NetStar: A Future/Promise Framework for Asynchronous Network Functions

Jingpu Duan, Xiaodong Yi, Junjie Wang, Chuan Wu, and Franck Le

***Abstract*— Network functions (NFs) are more than simple packet processors that apply various transformations to the packet content. Modern NFs often resort to various external services to achieve their purposes, *e.g.*, storing flow states in an external storage or looking up a DNS. Working with external services is usually implemented using callback-based asynchronous programming, which is complex and error-prone. This paper proposes *NetStar*, a new NF programming framework that brings the future/promise abstraction to the NF dataplane for flow processing. *NetStar* simplifies asynchronous NF programming via a carefully designed async-flow interface that exploits the future/promise paradigm by chaining multiple continuation functions for asynchronous operations handling. The programs implemented using the *NetStar* framework mimic simple synchronous programming but are able to achieve full flow processing asynchrony. We have used *NetStar* to implement a number of representative NFs. Our experience and evaluation results show that *NetStar* can effectively simplify asynchronous NF programming by substantially reducing the lines of code, while still approaching line-rate packet processing speeds.**

***Index Terms*— NFV, asynchronous programming, future/promise paradigm, high-performance software dataplane.**

I. Introduction

NETWORK Functions (NFs) are more than packet processors that perform simple transformations on each received packet. Modern NFs, *e.g.*, firewall [1], NAT [1], IDS [2], and proxies [3], [4], often need to contact external services while processing network flows, *e.g.*, for querying external databases [5], [6], or saving critical per-flow states on reliable storage to resist failures [1]. To ensure high-speed packet processing while executing external queries, these NFs must fully exploit asynchronous programming: after generating a request to an external service, the NF should not block-and-wait for the response in a synchronous fashion; instead, it should save the current processing context, register a callback function to handle the response upon its return, and switch to process other tasks.

Compared with synchronous NF programs, asynchronous NF implementation using callbacks is more efficient in packet processing, as it does not waste important CPU time. However, callback-based asynchronous programming has some inherent drawbacks that can prevent developers from building NFs with richer functionalities.

First, compared to a synchronous program, a callback-based asynchronous program is harder to implement and reason about. Such a program may define multiple callback functions, scattered in different places of the source files, to achieve a series of asynchronous operations. For instance, the Bro IDS can be configured to detect malware using two nested callback functions defined in different places (Sec. III-A); and a NAT may replicate important per-flow states in an external database using 4 consecutive callbacks, to read from and write to a remote database [1]. Dealing with multiple callbacks scattered in different places can be confusing, making it more difficult for a programmer to trace the execution order of the program.

Second, visiting saved context information inside a registered callback can be error-prone. Since an asynchronous program immediately switches to other tasks after saving the context and registering a callback, the program must ensure that the saved context is not accidentally freed until the callback is invoked. Failing to do so may lead to invalid memory access and program crash. However, when multiple callback functions are used, the programmer may accidentally free the context if he fails to correctly trace the execution order of the callbacks.

Third, callback-based asynchronous program may introduce redundant error handling code. Since exceptions may be generated when waiting for external responses, the program must properly process the generated exceptions by implementing error handling code in the callbacks registered for the responses. When a series of asynchronous operations are executed, the programmer has to add redundant error handling code for each asynchronous operation.

People have recognized the problems with callbacks when building event-driven systems such as browsers [7], [8], programming language runtime systems [9], web servers [10] and database servers [11]. Their solution to counter these problems is to use an advanced programming abstraction, such as the coroutine [12] and the future/promise paradigm [13]–[15].

Coroutine is a user-space cooperative thread that is able to execute synchronous program in a fully asynchronous fashion.

Manuscript received October 11, 2018; revised December 27, 2018; accepted January 11, 2019. Date of publication February 5, 2019; date of current version February 14, 2019. This work was supported in part by Hong Kong RGC under Contract HKU 17204715, Contract 17225516, and Contract C7036-15G (CRF), in part by NSFC under Grant 61628209, in part by the HKU URC Matching Fund, in part by Huawei under Grant HIRP HO2016050002BE, and in part by the project PCL Future Regional Network Facilities for Large-scale Experiments and Applications under Grant PCL2018KP001. *(Corresponding author: Jingpu Duan.)*

J. Duan is with the SUSTech Institute of Future Networks, Southern University of Science and Technology, Shenzhen 518005, China, and also with the PCL Research Center of Networks and Communications, Peng Cheng Laboratory, Shenzhen 518005, China (e-mail: duanjp@sustc.edu.cn).

X. Yi, J. Wang, C. Wu are with the Department of Computer Science, The University of Hong Kong, Hong Kong (e-mail: xdyi@cs.hku.hk; jjwang@cs.hku.hk; cwu@cs.hku.hk).

F. Le is with the IBM T. J. Watson Research Center, Yorktown Heights, NY 10598 USA (e-mail: fle@us.ibm.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSAC.2019.2894303

However, a coroutine is usually constructed with its own stack, which may be several kilobytes in size [16]. When NF processes a large number of small flows, saving and restoring coroutine stacks, as well as frequent stack memory allocation may incur performance penalties.

In contrast, the future/promise paradigm uses small runtime objects, including futures, promises and continuation functions, to mimic synchronous programming while being fully asynchronous. Besides making asynchronous program easier to implement, the future/promise abstraction also reduces redundant error handling code by effectively propagating exceptions to a consolidate error handling logic, and simplifies context management.

A recent open-source C++ library, Seastar [17], implements high-performance future/promise paradigm for building database server [18]. The library is integrated with DPDK [19] for high-performance packet IO and provides a customized user-space TCP/IP stack, making it a good candidate for building high-performance asynchronous NFs. However, the library misses a core interface for manipulating raw network packets with future/promise paradigm. Being a crucial part for building dataplane NFs, this interface is not trivial to implement. A straightforward design is to expose a regular packet handler function, that is invoked for each received packet. While this interface is easy to use, it falls back to callback-based asynchronous programming and leaves no room for utilizing the future/promise abstraction.

This paper discusses the design and implementation of *NetStar*, a future/promise-based framework for simplifying asynchronous programming in NFs. With the help of future/promise abstraction, programming asynchronous operations in *NetStar* is similar to implementing simple synchronous programs, and the resulting NFs are guaranteed to never block-and-wait for responses as in synchronous programs.

The power of *NetStar* is mainly attributed to the core interface that we design, called the async-flow interface. Powered by a simulated packet processing loop, the interface combines high-performance packet IO with future/promise abstraction and retains all the power of future/promise.

To evaluate the performance of *NetStar*, we build a number of NFs using the framework, including four NFs from the StatelessNF paper [1], an HTTP reverse proxy, an IDS and a malware detector. With extensive experiments, we show that NFs based on *NetStar* use substantially fewer lines of code to implement asynchronous packet processing, as compared to callback-based implementation, while delivering sufficiently good performance. We also compare *NetStar* with a coroutine based implementation, and show that the coroutine is a less desirable paradigm for implementing NFs that process a large number of concurrent network flows.

In summary, this paper makes the following contributions.

- We design and implement *NetStar*, the first high-performance framework that utilizes future/promise paradigm for simplifying the implementation of asynchronous NFs.
- Existing future/promise library lacks an important interface for processing network packets. We improve this by designing async-flow interface, which effectively combines future/promise paradigm with high-performance packet IO.
- We report our experience in building asynchronous NFs with *NetStar*, evaluate the performance of the implemented NFs and open-source our framework in [20].

The rest of the paper is organized as follows. Sec. II discusses the related work of *NetStar*. Sec. III presents the motivations for building *NetStar* and gives a brief tutorial for future/promise paradigm. Sec. IV presents an overview of *NetStar*. Sec. V describes the detailed design and usage of async-flow interface. Sec. VI reports the asynchronous NFs that we build with *NetStar*. Sec. VII demonstrates the performance evaluation of the implemented NFs. Sec. VIII discusses the limitations and miscellaneous issues related with *NetStar*. Sec. IX gives a concluding remark and presents future research directions for improving *NetStar*.

II. Related Work

There are several existing frameworks [21]–[24] that are designed to aid the implementation of NFs, but none of them aims for simplifying the asynchronous programming in NFs.

