OVS components

The main components of this distribution are:
1. ovs-vswitchd, a daemon that implements the switch, along with a companion Linux kernel module for flow-based switching.
2. ovsdb-server, a lightweight database server that ovs-vswitchd queries to obtain its configuration.
3. ovs-dpctl, a tool for configuring the switch kernel module.
4. Scripts and specs for building RPMs for Citrix XenServer and Red Hat Enterprise Linux. The XenServer RPMs allow Open vSwitch to be installed on a Citrix XenServer host as a drop-in replacement for its switch, with additional functionality.
5. ovs-vsctl, a utility for querying and updating the configuration of ovs-vswitchd.
6. ovs-appctl, a utility that sends commands to running Open vSwitch daemons.

Open vSwitch also provides some tools:

1. ovs-ofctl, a utility for querying and controlling OpenFlow switches and controllers.
2. ovs-pki, a utility for creating and managing the public-key infrastructure for OpenFlow switches.
3. ovs-testcontroller, a simple OpenFlow controller that may be useful for testing (though not for production).
4. A patch to tcpdump that enables it to parse OpenFlow messages.

Datapath

NIC ---> OVS virtual port ---> OVS flow matching ---> OVS actions

The main files are:
(1) datapath.h and datapath.c implement the core procedures
(2) vport-*.h and vport-*.c implement the vport subsystem
(3) action.c defines the interfaces to operate on skb packets.
(4) flow.h and flow.c implements the core data structures and functions for local flow table maintenance, including the create, update, delete, etc., operations.

To do modification, you may need to do:

1. datapath.c: **ovs_dp_process_received_packet**(struct vport *p, struct sk_buff *skb), add piece of your custom code, since this is an entry point for each incoming packet.
2. Design your own flow tables
3. Related to step 2, i.e., design your own actions for the flows.

Interfaces for adding, deleting, modifying flows are defined in datapath.h and datapath.c files.
They use generic netlink APIs to communicate with the userspace, the related APIs are documented here:
http://www.linuxfoundation.org/collaborate/workgroups/networking/generic_netlink_howto#The_genl_ops_Structure

ovs_flow_cmd_new() [datapath.c: add new flow to the flow table.] ---> ovs_nla_get_match() [flow_netlink.c: parses Netlink attributes into a flow key and mask]; ovs_flow_mask_key(); ovs_nla_get_identifier() [flow_netlink.c: Extract flow identifier]; ovs_nla_copy_actions()[flow_netlink.c: Validate actions]; ovs_flow_tbl_insert() [flow_table.c: Put flow in bucket]; ovs_flow_cmd_fill_info(); or update actions...

Another entry point for flow management: **ovs_key_from_nlattrs() [flow_netlink.c]: parse the flows from vswitch, and update or create the keys.**

The packet processing procedure

1. Receive and send packets (vport.h/.c)
(1.1) ***ovs_vport_receive()*** - pass up received packet to the datapath for processing
(1.2) ***ovs_vport_send()*** - Send the packets out the OVS

2. Core packet processing

ovs_vport_receive() ---> ovs_flow_key_extract(); ovs_dp_process_packet() ---> flow = ovs_flow_tbl_lookup_stats(); ovs_execute_actions() (Execute a list of actions against 'skb') ---> do_execute_actions()

If there is no match, it will forward the packet to the ovsd via upcall…

(2.1) Extract the key information from skb packets

ovs_flow_key_extract() [datapath/flow.c; Extract flow from 'skb' into 'key'] ---> **key_extract**() [datapath/flow.c; extracts a flow key from an Ethernet frame; need to parse **xip headers** here]

(2.2) Flow table matching according to the key and skb packets

ovs_dp_process_packet() ---> flow = ***ovs_flow_tbl_lookup_stats()*** [datapath.c: if the flow is NULL, then it calls **ovs_dp_upcall**()]

struct **dp_upcall -** metadata to include with a packet to send to userspace

```
struct dp_upcall_info {
›   struct ip_tunnel_info *egress_tun_info;
›   const void *egress_tun_opts;
›   const struct nlattr *userdata;
›   const struct nlattr *actions;
›   int actions_len;
›   u32 portid;
›   u8 cmd;
›   u16 mru;
};
```

ovs_flow_tbl_lookup_stats() ---> **flow_lookup**() [flow_table.c; Flow lookup does full lookup on flow table. It starts with mask from index passed in *index.] ---> masked_flow_lookup() ---> **ovs_flow_mask_key**(); flow_hash()---> jhash2(); find_bucket() ---> jhash_1word(); flow_cmp_masked_key(flow, &masked_key, &mask->range)

