

BF-Runtime gRPC User's Guide

Introduction:	2
Version:	3
gRPC:	3
Protobuf:	3
APIs and Messages:	3
Messages:	3
Entity:	4
TableKey:	4
TableData:	6
DataField Type	7
TableEntry:	7
TableUsage:	8
TableAttribute:	8
IdleTable:	9
EntryScope:	9
TableOperation:	9
APIs (RPC calls)	10
Write:	10
Atomicity:	11
CONTINUE_ON_ERROR :	11
ROLLBACK_ON_ERROR:	11
USAGE GUIDELINES:	12
Read:	12
StreamChannel	13
DigestList	15
Subscribe	16
IdleTimeoutNotification	16
SetForwardingPipelineConfigResponse	16
PortStatusChgNotification	17
SetForwardingPipelineConfig	18
SetForwardingPipelineConfigRequest	18
Mastership	19
Refresh	19
Device lockdown	19

Example Scenarios	20
GetForwardingPipelineConfig	24
Non-P4 tables	25
Steps to connect to the gRPC server	25
Error Handling	26

Introduction:

BF-Runtime refers to the gRPC based RPC service exposed for the purposes of interacting and managing Barefoot switching devices. For more information please refer to the BF-Runtime API guide.

gRPC is a RPC framework which enables clients to execute methods on a remote server. For more information please refer to [gRPC](#) weblink.

The set of APIs available via this framework is based on a “.proto” file, which is the gRPC Interface Description Language (IDL). This file is part of the Barefoot SDE, under the name “bfruntime.proto”

For more information on this IDL, please refer to [PROTO](#) weblink.

The .proto file is fed through protoc compiler to generate the server and client implementations for the languages of choice. Barefoot SDE generates the server implementation in C++ and a sample client in Python. The SDE, when compiled using appropriate flags to enable gRPC and BF-RT will spawn a thread for the gRPC server, which listens in on a TCP port for client requests and communicates to the client over this port.

This documentation describes the usage and best practices of the BF-RT gRPC API assuming a C++ client. Note that based on the language of choice for the client, necessary modifications will have to be done, however much of the usage documentation will shed light on the use model regardless of the client language.

Version:

gRPC:

version = 1.17.0

To check ->

```
pip freeze | grep "grpc"
```

Protobuf:

Protobuf Python version = 3.6.1

To check->

```
pip freeze | grep "proto"
```

Protobuf C++ lib and protoc version = 3.6.1

To check->

```
protoc --version
```

APIs and Messages:

This section describes the various RPC APIs that BFRuntime exposes. The API's arguments are zero or more number of gRPC messages, which are declared and referenced in the APIs and the response is zero or one gRPC message.

Messages:

Messages are gRPC constructs to represent structures i.e.. They are used to represent a grouping of related fields. BFRuntime takes zero or more of these messages as arguments and return zero or one such message as responses. This section describes the messages that BFRuntime uses and exposes. One or more of these messages are composed to form the arguments of RPCs. Such messages are described in the context of the RPC itself for better clarity.

Entity:

Entity is a message representing the union of all the entities that an API can operate upon. The purpose of BFRuntime is to operate (mostly read or write) upon various entities.

The entity message is declared in the proto file as follows

```
message Entity {  
  oneof entity {  
    TableEntry table_entry = 1;  
    TableUsage table_usage = 2;  
    TableAttribute table_attribute = 3;  
    TableOperation table_operation = 4;  
    ObjectId object_id = 5;  
  }  
}
```

An entity can refer to any of the above types. Each of the types will be described in later sections.

One or more entities appear in write requests and read responses. There is a symmetry in here where a response from a read request can be directly passed along in a write request

TableKey:

TableKey is a message representing the Key of a table. Keys are made up of zero or more number of key fields. The TableKey message is defined in the proto file as follows

```
message TableKey { repeated KeyField fields = 1; }
```

Each KeyField is in-turn defined in the proto file as follows

```
message KeyField {  
  uint32 field_id = 1;  
  
  // Matches can be performed on arbitrarily-large inputs; the  
  protobuf type  
  // 'bytes' is used to model arbitrarily-large values.  
  message Exact { bytes value = 1; }
```

```

message Ternary {
    bytes value = 1;
    bytes mask = 2;
}

message LPM {
    bytes value = 1;
    int32 prefix_len = 2; // in bits
}

// A Range is logically a set that contains all values numerically
// between
// 'low' and 'high' inclusively.
message Range {
    bytes low = 1;
    bytes high = 2;
}

oneof match_type {
    Exact exact = 2;
    Ternary ternary = 3;
    LPM lpm = 4;
    Range range = 5;
}
}

```

Each KeyField is identified by a field-id. The field-id is unique within the keyfields of a given table.

Each KeyField can be of the following types (which fall out directly from the type of match that can be expressed in P4)

1. Exact:

This is for key fields which have to produce an exact match from the key in the table in order to produce a match. Such keyfields just have a value which is expressed as bytes.

2. Ternary:

This is for key fields which are used alongside a specified mask and then used in comparison process to produce a match. Such keyfields have a value and a mask, both of which are expressed as bytes.

3. LPM (Longest Prefix Match):

These key fields are similar to ternary key fields, except for how the mask is specified. The mask is specified as an integer representing the prefix length of the value to be used for matching. The value is expressed as bytes, the prefix length is expressed as an integer.

4. Range

These key fields specify a range of values that should match. The range is expressed using two integers, representing the low and high of the range.

TableData:

TableKey is a message representing the data of a table. Data is made up of zero or more number of data fields. The TableData message is defined in the proto file as follows

```
message TableData {
  uint32 action_id = 1;
  repeated DataField fields = 2;
}
```

TableData contains a action_id field which is the identifier of the action. There are tables for which action identifiers are applicable and there are tables for which they are not applicable. Tables for which action id is not applicable, this field of TableData is ignored.