NetBricks [21] provides a high-level abstraction for building fast packet processing pipelines, but it does no address how to simplify asynchronous programming on NF dataplane. *NetStar* leverages future/promise abstraction to process dataplane traffic, and simplifies asynchronous programming with the async-flow interface. P4 [22] provides a high-level programming language for describing dataplane packet processing pipelines, but it has no intrinsic support for manipulating dataplane packets asynchronously as in *NetStar*.

mOS [23] proposes a unified interface for managing connection oriented middleboxes. Using its interface, a middlebox can extract application-level payload and apply different callback functions to process the payload. *NetStar* shares a similar event layer as in mOS, where the packet is preprocessed to expose interested events to the core NF processing logic; however, *NetStar* uses the future/promise abstraction for event handling, and can effectively simplify implementation in case the NF requires to contact external services.

S6 [24] is a framework for building scalable NFs. It utilizes coroutine to process asynchronous messages when scaling out. S6 cannot be used to implement asynchronous operations in NFs, as the coroutines are completely hidden from the public APIs exposed by S6. On contrary, *NetStar* exposes async-flow interface to implement arbitrary asynchronous operations in NF core logic.

Except for packet processing, modern NFs require additional functionalities to resist failures and handle large workload. StatelessNF [1] proposes a new architecture that separates the storage of per-flow states from the processing of packets. This advanced architecture requires efficient and simplified asynchronous programming support, which is nicely provided by *NetStar*. OpenNF [25], Split/Merge [26] and PEPC [27] all use flow migration for dynamical scaling of NF instances (by migrating flows out of hotspots). Implementation of a flow migration protocol typically involves complex asynchronous interactions, and *NetStar* can be potentially useful to simplify flow migration implementation.

The mobile networks, including 4G-EPC and the upcoming 5G networks, have widely adopted NFV technology for dynamic resource provisioning and improved quality-of-service (QoS). In [27], the authors propose a new architecture for building high-performance EPC core. Bagaa *et al.* [28] propose a new approach based on coalition game for combining core elements of vEPC and 5G mobile network into a federated cloud. In [29]–[33], optimal algorithms for placing virtual network functions in virtualized environment are proposed under different scenarios, including EPC core, 5G mobile system and edge network. Addad *et al.* [34] propose an algorithm for computing optimal network slices in the cloud. *NetStar* can be used to simplify the implementation of many network functions in EPC and 5G mobile network [27], [28], and can borrow ideas from the algorithmic advancement in the placement of virtual network functions [29]–[34] for further improvement when used in virtualized environments.

III. Motivation and Background

A. Asynchronous Programming in Representative NFs

1) Bro: We use a concrete example, malware detection in Bro IDS [2], to demonstrate how callbacks are used to program asynchronous NFs. Bro can be configured to detect malware from transmitted files as follows. A local database server stores hash values of commonly-seen malware. For each TCP connection that goes through, Bro reconstructs its byte stream. If a transmitted file is detected within the byte stream, Bro computes a SHA1 hash value over the file content, and queries the local database to obtain a quick response about whether the hash value matches any malware's. In case of no hit in the local database, Bro can turn to the more complete and up-to-date Malware Hash Registry (MHR) [35], which provides a special DNS service: Bro can send a DNS request carrying the file hash to MHR, and the MHR responds whether there is a match with any of the malware hash values it has. If either of the two queries detects some malware, Bro raises an alarm in the log file.

We extract the code covering the execution path of malware detection from Bro version 2.5-359, and show it in Fig. 1. There are three callback functions registered in different places of the Bro's source files: after a transmitted file is detected, Bro calls `Hash::Finalize()` to calculate the file's hash value and posts a `file_hash` event to the event engine (line 5). Bro handles this event in the first callback `EventHandler::Call`, by performing an asynchronous query to the local database (lines 11 - 14). The response of the database query is handled by the second callback `StoreQueryCallback::Result`. If the file hash is not found in the local database, Bro performs a DNS query to the MHR (line 23), whose response is handled by the third callback `LookupHostCallback::Resolved`. Since both the database and MHR queries may fail, two additional callbacks, `Trigger::Timeout()` and `LookupHostCallback::Timeout()`, are registered to handle the timeout errors of database and MHR query, respectively.

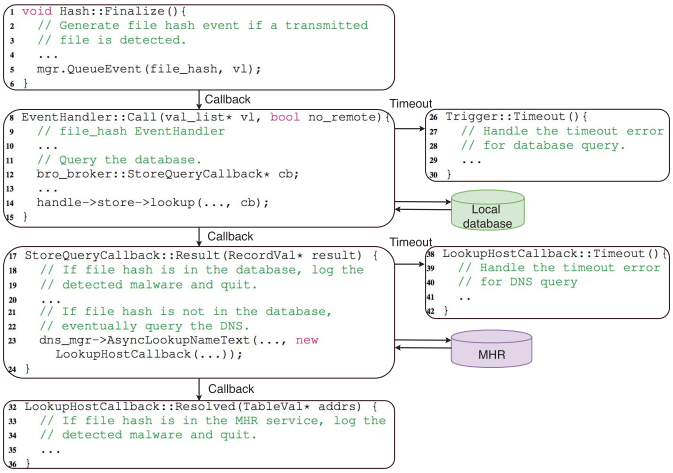


Fig. 1. Malware detection in Bro.

The code in Fig. 1 highlights the following disadvantages in callback-based asynchronous programming: (1) The asynchronous code is more complex to implement as compared to a synchronous program. For instance, if the second callback `StoreQueryCallback::Result` is to be implemented in a synchronous fashion, the programmer can handle the DNS response and the timeout event immediately after issuing a blocking DNS query, without the need for saving the context information (omitted from line 23) and defining two extra callback functions elsewhere (lines 32 and 38). (2) Omitted in line 23, saving and retrieving context information can be quite troublesome, as Bro relies on reference counting to keep allocated objects alive, and needs to correctly increase the reference counts of all the objects kept in the context when registering a callback. (3) Two functions are defined to handle potential errors in the two asynchronous operations. The two functions are redundant as they implement similar logic, which is to log a malware detection error.

2) HAProxy: We also inspect asynchronous programming in HAProxy [3], a high-performance proxy due to its callback-based asynchronous design. HAProxy can serve as a load-balancer or a gateway, which intercepts incoming connections, relays application requests to backend servers and then relays responses back to the clients. In order to retrieve data from incoming connections, HAProxy invokes three nested callbacks, including `iocb` (for accessing a connection), `recv` (for receiving data from the connection) and `rcv_buf` (for putting received data into buffer). The three callbacks are registered multiple times in various source files, making it hard to trace the execution flow of HAProxy without a deep knowledge of the source code.

B. Advanced Programming Abstractions

There is a tradeoff between simplicity and performance when implementing a NF program: on one hand, synchronous programming is easier but incurs poor performance. On the other hand, asynchronous programming with callbacks achieves better performance at the cost of implementation complexity. Similar tradeoffs have been identified when

people build server programs, such as web [10] and database servers [11]. The solution in those domains is to use advanced programming abstractions, including coroutine [12] and future/promise [9], [13]–[15], to manage the complexity of asynchronous programming while achieving good performance.

Coroutine is a user-space cooperative thread: a coroutine explicitly yields its execution to other coroutines when it waits for asynchronous operations to complete. The coroutine paradigm has found its success in building asynchronous web servers [10] and databases [11].

With coroutine, asynchronous NFs can be very easy to implement. The NF program can be directly written as a synchronous program, which runs as multiple coroutines (threads) to handle different flows. Whenever a coroutine needs to perform an asynchronous operation, it yields its execution to other coroutines after saving its thread context, and then waits to be resumed when the asynchronous response arrives in the future.

A coroutine is usually constructed with its own stack, which may be several kilobytes in size [16], and a typical high-performance NF needs to process hundreds of thousands of flows concurrently and millions of packets per second. An asynchronous NF implemented using coroutine needs to frequently allocate coroutine stacks and switch between different coroutine contexts, resulting in compromised performance (verified in Sec. VII-G).

In search for an advanced programming abstraction for simplifying the implementation of asynchronous NFs, we have identified that the future/promise paradigm can be a good alternative besides coroutine. Future/promise paradigm uses small and efficient runtime objects to simplify asynchronous programming, and can approach satisfactory performance even when processing a large number of concurrent network flows.

C. Future/Promise

The rest of this section briefly introduces the key concepts of future/promise abstraction, using pseudocode from Seastar library. To understand this section, we only assume the knowledge of basic C++ programming language. However, readers are recommended to possess background knowledge about C++11/14 features to have a better understanding of both this section and the *NetStar* framework. A good explanation of C++11/14 features can be found in [36]. The Seastar library [17] also provides a detailed documentation on future/promise paradigm.