(2.2.1) flow_lookup() - flow_table.c: Flow lookup does full lookup on flow table. It starts with mask from index passed in *index

```
static struct sw_flow *flow_lookup(struct flow_table *tbl,
›   ›   ›   ›   struct table_instance *ti,
›   ›   ›   ›   const struct mask_array *ma,
›   ›   ›   ›   const struct sw_flow_key *key,
›   ›   ›   ›   u32 *n_mask_hit,
›   ›   ›   ›   u32 *index)
```

(2.2.2) masked_flow_lookup() - flow_table.c: masked lookup the flow table

```
static struct sw_flow *masked_flow_lookup(struct table_instance *ti,
					const struct sw_flow_key *unmasked,
					const struct sw_flow_mask *mask,
					u32 *n_mask_hit)
```

(2.2.3) **ovs_flow_mask_key**(&masked_key, unmasked, false, mask) - flow_table.c: Basically, the operation is: masked_key = **unmasked** flow key & mask->key; the masked length is (**mask->end - mask->start**).

(2.2.4) hash = flow_hash(&masked_key, &mask->range); - obtain the hash value by using jhash2()

(2.2.5) head = find_bucket(ti, hash); - find the list head according to the hash value obtained above. Indeed, each list node contains **struct sw_flow**.

(2.2.6) Iterate each node in the flow list, if it matches, return the flow

```
	hlist_for_each_entry_rcu(flow, head, flow_table.node[ti->node_ver]) {
		if (flow->mask == mask && flow->flow_table.hash == hash &&
		    flow_cmp_masked_key(flow, &masked_key, &mask->range))
			return flow;
	}
```

list_for_each_entry_rcu — iterate over rcu list of given type

Synopsis

list_for_each_entry_rcu (pos, head, member);
pos: the type * to use as a loop cursor.
head: the head for your list.
member: the name of the list_struct within the struct.

(2.3) Execute actions according to the matched table; if no match, send packets to userspace.

Core data structures & functions

To deal with the ***nlmsg*** messages in the datapath, we may need to add or modify the flows.

(1) In datapath.h

struct datapath - datapath for flow-based packet switching

```
struct datapath {
	struct rcu_head rcu;
	struct list_head list_node;

	/* Flow table. */
	struct flow_table table;

	/* Switch ports. */
	struct hlist_head *ports;

	/* Stats. */
	struct dp_stats_percpu __percpu *stats_percpu;

	/* Network namespace ref. */
	possible_net_t net;

	u32 user_features;
};
```

In datapath.c

```
static const struct genl_ops dp_flow_genl_ops[] = {
	{ .cmd = OVS_FLOW_CMD_NEW,
	  .flags = GENL_ADMIN_PERM, /* Requires CAP_NET_ADMIN privilege. */
	  .policy = flow_policy,
	  .doit = ovs_flow_cmd_new
	},
	{ .cmd = OVS_FLOW_CMD_DEL,
	  .flags = GENL_ADMIN_PERM, /* Requires CAP_NET_ADMIN privilege. */
	  .policy = flow_policy,
	  .doit = ovs_flow_cmd_del
	},
	{ .cmd = OVS_FLOW_CMD_GET,
	  .flags = 0,		/* OK for unprivileged users. */
	  .policy = flow_policy,
	  .doit = ovs_flow_cmd_get,
```

```
		.dumpit = ovs_flow_cmd_dump
	},
	{ .cmd = OVS_FLOW_CMD_SET,
	  .flags = GENL_ADMIN_PERM, /* Requires CAP_NET_ADMIN privilege. */
	  .policy = flow_policy,
	  .doit = ovs_flow_cmd_set,
	},
};
```

(1) ovs_flow_from_nlattrs() [flow_netlink.c]: parse the flows from vswitch, and update or create the keys.
(2) validate_actions(): to decide the validity of the actions
(3) ovs_flow_tbl_lookup(): lookup the table to see whether there is match_fields. If so, modify the original flow actions, otherwise, create new flows.
(4) ovs_flow_actions_alloc(): allocate new actions
(5) ovs_flow_tbl_insert(): add new flows
(6) if it only needs to modify the flow actions, after updating the old_action, it needs to release the old_action.
(7) clear_stats(): clear all the statistical information.