TableData is composed of zero or more of Data Fields. Each DataField is in turn defined in the proto as follows

```
message DataField {
  uint32 field_id = 1;
  oneof value {
    bytes stream = 2;
    float float_val = 3;
    string str_val = 4;
    IntArray int_arr_val = 5;
    BoolArray bool_arr_val = 6;
    ContainerArray container_arr_val = 7;
    bool bool_val = 8;
  }
}
```

```

    }
    message IntArray { repeated uint32 val = 1; }
    message BoolArray { repeated bool val = 1; }
    message ContainerArray {
        message Container { repeated DataField val = 1; }
        repeated Container container = 1;
    }
}

```

Each DataField is identified by an id and has a value, the type of which depends on the field. This id is unique within a table and an action.

DataField Type

Each DataField can be one of the following types that BFRuntime exposes. The type of each data field is published in the bfrt.json. All of the data type are self-explanatory in nature and are typical of any well-known programming language.

1. Bytes : This type is for all data fields that are represented using a stream of bytes.
2. Float : This type is for all data fields that are represented using a floating point value.
3. String : This type is for all data fields that are represented using a string value. For such data-fields the list of supported strings are listed in bfrt.json.
4. IntArray : This type is for all data fields that are represented using an array of integers.
5. BoolArray : This type is for all data fields that are represented using an array of boolean values.
6. ContainerArray : This type is a repeated container. A container can have repeated DataFields recursively.
7. Bool : This type is for all data fields that are represented using a boolean value.

TableEntry:

TableEntry is one of the entity types. It refers to an individual entry in a BF-RT Table. The TableEntry message is declared in the proto file as follows

```

message TableEntry {
    uint32 table_id = 1;
    TableKey key = 2;
    TableData data = 3;
    bool is_default_entry = 4;
    TableReadFlag table_read_flag = 5;
}

```

```
TableModIncFlag table_mod_inc_flag = 6;
}
```

The table id in the message uniquely identifies the table id. The TableKey and TableData specifies the key and the data of the entry respectively.

A flag “is_default_entry” indicates if the table entry message is identifying a default entry of the table, in which case the TableKey field is immaterial.

The “table_read_flag”, in the context of a read operation indicates whether the read has to be performed from hardware or software.

The “table_mod_inc_flag”, in the context of a “modify incrementally” operation indicates whether the given data has to be added to or deleted from the existing table entry.

TableUsage:

TableUsage is an entity that can only be read. This refers to the usage of the tables i.e.. How many entries are currently in the table. TableUsage message is declared as follows in the proto file.

```
message TableUsage {
  uint32 table_id = 1;
  uint32 usage = 2;
  TableReadFlag table_read_flag = 3;
}
```

The table_id in the message indicates the table for which the usage is being read. The “usage” is populated with the usage of the table in response to the read. The TableReadFlag indicates whether the usage is on the number of entries that have been requested by the application to be added or the number of entries that have actually been written to the device. In most cases, they will be the same.

TableAttribute:

TableAttribute is an entity which refers to an applicable attribute of the table. This message is a union (which is represented using a “oneof” construct in the proto) of all possible table attributes. Each of the table attributes is explained in subsequent sections. The TableAttribute message is described as follows.

```
message TableAttribute {
  uint32 table_id = 1;
```

```

oneof attribute {
    IdleTable idle_table = 2;
    EntryScope entry_scope = 3;
    DynKeyMask dyn_key_mask = 4;
    DynHashing dyn_hashing = 5;
    ByteCountAdj byte_count_adj = 6;
    PortStatusChg port_status_notify = 7;
    StatePullIntvl intvl_ms = 8;
    PreDeviceConfig pre_device_config = 9;
}
}

```

The `table_id` in the message identifies the table for which the attribute is read or written.

IdleTable:

There are 2 modes in which IdleTime Attribute can be set. *POLL_MODE* and *NOTIFY_MODE*. If in *NOTIFY_MODE*, then `ttl_query_interval`, `max_interval`, `max_ttl`, `min_ttl` are required. In *POLL_MODE*, only `enable` is required.

```

message IdleTable {
    enum IdleTableMode {
        IDLE_TABLE_POLL_MODE = 0;
        IDLE_TABLE_NOTIFY_MODE = 1;
    }
    uint32 ttl_query_interval = 1;
    uint32 max_ttl = 2;
    uint32 min_ttl = 3;
    IdleTableMode idle_table_mode = 4;
    bool enable = 5;
}

```

EntryScope:

There are 3 different EntryScopes which can be set. One is for `gress(ALL/SINGLE)`, one for `pipe(ALL/SINGLE/user-defined)`, and one for `prsr(ALL/SINGLE)`

```

message EntryScope {
    Mode gress_scope = 1;
    Mode pipe_scope = 2;
    Mode prsr_scope = 3;
}

```

TableOperation:

TableOperation is an entity which can only be written. TableOperation refers to the operations that can be performed on an entire table. Each BF-RT table has a list of supported operations. The TableOperation message is described as follows

```
message TableOperation {  
    uint32 table_id = 1;  
    string table_operations_type = 2;  
}
```

The table operation is defined as a string. The table_id in the message identifies the table on which the operation is to be performed.

APIs (RPC calls)

Write:

Write API is a broad API that the client uses to send a write request to the server. Here is the IDL description of the write API

rpc Write(**WriteRequest**) returns **WriteResponse**

WriteRequest message is described as below

```
message WriteRequest {  
    TargetDevice target = 1;  
    uint32 client_id = 2;  
    repeated Update updates = 3;  
    enum Atomicity {  
        CONTINUE_ON_ERROR = 0;  
        ROLLBACK_ON_ERROR = 1;  
    }  
    Atomicity atomicity = 4;  
}
```

Update message is defined as follows

```
message Update {
  enum Type {
    UNSPECIFIED = 0;
    INSERT = 1;
    MODIFY = 2;
    // MODIFY_INC is used to add/delete the given data to/from the
    // existing table entry incrementally.
    MODIFY_INC = 3;
    DELETE = 4;
  }

  Type type = 1;
  Entity entity = 2;
}
```

Target specifies the granularity of the write request. The granularity depends on the table itself. Some tables have granularity at the pipe level of the device, and some tables have more fine grained granularity, for instance the parser instance.

Client-id identifies the client who is making the request. This field is used by the server to validate the request based on various conditions, which are described in this section.

Atomicity:

Atomicity dictates transactional semantics over a series of Write updates in a single Write request.

