "clear": "nop",
  "modify": "nop",
  "CntFwd": {
    "to": "SRC",
    "threshold": 0,
    "key": "NULL",
  },
}
```

Figure 17: `NetFilter` of Distributed MapReduce

```
shared_ptr<Channel> channel =
    CreateCustomChannel(server_ip,
    InsecureChannelCredentials());
unique_ptr<Stub> stub_(NewStub(channel));
pair<string,int>* MapReduce(pair<string,
    int>* data, int length) {
    ReduceRequest request1;
    ReduceReply reply1;
    ClientContext context1;
    for(int i = 0; i<length; i++){
        (*request1.mutable_kvs()->
            mutable_map())[data[i].first]
            = data[i].second;
    }
    Status status = stub_->ReduceByKey(&
        context1, request1, &reply1);
    QueryRequest request2;
    QueryReply reply2;
    ClientContext context2;
    stub_->Query(&context2, request2, &
        reply2);
    int sz = reply2.mutable_kvs()->
        mutable_map()->size(), idx = 0;
    pair<string,int>* output = new pair<
        string,int>[sz];
    for(auto it: (*reply2.mutable_kvs()->
        mutable_map())){
        output[idx].first = it.first;
        output[idx++].second = it.second;
    }
    return output;
}
```

Figure 18: Client Stub for Distributed MapReduce

```
import "netrpc.proto"
message LockRequest {
  netrpc.STRINTMap map = 1;
}
message LockReply {
  string msg = 1;
}
message ReleaseRequest {
  netrpc.STRINTMap map = 1;
}
message ReleaseReply {
  string msg = 1;
}
service Lock {
  rpc GetLock (LockRequest) returns (
      LockReply) {} filter "lock.nf"
  rpc Release (ReleaseRequest) returns (
      ReleaseReply) {} filter "release.nf"
}
```

Figure 19: RPC Service Definition of Distributed Lock Server

```
{ //lock.conf
  "AppName": "LS-1",
  "Precision": 0,
  "get": "nop",
  "addTo": "nop",
  "clear": "nop",
  "modify": "nop",
  "CntFwd": {
    "to": "SRC",
    "threshold": 1,
    "key": "LockRequest.kvs",
  },
}
{ //release.conf
  "AppName": "LS-1",
  "Precision": 0,
  "get": "nop",
  "addTo": "nop",
  "clear": "copy",
  "modify": "nop",
  "CntFwd": {
    "to": "SRC",
    "threshold": 0,
    "key": "ReleaseRequest.kvs",
  },
}
```

Figure 20: `NetFilter` of Distributed Lock Server

```
shared_ptr<Channel> channel =
    CreateCustomChannel(server_ip,
    InsecureChannelCredentials());
unique_ptr<Stub> stub_(NewStub(channel));
void BlockingLock(string* lockTarget, int
    length) {
    LockRequest request1;
    LockReply reply1;
    ClientContext context1;
    for(int i = 0; i<length; i++){
        (*request1.mutable_kvs()->
            mutable_map())[lockTarget[i]]
            = 1;
    }
    Status status = stub_->LockSend(&
        context1, request1, &reply1);
    /* critical section */
    ReleaseRequest request2;
    ReleaseReply reply2;
    ClientContext context2;
    for(int i = 0; i<length; i++){
        (*request2.mutable_kvs()->
            mutable_map())[lockTarget[i]]
            = 0;
    }
    stub_->Release(&context2, request2, &
        reply2);
}
```

Figure 21: Client Stub for Blocking Lock Acquire and Release

```
import "netrpc.proto"
message MonitorRequest {
  netrpc.STRINTMap kvs = 1;
  string payload = 1;
}
message MonitorReply {
  string payload = 1;
}
message QueryRequest {
  string message = 1;
}
message QueryReply {
  netrpc.STRINTMap kvs = 1;
}
service Monitor {
  rpc MonitorCall (MonitorRequest) returns
       (MonitorReply) {} filter "monitor.
      nf"
  rpc Query (QueryRequest) returns (
      QueryReply) {} filter "query.nf"
}
```

Figure 22: RPC Service Definition of Network Monitoring

```
{ //monitor.conf
  "AppName": "MON-1",
  "Precision": 0,
  "get": "nop",
  "addTo": "MonitorRequest.kvs",
  "clear": "nop",
  "modify": "nop",
  "CntFwd": {
    "to": "SERVER",
    "threshold": 0,
    "key": "NULL",
  },
}
{ //query.conf
  "AppName": "MON-1",
  "Precision": 0,
  "get": "QueryReply.kvs",
  "addTo": "nop",
  "clear": "nop",
  "modify": "nop",
  "CntFwd": {
    "to": "SRC",
    "threshold": 0,
    "key": "NULL",
  },
}
```

Figure 23: `NetFilter` of Network Monitoring

```
1  shared_ptr<Channel> channel =
       CreateCustomChannel(server_ip,
       InsecureChannelCredentials());
2  unique_ptr<Stub>stub_(NewStub(channel));
3  pair<string,int>* MonitorRPC(string*
       metrics, int length) {
4      MonitorRequest request1;
5      MonitorReply reply1;
6      ClientContext context1;
7      for(int i = 0; i<length; i++){
8          (*request1.mutable_kvs()->
               mutable_map())[metrics[i].
               first] = 1;
9      }
10     request1.payload = "Hello";
11     Status status = stub_->MonitorCall(&
           context1, request1, &reply1);
12     if (status.ok()) {
13       cout << reply1.payload << endl;
14     }
15     QueryRequest request2;
16     QueryReply reply2;
17     ClientContext context2;
18     stub_->Query(&context2, request2, &
           reply2);
19     int sz = reply2.mutable_kvs()->
           mutable_map()->size(), idx = 0;
20     pair<string,int>* output = new pair<
           string,int>[sz];
21     for(auto it: (*reply2.mutable_kvs()->
           mutable_map())){
22         output[idx].first = it.first;
23         output[idx++].second = it.second;
24     }
25     return output;
26 }
```

Figure 24: Client Stub for RPC with Monitoring