(2) In vport.h
struct vport_ops - definition of a type of virtual port

In vport.c

(2.1) ***ovs_vport_receive()*** - pass up received packet to the datapath for processing
int **ovs_vport_receive**(struct vport *vport, struct sk_buff *skb, const struct ip_tunnel_info *tun_info);

netdev_frame_hook() [vport-netdev.c] ---> **ovs_vport_receive**() ---> ovs_flow_key_extract(); ovs_dp_process_packet() ---> flow = ovs_flow_tbl_lookup_stats() ---> ovs_execute_actions() (Execute a list of actions against 'skb') ---> do_execute_actions()

If there is no match, it will forward the packet to the ovsd via upcall…

(2.2) Send the packets out the OVS

```
void ovs_vport_send(struct vport *vport, struct sk_buff *skb)
{
	int mtu = vport->dev->mtu;

	if (unlikely(packet_length(skb) > mtu && !skb_is_gso(skb))) {
		net_warn_ratelimited("%s: dropped over-mtu packet: %d > %d\n",
				     vport->dev->name,
```

```
›   ›   ›   ›       packet_length(skb), mtu);
›   ›   vport->dev->stats.tx_errors++;
›   ›   goto drop;
›   }

›   skb->dev = vport->dev;
›   vport->ops->send(skb);
›   return;

drop:
›   kfree_skb(skb);
}
```

(3) In flow.h

This data structure is quite important, it defines the *KEY* in a flow, which is mainly extracted from the packet header, and can be used to do the flow matching.

```
struct sw_flow_key {
        u8 tun_opts[255];
        u8 tun_opts_len;
        struct ip_tunnel_key tun_key;  /* Encapsulating tunnel key. */
        struct {
                u32     priority;       /* Packet QoS priority. */
                u32     skb_mark;       /* SKB mark. */
                u16     in_port;/* Input switch port (or DP_MAX_PORTS). */
        } __packed phy; /* Safe when right after 'tun_key'. */
        u32 ovs_flow_hash;              /* Datapath computed hash value.  */
        u32 recirc_id;                  /* Recirculation ID.  */
        struct {
                u8   src[ETH_ALEN]; /* Ethernet source address. */
                u8   dst[ETH_ALEN]; /* Ethernet destination address. */
                __be16 tci;             /* 0 if no VLAN, VLAN_TAG_PRESENT set otherwise. */
                __be16 type;            /* Ethernet frame type. */
        } eth;
        union {
                struct {
                        __be32 top_lse; /* top label stack entry */
                } mpls;
                struct {
                        u8   proto;     /* IP protocol or lower 8 bits of ARP opcode. */
                        u8   tos;          /* IP ToS. */
                        u8   ttl;          /* IP TTL/hop limit. */
                        u8   frag;      /* One of OVS_FRAG_TYPE_*. */
```

```
				} ip;
			};
			struct {
				__be16 src;		/* TCP/UDP/SCTP source port. */
				__be16 dst;		/* TCP/UDP/SCTP destination port. */
				__be16 flags;		/* TCP flags. */
			} tp;
			union {
				struct {
					struct {
						__be32 src;	/* IP source address. */
						__be32 dst;	/* IP destination address. */
					} addr;
					struct {
						u8 sha[ETH_ALEN];	/* ARP source hardware address. */
						u8 tha[ETH_ALEN];	/* ARP target hardware address. */
					} arp;
				} ipv4;
				struct {
					struct {
						struct in6_addr src;	/* IPv6 source address. */
						struct in6_addr dst;	/* IPv6 destination address. */
					} addr;
					__be32 label;			/* IPv6 flow label. */
					struct {
						struct in6_addr target;	/* ND target address. */
						u8 sll[ETH_ALEN];	/* ND source link layer address. */
						u8 tll[ETH_ALEN];	/* ND target link layer address. */
					} nd;
				} ipv6;
			};
			struct {
				/* Connection tracking fields. */
				u16 zone;
				u32 mark;
				u8 state;
				struct ovs_key_ct_labels labels;
			} ct;

} __aligned(BITS_PER_LONG/8); /* Ensure that we can do comparisons as longs. */
```

```
struct sw_flow {
›	struct rcu_head rcu;
```

```
	struct {
		struct hlist_node node[2];
		u32 hash;
	} flow_table, ufid_table;
	int stats_last_writer;		/* NUMA-node id of the last writer on
					 * 'stats[0]'.
					 */
	struct sw_flow_key key;
	struct sw_flow_id id;
	struct sw_flow_mask *mask;
	struct sw_flow_actions __rcu *sf_acts;
	struct flow_stats __rcu *stats[]; /* One for each NUMA node.  First one
					   * is allocated at flow creation time,
					   * the rest are allocated on demand
					   * while holding the 'stats[0].lock'.
					   */
};
```

```
struct sw_flow_key_range {
	unsigned short int start;
	unsigned short int end;
};

struct sw_flow_mask {
	int ref_count;
	struct rcu_head rcu;
	struct sw_flow_key_range range;
	struct sw_flow_key key;
};

struct sw_flow_match {
	struct sw_flow_key *key;
	struct sw_flow_key_range range;
	struct sw_flow_mask *mask;
};
```