CONTINUE_ON_ERROR:

This is the default behavior. The batch is processed in a non-atomic manner from a dataplane point of view. Each operation within the batch must be attempted even if one or more encounter errors. Every dataplane packet is guaranteed to be processed according to table contents as they are between two individual operations of the batch, but there could be several packets processed that see each of these intermediate stages.

ROLLBACK_ON_ERROR:

Operations within the batch are committed to dataplane until an error is encountered. At this point, the operations must be rolled back such that both software and dataplane state is consistent with the state before the batch was attempted. The resulting behavior is all-or-none, except the batch is not atomic from a data plane point of view. Every dataplane packet is guaranteed to be processed according to table contents as they are between two individual operations of the batch, but there could be several packets processed that see each of these intermediate stages. NOTE that, this is not supported yet.

USAGE GUIDELINES:

The WriteRequest message encloses an Update, which can contain zero/one or more write requests. Hence the Write API has inherent support for grouping multiple write requests in one API call. It is always efficient to group write requests, whenever possible. BFRuntime Server uses underlying batching mechanisms to execute the group of write requests. Write requests can be grouped across different tables as well, since each write request update consists of a table-id.

Read:

Read API is used to read various entities. The read API is defined in the proto as follows

```
rpc Read(ReadRequest) returns (stream ReadResponse) {}
```

Read API takes in a ReadRequest and returns a ReadResponse.

ReadRequest is defined in the proto as follows

```
message ReadRequest {  
    TargetDevice target = 1;  
    uint32 client_id = 2;  
    repeated Entity entities = 3;  
}
```

Target and client-id are part of the read request similar to the write request. The core of the read request consists of zero/one or more number of entities. Just like write requests, read requests can be grouped and is efficient as well. The read response is defined as follows

```
message ReadResponse { repeated Entity entities = 1; }
```

Read response returns one or more entities, depending on the request itself.

StreamChannel

StreamChannel is a bi-directional streaming GRPC API. For more information on bi-directional streaming in GRPC, please refer to this [link](#).

A streaming API is different from a regular request-response based API. Streaming on the client side implies that the client can send a stream of requests without waiting for a response, the server can then send a single response. Similarly, server side streaming implies that the server can send a stream of responses for a single client request.

Bi-directional streaming is a combination of server side streaming and client side streaming, hence the client and server can send request/responses in any order. This stream logically behaves like two streams, which operate completely independently.

StreamChannel is as such a bi-directional streaming API. The API is defined in the proto as follows

```
rpc StreamChannel(stream StreamMessageRequest)
    returns (stream StreamMessageResponse) {}
```

It takes in a message of type StreamMessageRequest, which is defined in the proto as follows

```
message StreamMessageRequest {
    uint32 client_id = 1;
    oneof update {
        Subscribe subscribe = 2;
        DigestListAck digest_ack = 3;
    }
}
```

StreamMessage Request consists of a client_id which identifies the client who is sending the request and a Subscribe message.

Subscribe : This message, within the StreamMessageRequest is to be sent by the client for registering with the server. The subscribe message is defined as follows

```
message Subscribe {
    bool is_master = 1; // This field is used to indicate whether this client
```

```

// wants mastership of the device. Only a master can
issue
// any of the SetForwardingPipelineConfigRequest Actions.
// Other clients can only execute BIND. At a time, only
// one master can be present.
uint32 device_id = 2; // Device ID
message Notifications {
    bool enable_learn_notifications =
        1; // Enable learn digest notifications. These notifications are
        // (device, P4-program) based so these will be triggered only after
a
        // client binds to a program.
    bool enable_idletimeout_notifications =
        2; // Enable idletimeout notifications. These are on per table basis
and
        // hence (device, P4-Program) based so these will be triggered only
        // after a client binds to a program.
    bool enable_port_status_change_notifications =
        3; // Enable port status change notifications. These notifications are
        // device based and so they will be triggered whether a client is
        // bound to a program or not.
}
Notifications notifications = 3; // Contains which notifications need to be
// enabled for this client. Default value of
// these notifications are false.
.google.rpc.Status status = 4; // The controller doesn't populate this field.
// This is filled in by the GRPC server to
// indicate status of a subscribe request
}

```

It is required that each client calls this StreamChannel API as the first step and send a “Subscribe” message in the request. This creates the bi-directional stream, and is a way for the client to register with the server. The server uses the GRPC provided heartbeat mechanism to track the client’s existence. This bi-directional stream is broken by either the client issuing a “WritesDone” (please refer to GRPC public documentation on this), or if the client to server network connection breaks. In the event of the stream breaking, the server cleans up the client state. Thus this bi-directional stream defines the life-times of the client-server relationship.

A client wishing to be the master of the system will assert `is_master` to `True`. Only one master is allowed. The role of the master is described in later sections. Sending a Subscribe message is way for the client to get introduced to the server. Server maintains client-specific state, and this message serves the purpose of populating that state.

Notifications message contains the messages the client wishes to enable. Default value of these notifications are false. Note that for `idletimeout` and `port_status_change`, the corresponding

attributes also need to be set along with these flags if the client wishes to receive these notifications on the StreamChannel.

This bi-directional stream is used by the server to send asynchronous events to the client. Currently, asynchronous events such as learning and aging of flow entries etc are sent on this stream. This is sent as part of the response message, StreamMessageResponse which is defined as follows

```
message StreamMessageResponse {
  oneof update {
    Subscribe subscribe = 1; // This message is only used to let the
    server know
                                // of the existence of client with this
    client_id
    DigestList digest = 2;    // Learn Digest
    IdleTimeoutNotification idle_timeout_notification =
    3; // Idle timeout notification
    PortStatusChgNotification port_status_change_notification =
    4; // Port status change notification
    SetForwardingPipelineConfigResponse
    set_forwarding_pipeline_config_response =
    5; // Response for a SetForwardingPipelineConfigRequest
    is sent here
    // on the stream as well because all the active
    clients need to
    // be informed of a COMMIT_STARTED or COMMIT_FINISHED
  }
}
```

StreamMessage response can contain one of the following updates

DigestList

DigestList is defined as follows

```
message DigestList {
  // Identifies the digest extern instance
  uint32 digest_id = 1;
  uint32 list_id = 2;
  repeated TableData data = 3;
}
```

DigestList is the message that the server sends to report flow entries that are being learnt. Multiple flow entries are packed into one message in the form of a repeated TableData message within the DigestList.