1) Future: A future object has type `future<T>` and contains a value of type `T`. There are two states for each future object, available and unavailable. When in the available state, a future object either contains a concrete value of type `T` that can be directly used, or an exception. In the unavailable state, a future object is associated with a promise object, and can be turned into available state using the associated promise object.

2) Promise: A promise object has type `promise<T>`. A future object can be obtained from a promise object with `promise::get_future` method. When needed (*e.g.*, when the response of a remote query arrives, or the response times out and an exception is generated), the programmer can set either a concrete value, or an exception, to the promise object with the `promise::set_value` method. The value/exception is then automatically propagated to the associated future object and turns it into the available state.

3) Continuation Function: Future/promise works together with continuation functions to achieve asynchronous programming. A continuation function can be appended to a future object to work as a special callback function, using the `future::then` method. If the appended future object is available, the continuation function is immediately invoked; otherwise, it is invoked when the future object becomes available.

The following code shows how to append a continuation function to the future object `fur`.

```
future<T> fur;
fur.then([captured_variables...](T t){
  // Do regular processing.
  ...
  future<X> new_fur;
  return new_fur;
});
```

Here, the continuation function is represented as a C++ lambda function [37], and passed as an argument to `then` method. `captured_variables` is a list of variables which can be freely used in the continuation function. In practice, `captured_variables` may contain a pointer to a context object and the continuation function can visit the context by following the pointer. When `fur` becomes available, the continuation function is called. `fur` passes its concrete value as the argument `t` into the continuation function, which processes the argument and returns a new future object `new_fur`.

4) Chaining Multiple Continuation Functions: The return value of `future::then` method is a new future object, which is generated by the return value of the appended continuation function. Apparently, the new future object can be appended with another continuation function by calling `future::then` again. This feature allows the programmer to chain multiple continuation functions to mimic synchronous program. Consider the following example that performs asynchronous database and DNS queries sequentially:

```
// Query database.
future<db_res> db_fur=_db.lookup(_hash);
db_fur.then([this](db_res res){
  // Do something with the database query response.
  ...
  // Query DNS.
  future<dns_res> dns_fur=_dns.lookup(domain_name);
  return dns_fur;
}).then([this](dns_res res){
  // Do something with the DNS response.
  ...
});
```

Initially, an asynchronous database query `_db.lookup()` is called to acquire a future object `db_fur`, which contains the query response when it becomes available. A continuation function is then appended to `db_fur`. Inside this function, a DNS query `_dns.lookup()` is invoked to obtain a future

object `dns_fur`, which contains the DNS response upon available. `db_fur.then` returns a new future object, which is generated by `dns_fur`. Finally, a continuation function for handling the DNS response is chained to the future object returned by `db_fur.then`.

In the above example, the two queries appear to be synchronously executed but are in fact asynchronously performed, which demonstrates how future/promise paradigm mimics synchronous program.

5) Consolidated Error Handling: In the previous example, to catch exceptions raised during database and DNS queries, we can simply append another continuation function as follows:

```
... // Same as the previous example
}).then([this](dns_res res){
  // Do something with the DNS response.
  ...
}).then_wrapped([](future<> f){
  try{
    f.get();
  }
  catch(...){
    // catch whatever exceptions thrown
    // during database or DNS query
  }
});
```

Here, `f` may contain an exception that is generated either by the database query or by the DNS query. When an exception is raised inside the continuation function, it is propagated and pushed to the eventual future object, bypassing uncalled continuation functions in the chain. In the last continuation function appended to the eventual future object, the contained exception is thrown using `f.get` and caught inside a try-catch pair [38].

Leveraging this consolidated approach, we can effectively reduce redundant error handling code when developing asynchronous programs.

D. Bring Future/Promise to Dataplane

To implement our future/promise NF programming model, we choose Seastar [17] library as the base, due to the following reasons: (i) Seastar is a mature future/promise library written in C++. The C++-based implementation introduces minimum runtime overhead as compared to other future/promise libraries [15] implemented in functional programming languages with garbage collection. This is critical for building high-performance NFs. (ii) Seastar is integrated with DPDK and has a built-in user-space TCP/IP stack, rendering a feasible ground for building various NFs.

However, Seastar is originally designed for database servers [18]. It only exposes a socket-like interface for interacting with application-layer payload and has no interface for directly retrieving and sending raw network packets. Yet, an interface for manipulating raw network packets is crucial for implementing L4 NFs, such as firewall, NAT and IDS.

This leads to the design of *NetStar*, a framework for building asynchronous NFs using future/promise paradigm. *NetStar* is built on top of Seastar, it introduces a new async-flow interface

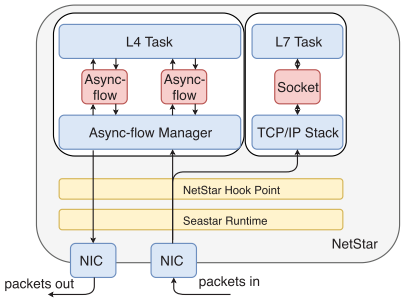


Fig. 2. An Overview of *NetStar*.

for handling raw network packets and combines the socket interface of Seastar to deal with application-layer traffic. The details of *NetStar* framework are covered in the following sections.

IV. The *NetStar* Framework

NetStar can build various L4 NFs that directly process raw network packets, including IDS, firewall and NAT. The capability of *NetStar* is further extended to build NFs that handle L7 application payload, including proxies that forward application messages between clients and servers. With future/promise paradigm, NFs built by *NetStar*can easily interact with external services, like database and DNS service. For the current *NetStar* version, a single server instance running *NetStar* framework only supports one set of NF processing logic. To run multiple NFs, different server instances should be provisioned, each running a distinct NF. An overview of the *NetStar* framework is given in Fig. 2.

NetStar is integrated with Seastar's runtime module, which uses DPDK to fetch packets from NIC queues into the user space. The hook point is an intermediate layer which parses packet headers and forwards packets to upper layer according to the parsed results. When *NetStar* is used to implement an NF which handles L4 network packets (*i.e.*, transport-layer packets), the hook point is configured to forward the packets received by the runtime to the async-flow manager. For NFs that handle L7 application payload (*e.g.*, HTTP requests), the hook point forwards the flows to the user-space TCP/IP stack to extract application payload.

Some NFs may process both L4 and L7 traffic. For an IDS that carries out malware detection as in Sec. III-A, it handles both L4 traffic (the TCP flows being inspected) and L7 traffic (database and MHR queries). To satisfy the requirements of these NFs, the processing of L4 and L7 traffic can coexist in *NetStar* by configuring the hook point. The TCP/IP stack of *NetStar* is associated with a default IP address. The hook point can be configured to classify all the input packets according to their IP addresses, forward packets with matching destination IP addresses to the TCP/IP stack, and others to the async-flow manager. The scheduler of *NetStar* can concurrently schedule both the stack and the manager to process their input traffic.

A. Process L4 Traffic

We design a special *async-flow* interface to process L4 packets, which consists of an async-flow manager and several

async-flow objects. The manager classifies packets received from the hook point into different network flows and pushes each flow to each async-flow object. The async-flow object implements the NF processing logic inside a simulated packet processing loop, using the future/promise abstraction for asynchronous operations. We will discuss our detailed design of the async-flow interface in Sec. V.

B. Process L7 Traffic

To handle L7 application payload, we reuse Seastar's TCP/IP stack. The socket interface of this TCP/IP stack is a modern re-implementation of the traditional POSIX socket based on the future/promise abstraction. For each established TCP connection, the stack further exposes one input stream for receiving data and one output stream for sending data.

C. Thread Model

An NF built with *NetStar* uses a share-nothing thread model. Each NF runs multiple threads. Each thread is pined to a unique CPU core and creates a complete set of modules, including the async-flow interface (one async-flow manager and multiple async-flow objects), the TCP/IP stack and the NF logic. A thread never shares these modules with other threads. Incoming packets are distributed to different threads in a load-balanced fashion, by configuring RSS [39] on the incoming NIC port. In this way, performance overhead associated with thread scheduling and shared resource contention is avoided.