(4) In flow_table.h

```
struct flex_array_part {
	char elements[FLEX_ARRAY_PART_SIZE]; // PAGE_SIZE
};
```

```
struct flex_array {
	union {
		struct {
			int element_size;
			int total_nr_elements;
			int elems_per_part;
			struct reciprocal_value reciprocal_elems;
			struct flex_array_part *parts[];
		};
		/*
		 * This little trick makes sure that
		 * sizeof(flex_array) == PAGE_SIZE
		 */
		char padding[FLEX_ARRAY_BASE_SIZE];
	};
};
```

```
struct table_instance {
	struct flex_array *buckets;
	unsigned int n_buckets;
	struct rcu_head rcu;
	int node_ver;
	u32 hash_seed;
	bool keep_flows;
};

struct flow_table {
	struct table_instance __rcu *ti;
	struct table_instance __rcu *ufid_ti;
	struct mask_cache_entry __percpu *mask_cache;
	struct mask_array __rcu *mask_array;
	unsigned long last_rehash;
	unsigned int count;
	unsigned int ufid_count;
};
```

(5) In flow_netlink.c

```
/**
 * ovs_nla_get_match - parses Netlink attributes into a flow key and
 * mask. In case the 'mask' is NULL, the flow is treated as exact match
 * flow. Otherwise, it is treated as a wildcarded flow, except the mask
 * does not include any don't care bit.
 * @net: Used to determine per-namespace field support.
```

```
 * @match: receives the extracted flow match information.
 * @key: Netlink attribute holding nested %OVS_KEY_ATTR_* Netlink attribute
 * sequence. The fields should of the packet that triggered the creation
 * of this flow.
 * @mask: Optional. Netlink attribute holding nested %OVS_KEY_ATTR_* Netlink
 * attribute specifies the mask field of the wildcarded flow.
 * @log: Boolean to allow kernel error logging.  Normally true, but when
 * probing for feature compatibility this should be passed in as false to
 * suppress unnecessary error logging.
 */
int ovs_nla_get_match(struct net *net, struct sw_flow_match *match,
›   ›       const struct nlattr *nla_key,
›   ›       const struct nlattr *nla_mask,
›   ›       bool log)
```

```
static int ovs_key_from_nlattrs(struct net *net, struct sw_flow_match *match,
›   ›   ›   ›   u64 attrs, const struct nlattr **a,
›   ›   ›   ›   bool is_mask, bool log)
```

```
/**
 * ovs_nla_get_flow_metadata - parses Netlink attributes into a flow key.
 * @key: Receives extracted in_port, priority, tun_key and skb_mark.
 * @attr: Netlink attribute holding nested %OVS_KEY_ATTR_* Netlink attribute
 * sequence.
 * @log: Boolean to allow kernel error logging.  Normally true, but when
 * probing for feature compatibility this should be passed in as false to
 * suppress unnecessary error logging.
 *
 * This parses a series of Netlink attributes that form a flow key, which must
 * take the same form accepted by flow_from_nlattrs(), but only enough of it to
 * get the metadata, that is, the parts of the flow key that cannot be
 * extracted from the packet itself.
 */

int ovs_nla_get_flow_metadata(struct net *net, const struct nlattr *attr,
›   ›   ›       struct sw_flow_key *key,
›   ›   ›       bool log)
```

```
/* 'key' must be the masked key. */
```

```
int ovs_nla_copy_actions(struct net *net, const struct nlattr *attr,
			const struct sw_flow_key *key,
			struct sw_flow_actions **sfa, bool log)
{
	int err;

	*sfa = nla_alloc_flow_actions(nla_len(attr), log);
	if (IS_ERR(*sfa))
		return PTR_ERR(*sfa);

	(*sfa)->orig_len = nla_len(attr);
	err = __ovs_nla_copy_actions(net, attr, key, 0, sfa, key->eth.type,
				     key->eth.tci, log);
	if (err)
		ovs_nla_free_flow_actions(*sfa);

	return err;
}
```

vswitchd

This part is an important component for managing the datapath, and it implements the OpenFlow logic, switch management and forward, etc. The main file is **vswitchd/ovs-vswitchd.c**.