Subscribe

A response to the subscribe request is sent using this response. The server just populates the "status" field of the subscribe message in response to the subscribe request.

IdleTimeoutNotification

IdleTimeoutNotification message is defined as follows

```
message IdleTimeoutNotification {  
    // Only "key" fields are required to be set in each TableEntry.  
    TargetDevice target = 1;  
    TableEntry table_entry = 2;  
}
```

IdleTimeOutNotification message is sent by the server on the bi-directional stream. One such message is sent for each flow entry that ages out. The message contains target, which specifies the pipe-id, along with the device id on which the entry aged out. The pipe-id sent in the message depends on the EntryScope set on the table. For a table with EntryScope of all pipe-lines, a pipe-id of ALL PIPES is sent, and for a table with EntryScope with single pipe-line, the individual logical pipe-id of the entry aging out is sent. Finally, for tables with EntryScope set to a user defined one, the lowest pipe-id of the scope in which the entry is aging out is sent. For instance, a user defined scope of pipe-id 1, 2 and 3, a pipe-id of 1 is sent.

The table entry aging out is sent in the message in the form of a TableEntry message.

SetForwardingPipelineConfigResponse

SetForwardingPipelineConfigResponse is defined as follows

```
message SetForwardingPipelineConfigResponse {  
    SetForwardingPipelineConfigResponseType  
        set_forwarding_pipeline_config_response_type = 1;
```

```
}
```

SetForwardingPipelineConfigResponseType is in turn defined as follows

```
enum SetForwardingPipelineConfigResponseType {  
    WARM_INIT_STARTED = 0; // WARM_INIT_STARTED indicates a successful  
                             // WARM_INIT_BEGIN. This is issued in case  
of  
                             // VERIFY_AND_WARM_INIT_BEGIN and  
                             // VERIFY_AND_WARM_INIT_BEGIN_AND_END  
  
    WARM_INIT_FINISHED = 1; // WARM_INIT_FINISHED indicates a  
successful  
                             // WARM_INIT_END. This is issued in case of  
                             // VERIFY_AND_WARM_INIT_BEGIN_AND_END and  
                             // WARM_INIT_END  
}
```

Even though set forwarding pipeline request is a separate RPC, the response is sent to all the clients on this bi-directional stream to let clients know that the system is beginning to going through a warm init and its completion. Please refer to the section on set forwarding pipeline for more details.

PortStatusChgNotification

PortStatusChgNotification is defined as follows

```
message PortStatusChgNotification {  
    // Only "key" fields are required to be set in each TableEntry.  
    TableEntry table_entry = 1;  
    bool port_up = 2;  
}
```

This message is sent by the server to convey the operational status of the port. The port is represented by the TableEntry and the status of (up/down) via a boolean flag.

SetForwardingPipelineConfig

SetForwardingPipelineConfig is an rpc exposed to push P4 program(s) and related collaterals to a switch, in order to have the switch running the desired P4 program(s)

SetForwardingPipelineConfigRequest consists of a device_id, a client_id, an Action and detailed config of the device.

SetForwardingPipelineConfigRequest

Action:

- **BIND** : Default Action. Only binds the client to the program specified in the p4_name. One client can bind to only one program. One program can have only one client as of now. Even in case of multiple programs on a single device, BIND requires just one program's config msg. If multiple repeated forwarding_pipeline_config msgs are sent as part of this request, then google.rpc.INVALID_ARGUMENT is sent. If a client doesn't BIND, then it can only access SetForwardingPipelineConfigRequest, GetForwardingPipelineConfigRequest and StreamMessageRequest RPCs. Read and Write RPCs are not allowed for non-bound clients
- **VERIFY(Master)**: Verifies whether this config is valid or not. Upon failure or incomplete config in the msg, google.rpc.Code::INVALID_ARGUMENT is sent.
- **VERIFY_AND_WARM_INIT_BEGIN(Master)**: Verifies the config and then begins warm_init with this config. This does not modify the forwarding state of the device. However, any subsequent Read / Write requests must refer to fields in the new config. Returns an INVALID_ARGUMENT error if the forwarding config is not provided or if the provided config cannot be realized.
- **VERIFY_AND_WARM_INIT_BEGIN_AND_END(Master)**: Verifies, starts warm_init and then initiates warm_init_end on the switch. The existing forwarding state is reset. Returns an INVALID_ARGUMENT error if the forwarding config is not provided or if the provided config cannot be realized.
- **WARM_INIT_END(Master)**: Issues a warm_init_end. If forwarding_pipeline_config contains nothing, or if no WARM_INIT_BEGIN was previously called on the device with a valid config, then google.rpc.Code::INVALID_ARGUMENT is sent. The forwarding state in the target is updated by replaying the write requests to the target device since the last config was saved by the client.

- **RECONCILE_AND_WARM_INIT_END(Master)**: Try and reconcile with the current config and preserve existing forwarding state. Minimal traffic loss is tried to be incurred in this scenario. Not supported currently. google.rpc.Code::UNIMPLEMENTED is sent.

Client_id : A client in order to gain “master” privileges needs to set *is_master* true while subscribing. Non-MasterClients can only issue BIND requests. While any of the Master functions are underway, the device is locked and service to any of Read / Write / GetForwardingPipeline / SetForwardingPipeline are not allowed. Only subscribe messages are allowed

Forwarding_pipeline_config : Contains the configuration for the entire device. Each msg in this repeated field contains configuration for one P4 program.

Base_path : This is the base_path on the switch where the clients have an option to store all the data-files. If empty, then the current directory is used and everything is truncated-overwritten.

Mastership

A master client can issue special SetForwardingPipelineConfig requests like VERIFY_AND_WARM_INIT_BEGIN and WARM_INIT_END. Only one client can be the master at a time. Master client can also be kicked out when *refresh* is going on if it was bound to a program which is no longer valid.