V. Async-Flow Interface

This section gives a detailed introduction to the async-flow interface. The async-flow interface is divided into a manager, which distributes flow packets to different async-flow objects; and async-flow objects, which can be used for implementing NF logic that need complex asynchronous interactions. Our major challenge when designing the async-flow interface is how to exploit the power of the future/promise abstraction while exposing easy-to-use interfaces for building NFs. We carefully design a simulated packet processing loop and use returned future objects to concatenate asynchronous operations, to address the challenge.

A. Async-Flow Manager

After a packet is delivered to the async-flow manager from the hook point, the manager first retrieves flow information from the packet header to build a flow key. By default, the flow key is based on the flow 5 tuple (source/destination IP addresses, source/destination port and protocol type) of each TCP/UDP packet. The manager uses the key to identify an async-flow object from a hash map: if a corresponding async-flow object is found, the received packet is sent to the async-flow object; otherwise, the packet belongs to a new flow, and the manager creates a new async-flow object for the key, and updates the hash map. The manager also configures the new async-flow object with interested events (file hash event) and core processing logic of the NF, using programming interfaces exposed by the async-flow object.

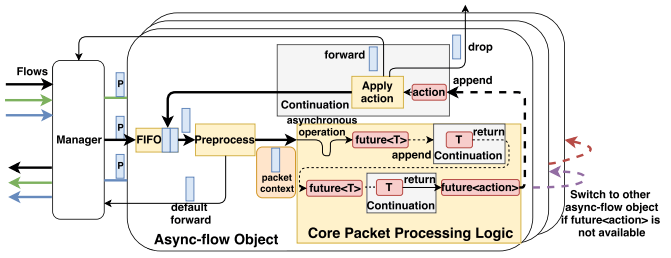


Fig. 3. Workflow of async-flow objects (P represents a packet).

B. Async-Flow Object

The async-flow objects are key components to implement asynchronous packet processing in an NF. We target the following objectives in its design. (1) **Ease of Use.** The exposed programming interfaces should be easy to use, especially for programmers to implement NFs. (2) **Full Processing Asynchrony.** After an async-flow object launches an asynchronous operation when processing a packet, it should yield its execution immediately to other async-flow objects, without blocking.

In *NetStar*, the programming interfaces exposed by the async-flow object include an interface for registering a special packet handler function that implements the core NF processing logic, and an interface for configuring what events to be reported to the registered packet handler function. The interfaces are easy to use: registration of a packet handler function is similar to implementing a packet handler in existing NF architectures [2], [40].

To achieve the second objective, we simulate a packet processing loop within each async-flow object via a series of future-continuation chains, pause a packet processing loop if any intermediate future object is unavailable, and only resume it when the future object becomes available.

An illustration of the workflow within and among async-flow objects is presented in Fig. 3. The rest of this section discusses how different components of async-flow object collaborate to form a simulated packet processing loop.

1) FIFO Queue: When a packet is sent from the manager to the async-flow object, the packet is first pushed into an FIFO queue and waits to be fetched by the packet processing loop.

2) Preprocessing: When a packet goes into the packet processing loop, it first goes through a pre-configured preprocessor to generate a set of events. For instance, if an in-sequence TCP packet arrives, the preprocessor may retrieve new payload from the reordering buffer to reconstruct the TCP byte stream and report it as an new event, along with arrival event of the TCP packet. In this way, some functionalities within the core processing logic can be offloaded to the preprocessor for simplification.

NetStar is equipped with four default preprocessors: a simple UDP preprocessor and a simple TCP preprocessor report packet arrival events; a TCP tracking preprocessor reports events on TCP packet arrival, TCP connection status change and the reconstruction of the TCP bytestream; a file extraction preprocessor extracts transmitted files from the byte stream and reports hash values of the files as events.

All four preprocessors are equipped with a timer to report flow time-out events, so that the async-flow object can gracefully shut down its packet processing loop. *NetStar* users can freely configure what preprocessor to use for different types of packets. When implementing an NF, the programmer can freely choose what preprocessors to use, so that only useful events are generated to the NF.

After preprocessing, a packet *context* object is constructed, containing the current packet and all the generated events. If none of the events matches any of the events that the NF is interested in (*e.g.*, an IDS may only be interested in reconstructed TCP byte stream, not the arrival of an out-of-order TCP packet), a default action is performed, *i.e.*, forwarding the packet out (to the next-hop NF or the flow destination), and another packet is fetched for preprocessing from the FIFO queue; otherwise, the packet context is sent to the core processing logic for further processing.

3) Core Packet Processing Logic: In the core processing logic, events from the packet context are retrieved and packet processing logic is executed with a series of asynchronous operations. To handle one asynchronous operation, an associated future object is obtained from a promise object, and a continuation function is appended to the future object. Upon the completion of the asynchronous operation, the promise object turns the future object into available state and invokes the appended continuation function, which carries out corresponding processing based on the received response and returns another future object for handling the next asynchronous operation. Multiple asynchronous operations can be handled using such a 'future/promise-continuation' chain.

Using malware detection in IDS as an example, the core processing logic includes two asynchronous operations: querying the local database and querying the MHR. A future object is first constructed to receive query response from the local database, and the database response is obtained as the concrete type of value in the future object. A promise is associated to turn the future into available state when the response returns. The appended continuation function is invoked then, and returns another future object, which receives query response form the MHR. The second future object obtains the MHR response as its concrete type of value and passes it to an appended continuation function, which may raise an alert in case that the response indicates the detection of a malware.

4) Final Future Object and Its Continuation Function: The last continuation function in the asynchronous 'future/promise-continuation' chain is forced to return a future object containing an action decision (*e.g.*, drop or forward the current packet under processing). This future object becomes available when all asynchronous operations in the core processing logic have been completed. A continuation function is appended to this future object, which receives the action decision from the future object and carries out the action accordingly. The current packet context is then destroyed, and the packet processing loop moves on to fetch another packet from the FIFO queue (if it is not empty), and recursively restarts itself.

5) Switching to Another Async-flow Object: After an async-flow object has issued an asynchronous operation, the current thread moves on to handle another async-flow object, *i.e.*, the packet processing loop in another async-flow object starts to fetch packets from its own FIFO queue and processes them.

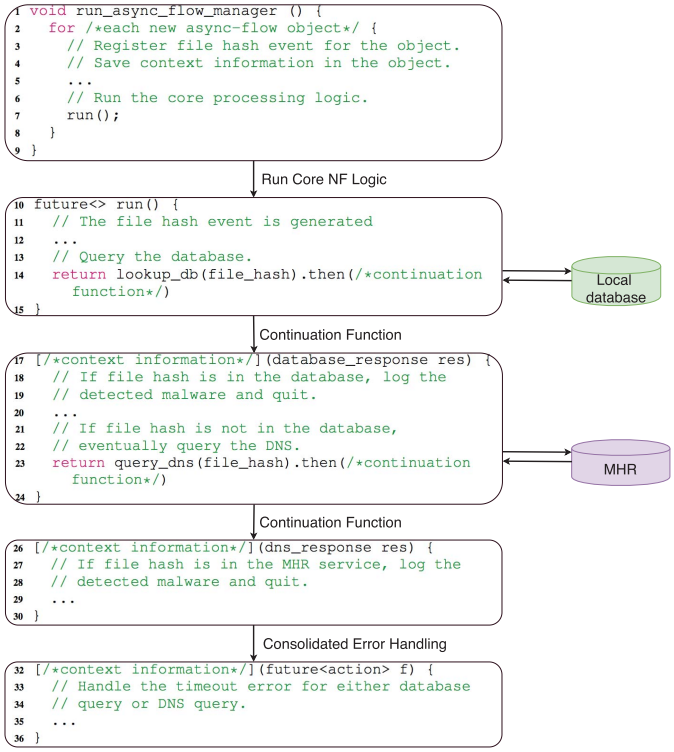


Fig. 4. Malware detector using *NetStar*.

C. Asynchronous Programming With NetStar: the IDS Example

Now we present the implementation of a malware detector using the async-flow interface designed above, which achieves similar functionality as in Fig. 1.

The pseudocode implementation of this malware detector is given in Fig. 4. `run_async_flow_manager` runs the async-flow manager, which constantly checks (lines 2) new async-flow object that is created for each new TCP flow. The manager registers the file hash event (line 3), saves context information (line 4) and launches the core processing logic (line 6 - 7) for each new async-flow object. The object then starts to detect malware by inspecting the flow packets.