(1) bridge.h and bridge.c

(1.1) functions:

```
void bridge_init(const char *remote);
void bridge_exit(void);

void bridge_run(void);
void bridge_wait(void);

void bridge_get_memory_usage(struct simap *usage);
```

bridge_run() ---> bridge_init_ofproto() ---> ofproto_init() [initialize the ofproto library] ---> ofproto_class_register(&ofproto_dpif_class) [ofproto.c; ofproto-dpif.c]

bridge_run() ---> bridge_run__() [vswitchd/bridge.c] ---> ofproto_run() ---> run_rule_executes()

```
static void
bridge_run__(void)
{
    struct bridge *br;
    struct sset types;
    const char *type;

    /* Let each datapath type do the work that it needs to do. */
    sset_init(&types);
    ofproto_enumerate_types(&types);
    SSET_FOR_EACH (type, &types) {
        ofproto_type_run(type);
    }
    sset_destroy(&types);

    /* Let each bridge do the work that it needs to do. */
    HMAP_FOR_EACH (br, node, &all_bridges) {
        ofproto_run(br->ofproto);
    }
}
```

(1.2) data structures:

```
struct bridge {
    struct hmap_node node;      /* In 'all_bridges'. */
    char *name;                 /* User-specified arbitrary name. */
    char *type;                 /* Datapath type. */
    struct eth_addr ea;         /* Bridge Ethernet Address. */
    struct eth_addr default_ea; /* Default MAC. */
    const struct ovsrec_bridge *cfg;

    /* OpenFlow switch processing. */
    struct ofproto *ofproto;    /* OpenFlow switch. */

    /* Bridge ports. */
    struct hmap ports;          /* "struct port"s indexed by name. */
    struct hmap ifaces;         /* "struct iface"s indexed by ofp_port. */
    struct hmap iface_by_name;  /* "struct iface"s indexed by name. */

    /* Port mirroring. */
    struct hmap mirrors;        /* "struct mirror" indexed by UUID. */

    /* Auto Attach */
    struct hmap mappings;       /* "struct" indexed by UUID */

    /* Used during reconfiguration. */
    struct shash wanted_ports;

    /* Synthetic local port if necessary. */
    struct ovsrec_port synth_local_port;
    struct ovsrec_interface synth_local_iface;
    struct ovsrec_interface *synth_local_ifacep;
};
```

In particular, the ***ofproto*** pointer is very important, which points to an OpenFlow switch, and is responsible for all the processes of the OpenFlow switch. Indeed, the ***vswitchd*** component will detect and call the ofproto on all bridges, and run the corresponding **processing functions**.

struct ofproto contains a member ***struct ofproto_class***, which is defined by each ofproto implementation. These functions (implemented in ofproto-dpif.c) work primarily with four different kinds of data structures:

```
 *   - "struct ofproto", which represents an OpenFlow switch.
 *
 *   - "struct ofport", which represents a port within an ofproto.
```

```
 *
 *   - "struct rule", which represents an OpenFlow flow within an ofproto.
 *
 *   - "struct ofgroup", which represents an OpenFlow 1.1+ group within an
 *     ofproto.
```

ofproto_class_register(&ofproto_dpif_class); [ofproto/ofproto.c] ---> const struct ofproto_class ofproto_dpif_class [ofproto/ofproto-dpif.c]

include/linux/openvswitch.h

enum ovs_datapath_cmd
enum ovs_datapath_attr - attributes for %OVS_DP_* commands.
enum ovs_packet_attr - attributes for %OVS_PACKET_* commands.
enum ovs_key_attr - Key types
enum ovs_flow_attr - attributes for %OVS_FLOW_* commands
enum ovs_action_attr - Action types.

```
enum ovs_datapath_cmd {
›   OVS_DP_CMD_UNSPEC,
›   OVS_DP_CMD_NEW,
›   OVS_DP_CMD_DEL,
›   OVS_DP_CMD_GET,
›   OVS_DP_CMD_SET
};
```

```
enum ovs_key_attr {
…...
›   OVS_KEY_ATTR_ETHERTYPE,›/* be16 Ethernet type */
›   OVS_KEY_ATTR_IPV4,      /* struct ovs_key_ipv4 */
›   OVS_KEY_ATTR_IPV6,      /* struct ovs_key_ipv6 */
…...
›   __OVS_KEY_ATTR_MAX
};
```