Refresh

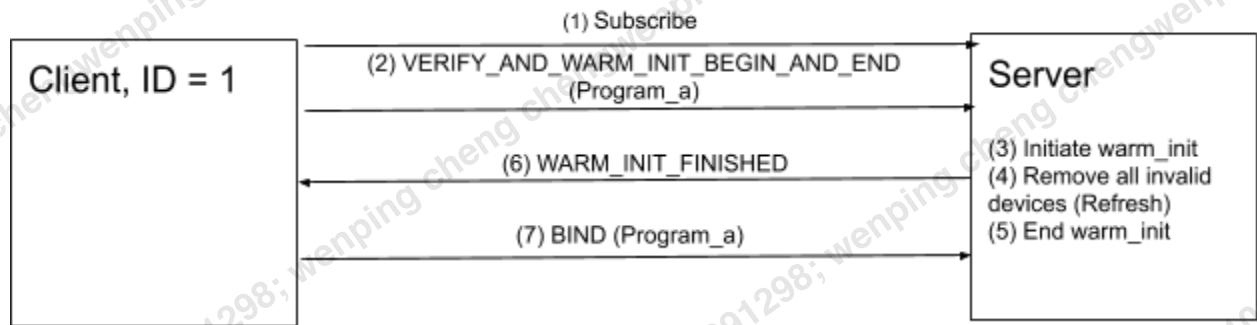
After server receives a WARM_INIT_BEGIN (WARM_INIT_BEGIN is a part of VERIFY_AND_WARM_INIT_BEGIN and VERIFY_AND_WARM_INIT_BEGIN_AND_END), all “invalid” clients are refreshed out. Upon removal of “invalid” clients, only the clients bound to invalid P4 programs are kicked out with a grpc::CANCELLED error. Invalid P4 programs are determined by the name of the new program against the old program. Comparison of the data files are not done in order to determine invalidity. Clients which are not bound to any program are not kicked.

Device lockdown

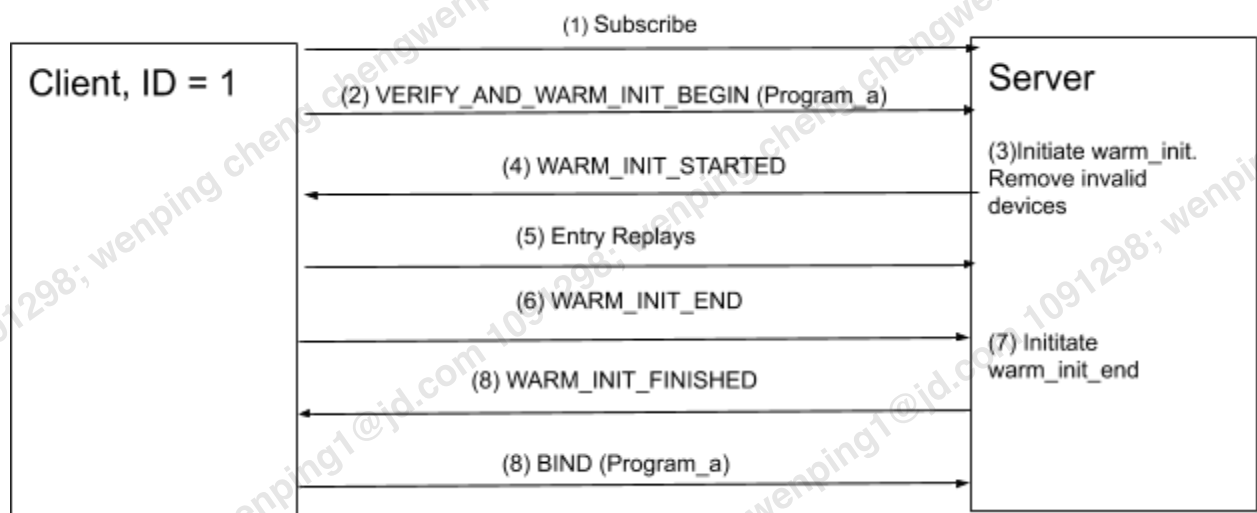
While any of the master functions are underway, the entire device is in an exclusive lock state and cannot service any RPCs other than StreamMessageRequest. An error message with grpc::UNAVAILABLE is sent upon failure to acquire this non-blocking lock by other RPCs. If a master function tries to acquire this lock and fails, then it blocks until it can acquire lock ownership.

Example Scenarios

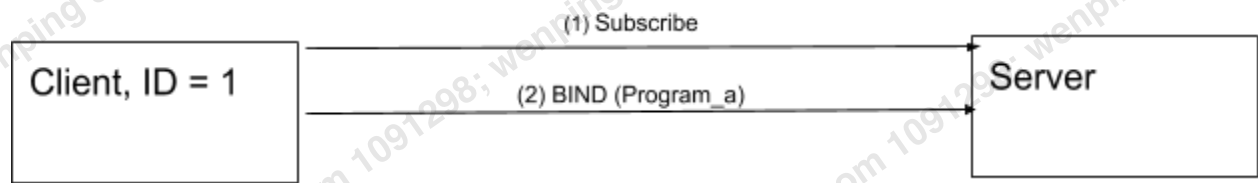
Case 1: Setting one P4 on all pipelines without replay.