First, the preprocessor reconstructs the TCP byte stream as it receives each new flow packet (code omitted from Fig. 4). After processing a packet, if the preprocessor detects a newly transmitted file, it calculates the hash value of this file and raises a the file hash event. In case that the preprocessor fails to detect a transmitted file, it only raises a packet arrival event, which is ignored by the core processing logic, and the packet is forwarded out directly.

After a file hash event is generated (line 11), The async-flow object issues a database query and obtains a future object containing the database response (line 14). A continuation function (line 17) is appended for checking the query result. If some malware is detected, the detection result is logged and

the async-flow object stops processing the flow by dropping all the following packets (line 18 - 19). Otherwise, the async-flow object moves on to query the MHR service by sending a DNS request (line 23). A continuation function (line 26) is then appended to check the result of the DNS response. If some malware is detected, the detection result is logged and the async-flow stops the processing (line 27 - 28). Otherwise, the async-flow object continues to process other flow packets by repeating the previous procedures. The code from line 32 to 36 handles potential exceptions generated during the database or DNS query in a consolidated fashion.

Comparison: The malware detector implemented with *NetStar* has the following advantages over that in Fig. 1. (1) *Simplified Implementation.* While some details are omitted in Fig. 4 for simplicity, the entire malware detection process can be concisely implemented in *NetStar* using only 29 lines of code in one function call. Our code mimics the sequential execution of database and DNS queries within a single function, instead of spreading the execution flow across multiple functions in different files. (2) *Simplified Context Management.* In our code, the context information is put directly into the async-flow object (line 4). Different continuation functions (lines 17, 26, 32) can easily visit the saved context by capturing a pointer to the context. Since the saved context is destroyed when the async-flow object stops running, the above code guarantees that the saved context is always alive when the continuation functions are being called. (3) *Consolidated Error Handling.* Instead of defining two error handling functions, our implementation consolidates the error handling logic within a single continuation function (lines 32 - 36). The programmer can be more focused on the core NF processing logic, while simply chaining another continuation function at the end for handling all errors that might be generated from the core code.

VI. Implemented NFs

We have built multiple representative NFs using *NetStar*.

A. NFs From the StatelessNF Paper [1]

We reimplement four NFs from the StatelessNF paper, *i.e.*, firewall, NAT, IDS and load balancer. Our implementation follows the pseudo-code logic in the paper, and leverages our async-flow interface. The major differences are: (1) we do not need to set up a dedicated polling thread for each worker thread to poll the NIC queue. Each thread in *NetStar* performs all the tasks including port-polling and packet-processing. (2) We use a fast in-memory key-value store, mica [41], which has a larger throughput than the RAMCloud database [42] used in StatelessNF. (3) In StatelessNF, a unique thread is dedicated to contact RAMCloud for storing the per-flow states; with *NetStar*, the async-flow objects running in each thread can use the thread-local mica client library to contact mica server concurrently.

B. An HTTP Reverse Proxy

We use the TCP/IP stack in *NetStar* to implement an HTTP reverse proxy, whose functionality is similar to HAProxy (see Sec. III-A) and TinyProxy [43]: it intercepts incoming TCP connections from clients, and relays HTTP requests to servers; it then receives HTTP responses from the servers and pushes them back to the clients.

C. An IDS That Inspects HTTP Payload

This IDS detects potential intrusion from the reconstructed HTTP request payload using our async-flow interface, which is more complicated than the IDS implemented following the StatelessNF paper.

D. A Malware Detector

as introduced in Sec. V-C. Due to its need to process mixed L4 and L7 traffic (Sec. IV), it utilizes both the async-flow interface and the TCP/IP stack of *NetStar*.

VII. Evaluation

In this section, we evaluate the performance of the various NFs built with *NetStar*. Even though future/promise paradigm can simplify asynchronous programming, using this paradigm adds additional overhead to dynamically allocate/deallocate special runtime objects on the heap (Sec. III-C). Therefore a natural question to ask is whether the performance of *NetStar* is good enough to approach line-rate processing. To answer this question, we design a series of experiments to evaluate the maximum throughput achieved by the *NetStar* NFs. We also compare *NetStar* with several NFs that are implemented with fast, low-overhead callback-based method to reveal the overhead associated with using future/promise abstraction. The second question involves the effectiveness of future/promise abstraction for simplifying the implementation. To answer this question, we adopt a similar quantitative methodology [9] used for evaluating F# [9] future/promise abstraction and compare the required lines of code for implementing the core processing logic between *NetStar* NFs and NFs built with callback method.

A. Methodology

1) Testbed: Our testbed consists of 5 servers: three Dell R430 servers, each equipped with one Intel Xeon E5-2650 CPU 2.30GHz with 10 physical cores and 48GB memory, and two Supermicro servers, each with one Intel Xeon E5-1620 CPU 3.50GHz with 4 physical cores and 32GB memory. All servers are equipped with two Intel X710 10Gbps NICs and they are connected through a Dell 10Gbps Ethernet switch. We divide the servers to run our NFs and traffic generators (flow sources and destinations).

2) Traffic Generation: We use two types of traffic generators. The first is a custom packet generator that we build on top of *NetStar*, which can generate 64-bytes UDP/TCP packets at 14Mpps (packets per second), *i.e.*, 10Gbps. The number of flows and the packet size for generation are adjustable. This generator is used to inject flows into NFs such as the packet forwarder (Sec. VII-B and Sec. VII-G) and the NFs from StatelessNF paper (Sec. VII-C). To measure packet processing latency, this generator tags each produced packet

with a generation time and computes the packet processing latency by an NF after receiving the packet back from the NF (in this case, source and destination of the flows are the same). The second traffic generator is the default HTTP benchmarking tool provided by Seastar. The tool runs in a client-server setting by sending a large number of HTTP requests from a client and receiving corresponding HTTP responses at the server. We modified this tool, including adding support for HTTP POST method for a client to upload files to the server and recording the HTTP transaction completion time, which is the time interval between sending HTTP request and receiving HTTP response. We use flows produced by this traffic generator to evaluate NFs such as the HTTP reverse proxy (Sec. VII-D), the IDS (Sec. VII-E) and the malware detector (Sec. VII-F).

3) Metric: We focus on three types of key performance metrics. (1) **Packet processing throughput** achieved by an NF, measured in the number of processed packets per second (Sec. VII-B, VII-C, VII-G), the number of processed HTTP requests per second (Sec. VII-D) and total bandwidth consumed by all the HTTP connections (Sec. VII-E, VII-F). (2) **Latency**, computed as average packet processing delay (Sec. VII-C) or the average HTTP transaction completion time (Sec. VII-D, VII-E, VII-F). (3) **Lines of code (LOC)** for implementing the core packet processing logic, meant for comparing the implementation difficulty using our future/promise based framework and the callback-based asynchronous programming.

4) Baselines: To compare with NFs implemented using *NetStar*, we implement several NFs using callback based asynchronous programming (except for HAProxy [3] and TinyProxy [43]), following the practice in existing NF implementation.

B. Micro Benchmarks

We first run a set of micro benchmarks to evaluate the performance of *NetStar* for basic packet processing and asynchronous operations.

1) Packet Processing Throughput: As a high-performance framework for building dataplane NFs, *NetStar* should have adequate performance and multi-core scalability for basic packet forwarding task. To evaluate this, we build a simple packet forwarder using *NetStar*, where the processing logic in each async-flow object is to forward all the received packets.

For comparison, we also build a baseline forwarder using DPDK [19], which classifies packets into different flows by checking their flow-5-tuple against a cuckoo hash table adopted from the BESS virtual switch [44], and passes packets belonging to a flow to its flow object for forwarding. Compared with *NetStar*, the baseline forwarder has no overhead for creating future objects and chaining continuation functions, making it run faster. We evaluate *NetStar* using small packets to better understand its packet forwarding performance, since larger packets can easily saturate the NIC bandwidth, making NIC a bottleneck rather than revealing any performance bottleneck in our framework.

Fig. 5a shows packet processing throughput when the NFs are running on a single thread. 10K UDP flows with different packet sizes are sent to an NF at the rate that is slightly larger than the maximum throughput achieved by the NF. The throughput achieved by *NetStar* is very close to the baseline.