Lib

lib/flow.h; lib/flow.c; lib/meta-flow.h; lib/meta-flow.c

1. The flow data structure in the network.

```
struct flow {
    /* Metadata */
    struct flow_tnl tunnel;     /* Encapsulating tunnel parameters. */
    ovs_be64 metadata;          /* OpenFlow Metadata. */
    uint32_t regs[FLOW_N_REGS]; /* Registers. */
    uint32_t skb_priority;      /* Packet priority for QoS. */
    uint32_t pkt_mark;          /* Packet mark. */
    uint32_t dp_hash;           /* Datapath computed hash value. The exact
                                 * computation is opaque to the user space. */
    union flow_in_port in_port; /* Input port.*/
    uint32_t recirc_id;         /* Must be exact match. */
    uint16_t ct_state;          /* Connection tracking state. */
    uint16_t ct_zone;           /* Connection tracking zone. */
    uint32_t ct_mark;           /* Connection mark.*/
    uint8_t pad1[4];            /* Pad to 64 bits. */
    ovs_u128 ct_label;          /* Connection label. */
    uint32_t conj_id;           /* Conjunction ID. */
    ofp_port_t actset_output;   /* Output port in action set. */
    uint8_t pad2[2];            /* Pad to 64 bits. */

    /* L2, Order the same as in the Ethernet header! (64-bit aligned) */
    struct eth_addr dl_dst;     /* Ethernet destination address. */
    struct eth_addr dl_src;     /* Ethernet source address. */
    ovs_be16 dl_type;           /* Ethernet frame type. */
    ovs_be16 vlan_tci;          /* If 802.1Q, TCI | VLAN_CFI; otherwise 0. */
    ovs_be32 mpls_lse[ROUND_UP(FLOW_MAX_MPLS_LABELS, 2)]; /* MPLS label stack
                                                            (with padding). */
    /* L3 (64-bit aligned) */
    ovs_be32 nw_src;            /* IPv4 source address. */
    ovs_be32 nw_dst;            /* IPv4 destination address. */
    struct in6_addr ipv6_src;   /* IPv6 source address. */
    struct in6_addr ipv6_dst;   /* IPv6 destination address. */
    ovs_be32 ipv6_label;        /* IPv6 flow label. */
    uint8_t nw_frag;            /* FLOW_FRAG_* flags. */
    uint8_t nw_tos;             /* IP ToS (including DSCP and ECN). */
    uint8_t nw_ttl;             /* IP TTL/Hop Limit. */
```

```
    uint8_t nw_proto;           /* IP protocol or low 8 bits of ARP opcode. */
    struct in6_addr nd_target;  /* IPv6 neighbor discovery (ND) target. */
    struct eth_addr arp_sha;    /* ARP/ND source hardware address. */
    struct eth_addr arp_tha;    /* ARP/ND target hardware address. */
    ovs_be16 tcp_flags;         /* TCP flags. With L3 to avoid matching L4. */
    ovs_be16 pad3;              /* Pad to 64 bits. */

    /* L4 (64-bit aligned) */
    ovs_be16 tp_src;            /* TCP/UDP/SCTP source port/ICMP type. */
    ovs_be16 tp_dst;            /* TCP/UDP/SCTP destination port/ICMP code. */
    ovs_be32 igmp_group_ip4;    /* IGMP group IPv4 address.
                                 * Keep last for BUILD_ASSERT_DECL below. */
};
```

2. A sparse representation of a "struct flow".

2.1 struct **miniflow** {
    struct flowmap map;
    /* Followed by:
     *   uint64_t values[n];
     * where 'n' is miniflow_n_values(miniflow). */
};

 * The map member ***hold one bit for each uint64_t in a "struct flow"***. Each
 * 0-bit indicates that the corresponding uint64_t is zero, each 1-bit that it
 * *may* be nonzero (see below how this applies to minimasks).