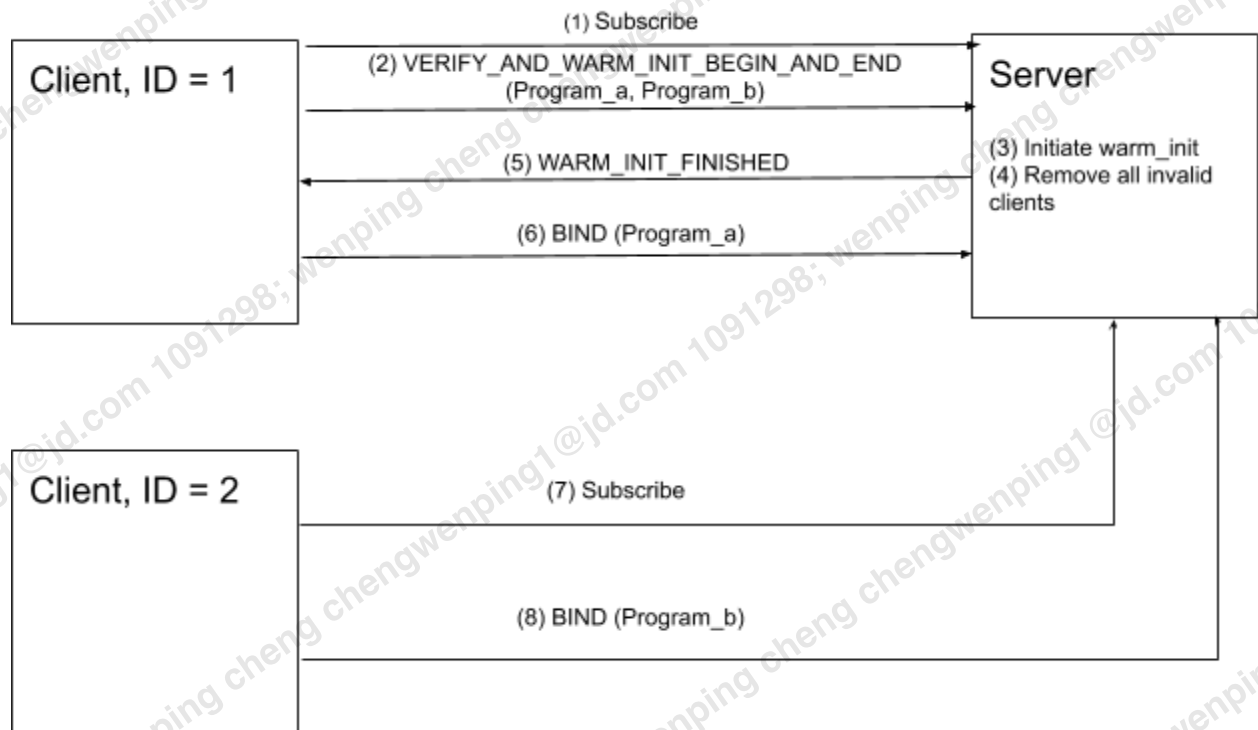
Case 2: Setting one P4 on all pipelines with replay.



Case 3: Switch was already started with a P4 Program. A client doesn't want to change the ForwardingPipelineConfig but only wants to BIND to a P4 Program. (Previously supported scenario, ensuring backward compatibility)