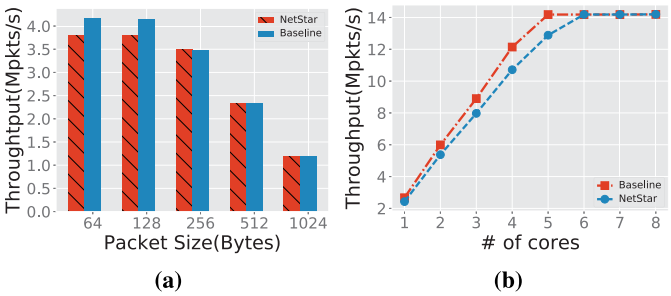


Fig. 5. Micro benchmarking on packet processing speed.

Fig. 5b illustrates the multi-core scalability when the NF runs with multiple threads on different CPU cores. 100K UDP flows with 64-byte packets at 10Gbps line rate are injected to test the NF. According to Fig. 5b, the throughput of *NetStar* is always close to the baseline. With 6 cores, *NetStar* achieves 10Gbps line rate.

Since *NetStar* carries out additional work for managing async-flow objects and future/promise objects, the single core throughput achieved by *NetStar* cannot match that achieved by a DPDK-enabled NF. However, due to the multi-core scalability brought by *NetStar*'s shared-nothing architecture, we believe that *NetStar* can still scale up to process input traffic at 40Gbps when running on a modern multi-core server.

2) Asynchronous Database Query: We next develop a simple NF, whose core processing logic is to read from and write to a mica database for each received packet. A write operation stores a key-value pair to the database, where the key is the packet's flow-5-tuple and the value is a 24-byte random array. A read operation retrieves a key-value pair from the database using the packet's flow-5-tuple. We also implement a similar NF using a callback-based framework built on top of DPDK: a callback function is registered with each database query, and is invoked when the response arrives.

Compared with the *NetStar*-based implementation, this framework imposes minimum runtime overhead when executing asynchronous operations: the registered callback is simply a function pointer without any heap allocation, whereas dynamic allocation and deallocation of future/promise object on the heap are involved in a future/promise-based framework.

We use 100K UDP flows with 64-byte small packets at 10Gbps line rate for testing. We vary the number of database read/write carried out by the NF when processing each packet. After receiving each packet, those read/write operations are consecutively carried out, before sending the packet out. Both *NetStar* and the callback implementation run with 10 threads. Fig. 6 shows that the packet processing throughput and the number of database queries that *NetStar* can achieve is very close to that of the callback-based implementation. The average performance gap under different database access patterns is 4.23% for packet throughput and 3.98% for database operations per second.

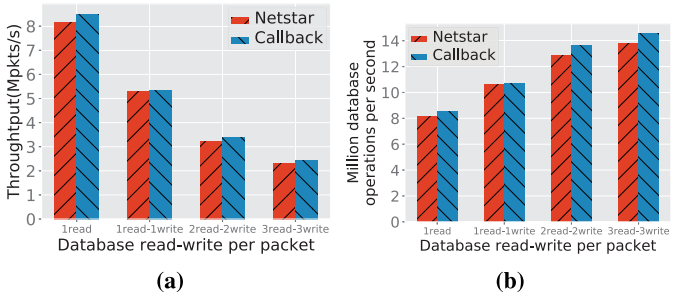


Fig. 6. Micro benchmarking on asynchronous DB queries.

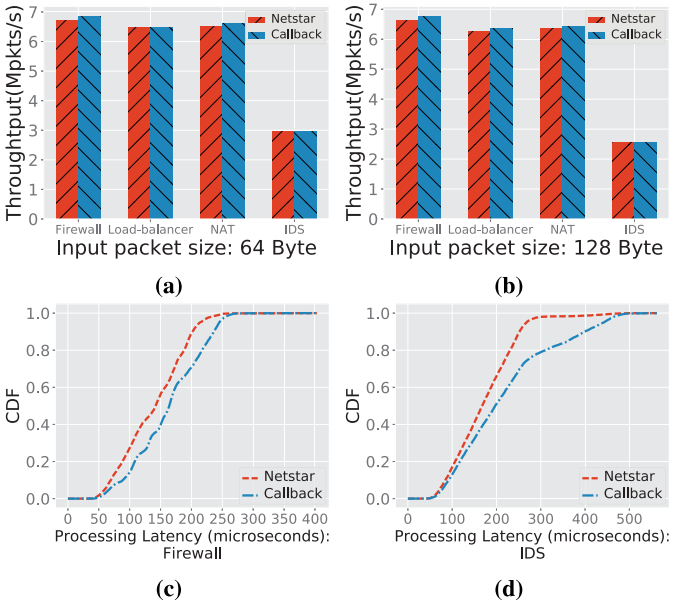


Fig. 7. Performance comparison: NFs from the StatelessNF paper.

C. NFs From the StatelessNF paper [1]

We compare our *NetStar* based implementation of the four NFs with callback-based implementation. We inject 100K TCP flows at 10Gbps to stress test the NFs and vary the size of the flow packets. Each NF runs using 10 threads on a server, and accesses the mica database on a different server.

In Fig. 7a and Fig. 7b, we observe that the packet processing throughput of our *NetStar* NFs is very comparable with the callback-based NFs. When more complicated packet processing logic is involved (*e.g.* intrusion detection in IDS) to process each packet besides executing asynchronous operations, the performance gap between *NetStar* and the callback-based implementation decreases to only 2%. We also measure the packet processing throughput when the packet size is 256, 512 and 1024 bytes respectively. With a larger packet size, the NFs (except the IDS) can approach a 10Gbps processing rate. For IDS, the rate can reach 6.78 Gbps when processing 1024 bytes packets.

Fig. 7c and Fig. 7d show the CDF of packet processing latency at the firewall and the IDS when the packet size is 64 bytes. The packet processing latency between *NetStar* and the baseline is close to each other.

TABLE I
LOC COMPARISON: NFS FROM THE STATELESSNF PAPER

NF	Callback	*NetStar*	Reduction %
Firewall	52(44+8)	41(34+7)	21.1%
Load balancer	82(66+16)	64(57+7)	21.9%
NAT	99(79+20)	80(73+7)	19.1%
IDS	50(42+8)	43(36+7)	14%

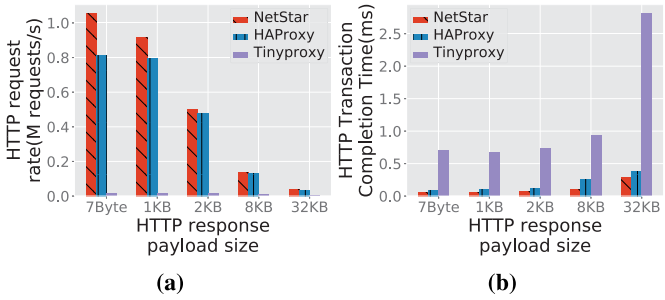


Fig. 8. Performance comparison: different proxies.

To compare the implementation difficulty, we list the LOC for implementing the core processing logic of the four NFs using *NetStar* and the callback framework in Table I. We only count the core processing code because a full-fledged implementation of each NF involves a large volume of auxiliary code on memory management, packet preprocessing and communication with mica. We can see that with the future/promise abstraction, the LOC can be reduced by as much as 21.9%.

In Table I, the total LOC for each NF is also represented as a sum between the LOC for basic packet processing functionality (left-hand-side of the + operator) and the LOC for error handling (right-hand-side of the + operator). Due to consolidated error handling in *NetStar*, only 7 lines of code are needed for handling all the errors in the four NFs.

D. HTTP Reverse Proxy

We compare our proxy implemented using *NetStar* with both HAProxy version 1.8 [3] and TinyProxy version 1.8.4 [43]. TinyProxy does not use a callback-based asynchronous design; instead, it creates a new thread to handle each TCP flow in a synchronous manner, and relies on the kernel scheduler to schedule different threads. Both HAProxy and TinyProxy use the TCP/IP stack of Linux kernel, whereas *NetStar* proxy uses the user-space TCP/IP stack of *NetStar*. Each proxy runs on 10 threads. We use the HTTP benchmarking tool to generate 200 connections that go through the proxy, and then keeps producing HTTP requests at the generator's maximal capability.

In Fig. 8a, we see that *NetStar* out-performs HAProxy by up to 20% and is way better than TinyProxy. As the payload size of HTTP response increases, the consumed bandwidth on the NIC gradually reaches 10Gbps, making the NIC a bottleneck and hence decreasing the performance gap. Fig. 8b shows that the smallest HTTP transaction completion time is achieved by *NetStar*.