2.2 #define FLOWMAP_UNITS **DIV_ROUND_UP**(FLOW_U64S, MAP_T_BITS)

2.3 struct **flowmap** {
    map_t bits[FLOWMAP_UNITS];
};

3. Buffer for holding packet data

```
struct dp_packet {
#ifdef DPDK_NETDEV
    struct rte_mbuf mbuf;       /* DPDK mbuf */
#else
    void *base_;                /* First byte of allocated space. */
    uint16_t allocated_;        /* Number of bytes allocated. */
    uint16_t data_ofs;          /* First byte actually in use. */
    uint32_t size_;             /* Number of bytes in use. */
```

```
    uint32_t rss_hash;          /* Packet hash. */
    bool rss_hash_valid;        /* Is the 'rss_hash' valid? */
#endif
    enum dp_packet_source source;  /* Source of memory allocated as 'base'. */
    uint8_t l2_pad_size;           /* Detected l2 padding size.
                                * Padding is non-pullable. */
    uint16_t l2_5_ofs;            /* MPLS label stack offset, or UINT16_MAX */
    uint16_t l3_ofs;              /* Network-level header offset,
                                * or UINT16_MAX. */
    uint16_t l4_ofs;              /* Transport-level header offset,
                                 or UINT16_MAX. */
    struct pkt_metadata md;
};
```

4. Context for pushing data to a miniflow.

```
struct mf_ctx {
    struct flowmap map;
    uint64_t *data;
    uint64_t * const end;
};
```

5. Useful Functions

5.1 offsetof(), container_of(): https://en.wikipedia.org/wiki/Offsetof

offsetof(struct flow, FIELD);

5.2 **miniflow_push_***(MF, FIELD, VALUE) macros allow filling in a miniflow data values in order.

e.g.,

```
#define miniflow_push_uint64_(MF, OFS, VALUE)             \
{                                                         \
    MINIFLOW_ASSERT(MF.data < MF.end && (OFS) % 8 == 0);  \
    *MF.data++ = VALUE;                                   \
    miniflow_set_map(MF, OFS / 8);                        \
}
```

5.3 **miniflow_set_map**(MF, OFS)

```
#define miniflow_set_map(MF, OFS)           \
```

```
{                                              \
    ASSERT_FLOWMAP_NOT_SET(&MF.map, (OFS));  \
    flowmap_set(&MF.map, (OFS), 1);          \
}
```

5.4 static inline void **flowmap_set**(struct flowmap *, size_t idx,
unsigned int n_bits);

```
/* Set the 'n_bits' consecutive bits in 'fm', starting at bit 'idx'.
 * 'n_bits' can be at most MAP_T_BITS. */
static inline void
flowmap_set(struct flowmap *fm, size_t idx, unsigned int n_bits)
{
    map_t n_bits_mask = (MAP_1 << n_bits) - 1;
    size_t unit = idx / MAP_T_BITS;

    idx %= MAP_T_BITS;

    fm->bits[unit] |= n_bits_mask << idx;
    /* The seemingly unnecessary bounds check on 'unit' is a workaround for a
     * false-positive array out of bounds error by GCC 4.9. */
    if (unit + 1 < FLOWMAP_UNITS && idx + n_bits > MAP_T_BITS) {
        /* 'MAP_T_BITS - idx' bits were set on 'unit', set the remaining
         * bits from the next unit. */
        fm->bits[unit + 1] |= n_bits_mask >> (MAP_T_BITS - idx);
    }
}
```

5.5 **flow_extract()**

1. dp_register_provider(const struct dpif_class *new_class) [lib/dpif.c: registers a new datapath provider.]
const struct dpif_class dpif_netdev_class; [dpif-provider.h, dpif-netdev.c; Datapath interface class structure, to be defined by each implementation of a datapath interface.]

dp_initialize() [lib/dpif.c] ---> dp_register_provider() ---> static const struct dpif_class *base_dpif_classes[] ---> const struct dpif_class ***dpif_netdev_class*** ---> dpif_netdev_operate() ---> dpif_netdev_execute() ---> dp_netdev_execute_actions() ---> dp_execute_cb() [lib/dpif-netdev.c]

lib/dpif.c

```
static const struct dpif_class *base_dpif_classes[] = {
#if defined(__linux__) || defined(_WIN32)
    &dpif_netlink_class,
#endif
```

```
    &dpif_netdev_class,
};
```

system, netdev

2. const struct ofproto_class ***ofproto_dpif_class*** [ofproto-dpif.c] ---> type_run() [Performs any periodic activity required on ofprotos of type 'type'.] ---> udpif_set_threads() ---> udpif_start_threads() ---> udpif_upcall_handler() ---> recv_upcalls() ---> flow_extract(); process_upcall(); handle_upcalls()

3. run_rule_executes() [ofproto/ofproto.c]

lib/odp-util.c

We need to make sure that the KEYs in userspace and kernel path are consistent. Otherwise, it may lead to weird errors.