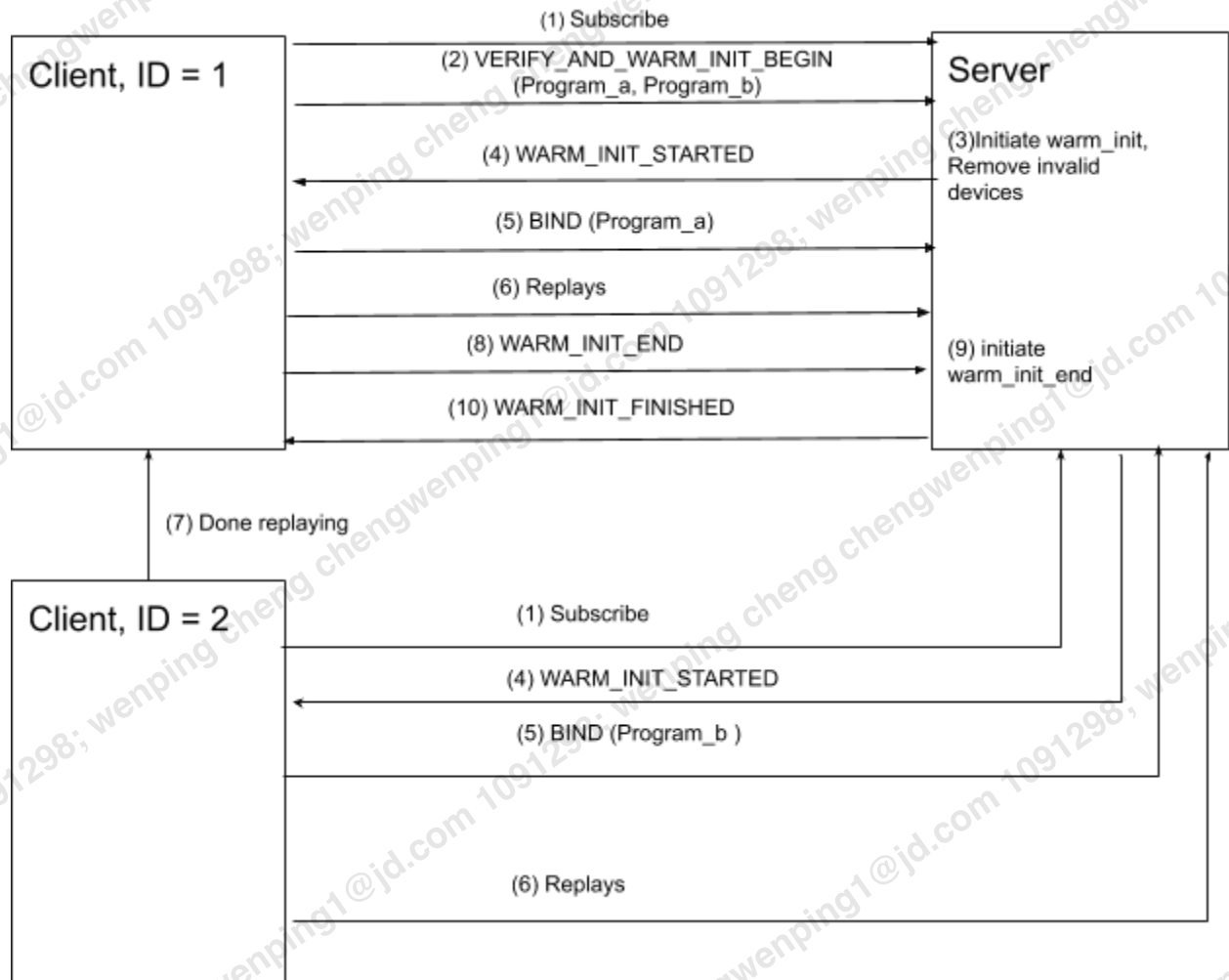


Case 4: Setting 2 P4 programs on 2 different scopes without replay



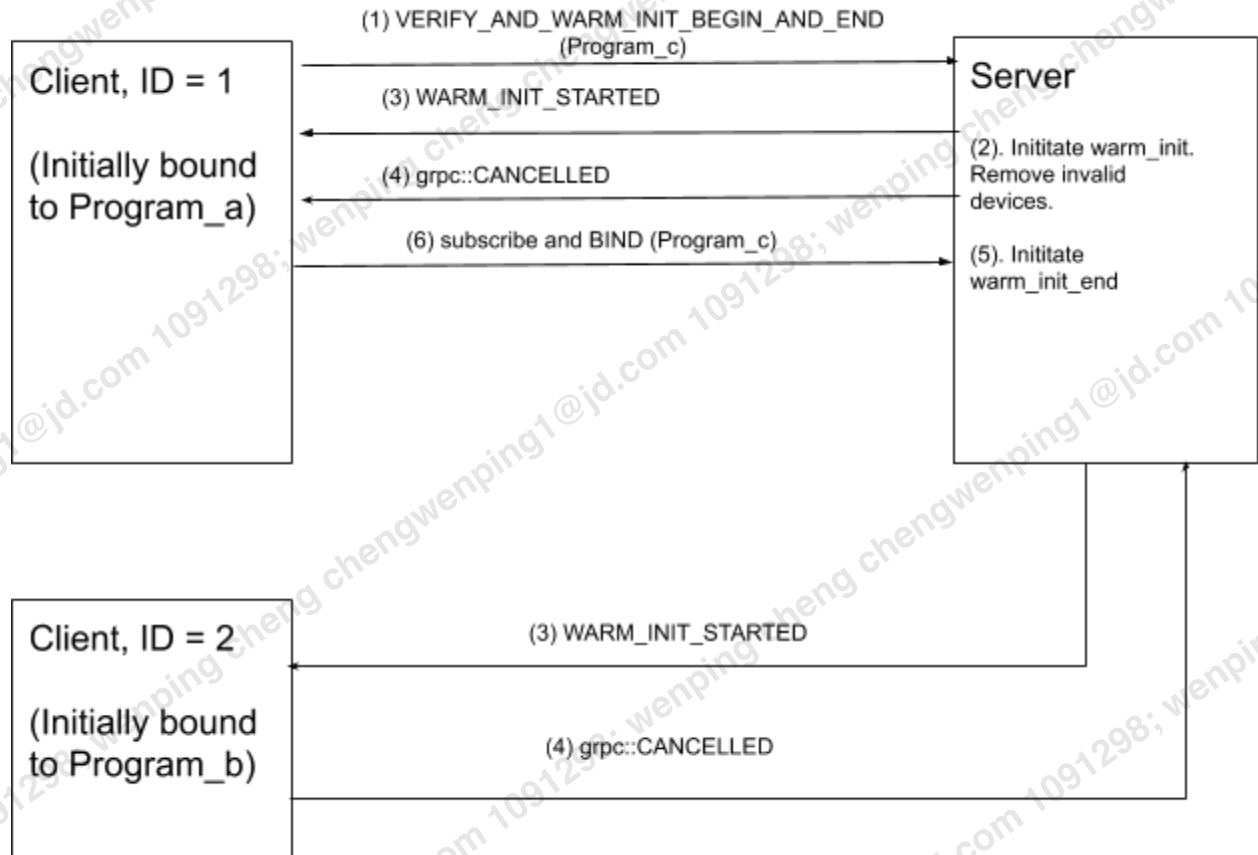
In the example case described above, both the clients are not attached to any program in the beginning. Client-ID 2 is Binding after the device is up in this case.