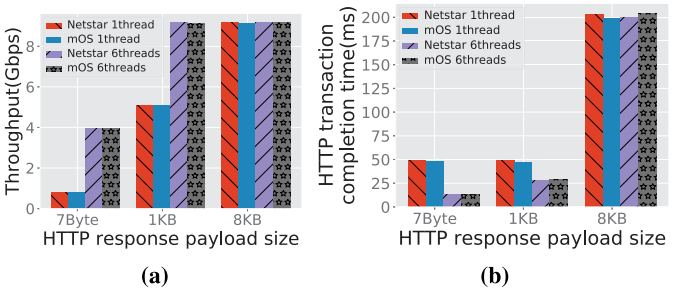


Fig. 9. Performance comparison: IDS.

The performance gain of *NetStar* over HAProxy is primarily due to the user-space TCP/IP stack used by *NetStar*, as the kernel network stack used by HAProxy introduces significant overhead for context switching [45]. To verify this, we also test the performance of *NetStar* using the same kernel network stack as HAProxy, and find that the performance of *NetStar* is worse than HAProxy when the payload size is smaller than 8KB.

Note that we implement our proxy by translating the processing logic in TinyProxy, as our future/promise based framework can easily mimic a synchronous execution flow. The a translated *NetStar* proxy is easy to implement, runs in full asynchrony and has superior performance even compared to HAProxy, as shown by the above results.

E. IDS

We compare the IDS built with *NetStar* and one built with mOS [23], which is one of the fastest frameworks for building middleboxes that process L7 application payload. mOS extensively uses the callback-based event-driven model. Our IDS implementation in mOS registers various callback functions to the mOS framework to obtain reconstructed TCP byte stream and parses HTTP requests for intrusion detection. We generate 24K concurrent HTTP connections to pass through each IDS. To test the scalability, the number of the threads used by each IDS is varied from 1 thread to 6 threads.

In Fig. 9, we can see that the performance of *NetStar* is very close to the mOS-based implementation. On the other hand, 493 lines of code are written to implement the core processing logic of our *NetStar* IDS, whereas 689 lines are used in the mOS-based implementation, achieving a 28% reduction. Together with the results discussed in Sec. VII-C, it shows that simplifying implementation using *NetStar* is a worthwhile choice especially when building NFs that require complex asynchronous operations.

F. Malware Detector

We compare our *NetStar* malware detector that queries an external database and a DNS server (both run in the same cluster as the malware detector) with a local malware detector that only visits a local hash table stored in its own memory. Both detectors are implemented using *NetStar*. We generate 1000 HTTP connections that pass through the detector and send files over these connections. We randomly populate the content of the file with malware. The size of the files is 8K and 32K bytes respectively. Both detectors run in one thread.

TABLE II
PERFORMANCE OF MALWARE DETECTORS

	Throughput (file size: 8k)	Throughput (file size: 32k)	HTTP transaction completion time (file size: 8k)	HTTP transaction completion time (file size: 32k)
Asynchronous Malware Detector	5.02 Gbps	6.19 Gbps	12.74 ms	41.33 ms
Local Malware Detector	5.86 Gbps	6.79 Gbps	10.77 ms	37.84 ms

In Table II, we can see that compared with the local detector, the *NetStar* detector experiences a 14% throughput drop when the size of the transmitted file is 8K and 8.8% throughput drop when the size increases to 32K. The process of accessing external database and DNS server adds a 1.97ms/3.48ms latency respectively.

We purposely compare our *NetStar* detector with a local detector, instead of a callback-based detector, in order to show the following: the performance of the *NetStar* detector is already comparable with a fast local detector, not to mention a callback-based detector; with *NetStar*, complicated asynchronous operations can be enabled on NFs without large performance drop, and the future/promise abstraction provided by *NetStar* renders easy implementation of the asynchronous operations.

G. Comparison With Coroutine

We compare *NetStar* with a coroutine-based NF framework. The coroutine-based framework is built on top of Seastar, leveraging Seastar's coroutine facility. For each new flow, a new coroutine is created and a stack with 4K size is allocated to the coroutine. A packet processing loop runs within the coroutine to process each input packet of the flow. Whenever a coroutine needs to wait for an asynchronous event, it saves stack information and yields its control to other coroutines in the NF. When the asynchronous event arrives, the coroutine waiting for the event is resumed by restoring the stack.

We compare the *NetStar* NF carrying out database queries in Sec. VII-B.2 with a coroutine-based implementation. Each NF runs in a single thread and reads the database once when processing each packet. Both of the two NFs are straightforward to implement with the two frameworks: the core processing logic of *NetStar* NF requires 18 lines of code, while that of coroutine-based NF needs 16 lines.

First, we configure the traffic generator to produce dynamically-generated UDP flows. For each second, 1K new UDP flows are generated and sent to the NFs under testing. Each UDP flow has a 10-second active time, resulting a total of 10K active UDP flows. A 714K pps throughput is achieved by the NF implemented with *NetStar*, while the throughput of the coroutine-based NF is 609K pps. Compared with *NetStar* NF, constant stack allocation and context switching of coroutine-based NF lead to a 14.7% performance loss.

Then we directly send 10K long-lasting UDP flows to the two NFs. We find that the *NetStar* NF achieves a 759K pps throughput while the coroutine-based NF has a 740K pps throughput, suffering a 2.5% performance degradation. Compared with the previous experiment, the coroutine stacks

are only allocated once at the beginning, and the major overhead comes from frequent context switching.

We can see that *NetStar* NF achieves better results regardless of the traffic pattern. Under a more realistic setting with highly-dynamic traffic, the *NetStar* NF is 14.7% faster than coroutine-based NF.

VIII. Discussions

We have shown that using *NetStar* framework can effectively simplify asynchronous programming in NFs, but the learning curve of future/promise abstraction may hinder further adoption of *NetStar*. Learning to use the *NetStar* indeed takes some efforts, as future/promise abstraction makes use of various C++11/14 features including move semantics, lambda functions and template meta programming. Our own experience is that, once a programmer has spent some time getting familiar with the future/promise abstraction, he can greatly improve his productivity when programming asynchronous code. There are also valuable text book [36] and online documentation [17], [20] that can simplify the learning process for the programmer. We believe that learning to use *NetStar* may produce higher reward for any interested programmer in the long run.

In addition, porting existing NF code to our *NetStar* framework is feasible, but may require some extra efforts on converting the callback-based programming interfaces to the future/promise abstraction, which usually involves exposing a new interface that returns a future object containing an asynchronous response. Once the concept of future/promise is mastered, this process can be made relatively easy.

The coroutines discussed previously are stackful coroutines. They maintain their own stacks and run synchronous code in asynchronous fashion. Besides stackful coroutines, there are stackless coroutines [46], as each coroutine has no stack and runs faster. Stackless coroutines are very similar to future/promise paradigm: they can only mimic synchronous program by chaining multiple callback functions together and save context information in special runtime object. However, to build real-world NF applications using stackless coroutines, one still needs a complete framework like *NetStar* that provides both high-performance packet IO, user-space TCP/IP stack and other management functionalities.

Even though C++ standard library [47] has provided future/promise paradigm, it does not support chaining arbitrary number of continuation functions as in *NetStar*. When it comes to building complex asynchronous NFs, *NetStar* is a better choice over C++ standard library.

Finally, this paper explores how to apply ideas in advanced programming abstractions to low-level programming tasks like processing dataplane packets. As NFs become more complex and involve more external interactions [1], [28], we believe that with the approach that we adopt in *NetStar*, we can make future NF software highly efficient, fully asynchronous, extremely robust and easy to implement.

IX. Conclusion and Future Work

This paper proposes *NetStar*, the first attempt to bring the future/promise abstraction to the NF dataplane for flow processing. To fully utilize the power of future/promise, we carefully design an async-flow interface that chains a number of future/promise and continuation functions for efficiently handling a series of asynchronous operations. Using *NetStar*, asynchronous programming in NFs is made easy, while good packet processing performance is still guaranteed. Our extensive evaluation shows that *NetStar* can effectively simplify asynchronous programming asynchronous in NFs, while easily achieve line-rate packet processing for NFs.

For future work, we plan to reimplement *NetStar* using Rust [48] programming language. Rust is system programming language with zero runtime overhead just like C/C++, but has a powerful type system that eliminates many runtime bugs such as invalid memory access and race condition. We are aware that Rust has a more efficient future/promise implementation than that of Seastar library. By porting to Rust, the performance and robustness of *NetStar* can be further improved.

Acknowledgment

The authors would like to thank all the reviewers for their helpful comments on improving the paper. The major part of this work was done when Jingpu Duan was with The University of Hong Kong.

References

[1] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2017, pp. 97–112.
[2] (2018). *The Bro Network Security Monitor*. [Online]. Available: https://www.bro.org/
[3] (2018). *HAProxy*. [Online]. Available: https://www.snort.org/
[4] (2018). *Project Clearwater*. [Online]. Available: http://www.projectclearwater.org/
[5] (2018). *RFC 3953—Telephone Number Mapping (ENUM) Service*. [Online]. Available: https://tools.ietf.org/html/rfc3953
[6] (2018). *Bro Scripting Tutorial*. [Online]. Available: https://www.bro.org/sphinx/scripting/
[7] K. Gallaba, A. Mesbah, and I. Beschastnikh, "Don't call us, we'll call you: Characterizing callbacks in javascript," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Oct. 2015, pp. 1–10.
[8] K. Kambona, E. G. Boix, and W. De Meuter, "An evaluation of reactive programming and promises for structuring collaborative Web applications," in *Proc. 7th Workshop Dyn. Lang. Appl.*, 2013, p. 3.
[9] D. Syme, T. Petricek, and D. Lomov, "The F# asynchronous programming model," in *Proc. Int. Symp. Pract. Aspects Declarative Lang.*, 2011, pp. 175–189.
[10] (2018). *Tornado Web Framework*. [Online]. Available: http://www.tornadoweb.org/
[11] (2018).*Rethinkdb Blog Post*. https://rethinkdb.com/blog/improving-a-large-c-project-with-coroutines/%
[12] (2018). *Coroutines—Boost C++ Libraries*. [Online]. Available: https://www.boost.org/doc/libs/1_63_0/libs/coroutine/doc/html/coroutine%/intro.html
[13] P. Li and S. Zdancewic, "Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives," in *Proc. 28th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI 07)*, 2007, pp. 189–199.
[14] K. Claessen, "A poor man's concurrency monad," *J. Funct. Program.*, 1999, pp. 313–323.
[15] J. Vouillon, "Lwt: A cooperative thread library," in *Proc. ACM SIGPLAN Workshop ML*. 2008, pp. 3–12.
[16] (2018). *Seastar Coroutine Implementation*. [Online]. Available: https://github.com/scylladb/seastar/blob/master/core/thread.hh
[17] (2018). *Seastar Project*. [Online]. Available: http://www.seastar-project.org/
[18] (2018). *Scylla Database*. [Online]. Available: http://www.scylladb.com/

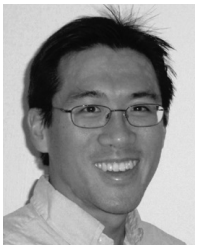
[19] (2018). *Intel Data Plane Development Tool Kit*. [Online]. Available: http://dpdk.org/
[20] (2018). *Netstar Source Code*. [Online]. Available: https://github.com/netstar-project/netstar
[21] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *Proc. 12th USENIX Symp. Oper. Syst. Design and Implement. (OSDI)*, 2016, pp. 203–216.
[22] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, 2014, pp. 87–95.
[23] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park, "mOS: A reusable networking stack for flow monitoring middleboxes," in *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2017, pp. 113–129.
[24] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2018, pp. 299–312.
[25] A. Gember-Jacobson *et al.*, "OpenNF: Enabling innovation in network function control," in *Proc. SIGCOMM Comput. Commun. Rev.*, 2014, pp. 163–174.
[26] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *Proc. 10th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2013, pp. 227–240.
[27] Z. A. Qazi, M. Walls, A. Panda, V. Sekar, S. Ratnasamy, and S. Shenker, "A high performance packet core for next generation cellular networks," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2017, pp. 348–361.
[28] M. Bagaa, T. Taleb, A. Laghrissi, A. Ksentini, and H. Flinck, "Coalitional game for the creation of efficient virtual core network slices in 5g mobile systems," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 3, pp. 469–484, Mar. 2018.
[29] A. Laghrissi, T. Taleb, and M. Bagaa, "Conformal mapping for optimal network slice planning based on canonical domains," *IEEE J. Sel. Areas Commun.*, vol. 36, no. 3, pp. 519–528, Mar. 2018.
[30] T. Taleb, M. Bagaa, and A. Ksentini, "User mobility-aware virtual network function placement for virtual 5G network infrastructure," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2015, pp. 3879–3884.
[31] M. Bagaa, T. Taleb, and A. Ksentini, "Service-aware network function placement for efficient traffic handling in carrier cloud," in *Proc. IEEE Wireless Commun. Netw. Conf.*, Apr. 2014, pp. 2402–2407.
[32] A. Laghrissi, T. Taleb, M. Bagaa, and H. Flinck, "Towards edge slicing: VNF placement algorithms for a dynamic & realistic edge cloud environment," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2017, pp. 1–6.
[33] M. Bagaa, T. Taleb, A. Laghrissi, and A. Ksentini, "Efficient virtual evolved packet core deployment across multiple cloud domains," in *Proc. IEEE Wireless Commun. Netw. Conf. (WCNC)*, Apr. 2018, pp. 1–6.
[34] R. A. Addad, T. Taleb, M. Bagaa, D. Dutra, and H. Flinck, "Towards modeling cross-domain network slices for 5G," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2018, pp. 1–6.
[35] (2018). *Malware Hash Registry*. [Online]. Available: http://www.team-cymru.org/MHR.html
[36] S. Meyers, *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. Sebastopol, CA, USA: O'Reilly Media, 2014.
[37] (2018). *C++ Lambda Expressions*. [Online]. Available: http://en.cppreference.com/w/cpp/language/lambda
[38] (2018). *C++ Try-Block*. [Online]. Available: http://en.cppreference.com/w/cpp/language/try_catch
[39] (2018). *Receive Side Scaling on Intel Network Adapters*. [Online]. Available: https://www.intel.com/content/www/us/en/support/articles/000006703/network-and-i-o/ethernet-products.html
[40] (2018). *Snort Intrusion Detection System*. [Online]. Available: https://www.snort.org/
[41] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in *Proc. 11th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*. Seattle, WA, USA: USENIX Association, 2014, pp. 429–444. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentatio% n/lim
[42] J. Ousterhout *et al.*, "The RAMCloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, p. 7, 2015.
[43] (2018). *Tinyproxy*. [Online]. Available: https://tinyproxy.github.io/
[44] (2018). *BESS: Berkeley Extensible Software Switch*. [Online]. Available: https://github.com/NetSys/bess
[45] E. Jeong *et al.*, "mTCP: A highly scalable user-level TCP stack for multicore systems," in *Proc. 11th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2014, pp. 489–502.
[46] (2018). *Stackless Coroutine*. [Online]. Available: https://github.com/jbandela/stackless_coroutine
[47] (2018). *std::future*. [Online]. Available: https://en.cppreference.com/w/cpp/thread/future
[48] (2018). *The Rust Programming Language*. [Online]. Available: https://www.rust-lang.org/

Jingpu Duan received the B.E. degree from the Huazhong University of Science and Technology in 2013 and the Ph.D. degree from The University of Hong Kong in 2018. He is currently an Assistant Researcher with the SUSTech Institute of Future Networks, Southern University of Science and Technology. His research interests include designing and implementing high-performance networking systems.

Xiaodong Yi received the B.E. degree from the Department of Computer Science, Huazhong University of Science and Technology, in 2017. He is currently pursuing the Ph.D. degree with the Department of Computer Science, The University of Hong Kong. His research interests include network function virtualization, and GPU and deep learning.

Junjie Wang received the B.E. degree from the Department of Computer Science, Huazhong University of Science and Technology, in 2018. He is currently pursuing the Ph.D. degree with the Department of Computer Science, The University of Hong Kong. His research interests include network function virtualization and operating systems.

Chuan Wu received the B.E. and M.E. degrees from Tsinghua University, China, in 2000 and 2002, respectively, and the Ph.D. degree from the University of Toronto, Canada, in 2008. She is currently an Associate Professor with the Department of Computer Science, The University of Hong Kong. Her research interests include cloud computing, network function virtualisation, and distributed machine learning.



Franck Le received the Ph.D. degree from Carnegie Mellon University and the Diplome d'Ingenieur from the Ecole Nationale Superieure des Telecommunications de Bretagne in France. He is currently a Researcher with the IBM T. J. Watson Research Center.