Case 5: Setting 2 P4 programs on 2 different scopes with replay

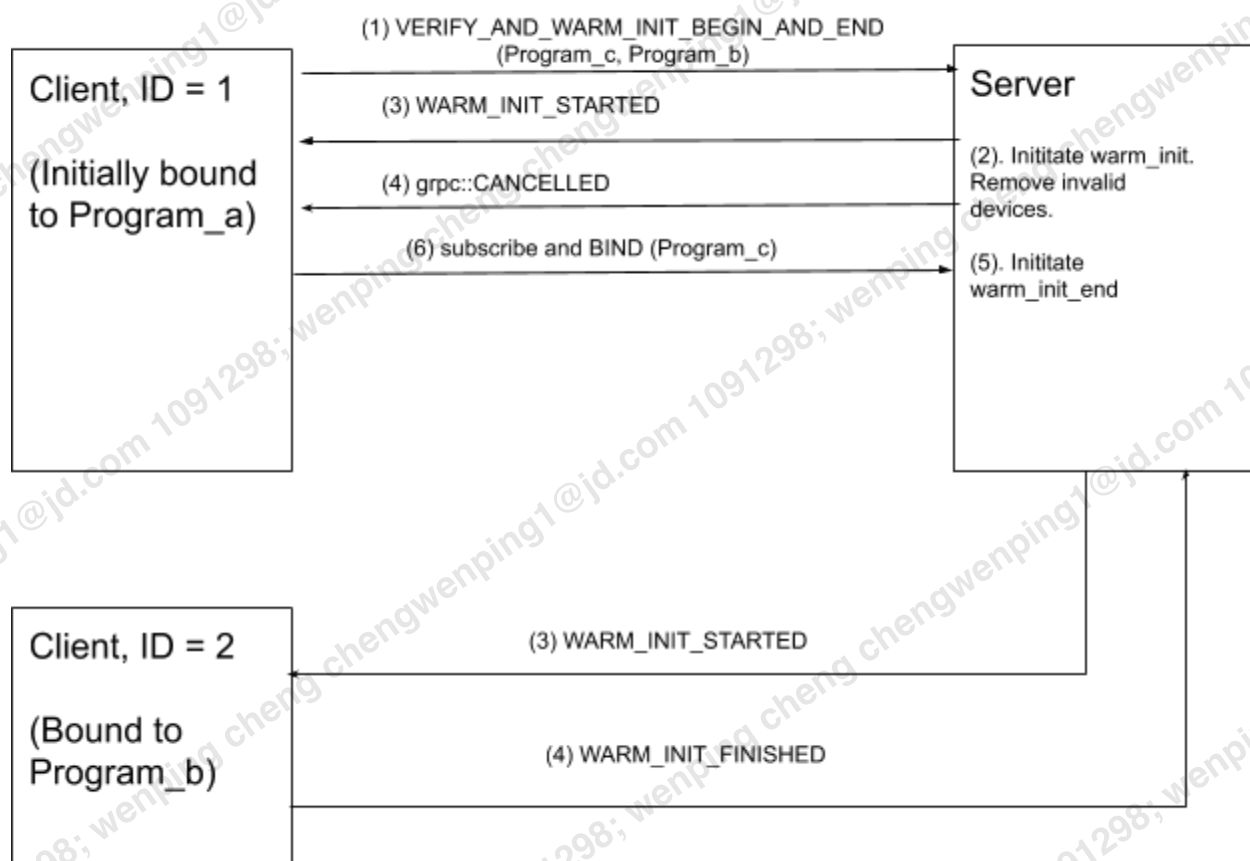


In the example case described above, both the clients are not attached to any program in the beginning. This is a simple case just like Case 4 but where both clients want to replay. Hence in a case like this clients need to communicate with each other since the master client needs to know when others are done replaying.

Case 6: Client 1 and 2 are connected on Program_a and Program_b respectively. Client 1 wants to move to a single program Program_c on the entire device



Case 7: Client 1 and Client 2 are already connected on Program_a and Program_b respectively. Client 1 wants to change the config to Program_c and Program_b and so that client 2 can remain on Program_b.



GetForwardingPipelineConfig

This RPC retrieves the forwarding configuration on a certain device. The rpc is defined as follows

```
rpc GetForwardingPipelineConfig (GetForwardingPipelineConfigRequest)
    returns (GetForwardingPipelineConfigResponse) {}
```

GetForwardingPipelineConfigRequest is defined as follows

```
message GetForwardingPipelineConfigRequest {
    uint32 device_id = 1;
    uint32 client_id = 2;
```

```
}
```

GetForwardingPipelineConfigResponse is defined as follows

```
message GetForwardingPipelineConfigResponse {  
  repeated ForwardingPipelineConfig config = 1;  
}
```

It returns a repeated ForwardingPipelineConfig, which is defined as follows

```
message ForwardingPipelineConfig {  
  string p4_name = 1; // P4 program name  
  bytes bfruntime_info = 2; // BF-RT info json file contents  
  message Profile { // P4 Pipeline Profile  
    string profile_name = 1; // profile name  
    bytes context = 2; // context json file contents  
    bytes binary = 3; // Binary to execute  
    repeated uint32 pipe_scope = 4; // Array of pipe_scope.  
  }  
  repeated Profile profiles = 3;  
}
```

Bfruntime_info represents the bfrt_info.json which is used by the device. A single device can have multiple Profiles. Each profile is represented using the Profile message, which contains the profile_name, the context.json, and the binary that the device is running and the list pipes that the program is running.

Non-P4 tables

Please note that GetForwardingPipeline config returns a concatenated bfruntime_info which contains tables of all non-P4 Tables like \$PORTS and \$PKTGEN as well. These tables are not to be sent as part of bfruntime_info as part of SetForwardingPipelineConfig. If no P4 is present on a device, then a dummy P4 program called \$\$SHARED is sent which contains all the non-p4 tables.

Steps to connect to the GRPC server

Here are the steps to be followed by a client to connect to the BFRuntime GRPC server

1. Open an *insecure Channel* to port 50052 on the host where grpc server is running
2. Send a “*subscribe*” message as part of the RPC ***StreamChannel***(*stream StreamMessageRequest*). This bidirectional stream is kept as a heartbeat channel as well. This also serves as the bidirectional stream for getting back learn and IdleTimeout notifications. Send a “*SetForwardingPipelineConfigRequest*” message as part of the RPC ***SetForwardingPipelineConfig***(*SetForwardingPipelineConfigRequest*) along with details of client-ID, device ID and P4 name. A `grpc::Status::OK` status is sent upon successful API completion.

Note:

1. One client is allowed to communicate with only one (p4name, device_id) at a time. Also, 1 (p4name,device_id) can be only managed by one client at a time. (strict one to one correspondence)
2. The heartbeat-stream opened is to be kept alive at all times for the server to be able to accept any *Read/Write* requests. Issuing a *write_done* command from the client side is just as good as closing the connection, and the server will clean up the client-info.
3. *Read* operation is only available for Entities *table_entry*, *table_usage* and *object_id* for now.
4. *Write* operation is only supported for Entities *table_entry*, *table_attribute* and *table_operation* for now. For *table_attribute* and *table_operation*; *update_type* doesn't matter.
5. All BF-RT IDs like table IDs and Data-field IDs, can be retrieved using 2 ways
 - a. By issuing a ***GetForwardingPipelineConfig***(*GetForwardingPipelineConfigRequest*) request. The entire table metadata json is passed as text and can be parsed by the client in order to cache all object IDs, size etc. This is the recommended method.
 - b. By issuing a *Read* request of an Objectid entity every time an ID is required. (Support to be discontinued soon)

Error Handling

For RPCs *SetForwardingPipelineConfig*, *GetForwardingPipelineConfig* and *StreamChannel*, the error reported is a single `grpc::Status`. Depending on the language of the client's implementation, it is handled in different ways (like exception in python but status in C++ and Go)

For RPCs *Read*, *Write*, the `error_code` of `grpc::Status` contains `grpc::Status::OK` if, everything went well but `grpc::Status::UNKNOWN` even if one of the write operations in the batch fails.

The error_details of `grpc::Status` contains one serialized `::google::rpc::Status`. In the details field of this `::google::rpc::Status` message, all the individual batch errors are encoded in as message "Error" of the `bfruntime.proto`

When a batch of read or write requests are submitted to the server, each of those requests are attempted to be executed. When an error is encountered

The behavior of our server is as follows. When there are repeated updates in a request, if there is an error on any one update, the server sends the status of all the updates in the request back to the client in ONE big message. For instance, in a batch of 10001 updates, where the first 1000 updates succeed and the last one fails, the server would send the status of the first 1000 updates as OK and the status of the 1001st update with the appropriate status.. This causes the message being received by the client to be large, and it can exceed the maximum size of the message that a client is configured to handle. This maximum size is a configurable parameter.