Debugging

http://openvswitch.org/slides/OVS-Debugging-110414.pdf

In openvswitch/vport.c
int ovs_vport_receive() needs to memset the **key**, otherwise, there are many random flows.

XIA Implementation

```
#define XIA_ENTRY_NODE_INDEX›   0x7e
```

```
static inline struct xia_row *xip_last_row(struct xia_row *addr,
›   int num_dst, int last_node)
{
›   return last_node == XIA_ENTRY_NODE_INDEX
›   ›   ? &addr[num_dst - 1]
›   ›   : &addr[last_node];
}
```

```
struct xip_dst {
›   struct dst_entry›   dst;

›   char›   ›   ›   after_dst[0];

›   /* Since the lookup key is big, keeping its hash is handy
›    * to minimize comparision time.
›    */
›   u32››   ›   key_hash;

›   /* Lookup key. */
›   struct xia_xid› ›   xids[XIA_OUTDEGREE_MAX];
›   /* If true, the traffic comes from a device. */
›   u8› ›   ›   input;

›   /* Action that is taken when the chosen edge is
›    * not a sink (passthrough), and when it is a sink.
›    * See enum XDST_ACTION for possible values.
›    */
›   u8› ›   ›   passthrough_action;
›   u8› ›   ›   sink_action;

›   /* -1› ›   ›   ›   None
›    *  0› ›   ›   ›   First
›    *› ›   ...››   ...
›    * (XIA_OUTDEGREE_MAX - 1)› Last edge
›    */
›   s8› ›   ›   chosen_edge;
```

```
› /* Extra information for dst.input and dst.output methods.
›  * This field should only be used by the principal that sets
›  * the positive anchor.
›  */
› void› › › *info;
}
```

The potential issues with the **struct flowmap**, since the **sizeof(struct flow)** is 560 bytes. However, the current implementation of the **struct flowmap** can support 64 * 8bytes = 512 bytes. That's the reason why our new added XIA fields, like xia_version, xia_last_node, cannot be recognized correctly at first.

Hash value + last_node + 4 lookup keys.

Do we need to add "chosen_edge"? I think so. But the chosen_edge cannot be extracted from the packets? How to deal with it in the struct flow?

There are two ways to perform the routing:
(1) actions
(2) after the flow is matched

How to encode the networking service quality in XIDs? By encoding it into the sink node? Or each node? If it's in the sink node, will every lookup also depends on the sink node?

Just encode it in the HIDs. If we have enough time, we can build a new principle to choose the network links

On simple solution is that, we can add a "**quality_of_net_service**" field in the **struct ovs_key_xia**, this field can be extracted from the sink node.

Then, how is the topology to test it? Will different networking requirements lead to different DAG addresses? Say, if two end hosts want to communicate with an app by choosing different links, how will we deal with it?

Dynamic node

http://www.kiranvemuri.info/computer-networks/sdn/mininet-advanced-users-dynamic-topology-changes/

Useful material for OVS datapath development
https://www.kernel.org/doc/Documentation/networking/openvswitch.txt

Match + Actions

1. struct nla_policy

2.

ovs/lib/netlink-protocol.h

```
struct nlattr {
    uint16_t nla_len;
    uint16_t nla_type;
};
BUILD_ASSERT_DECL(sizeof(struct nlattr) == 4);
```

ovs/datapath/flow.h

```
struct sw_flow_actions {
›   struct rcu_head rcu;
›   size_t orig_len;›   /* From flow_cmd_new netlink actions size */
›   u32 actions_len;
›   struct nlattr actions[];
};
```

```
/**
 * ovs_nla_get_match - parses Netlink attributes into a flow key and
 * mask. In case the 'mask' is NULL, the flow is treated as exact match
 * flow. Otherwise, it is treated as a wildcarded flow, except the mask
 * does not include any don't care bit.
 * @net: Used to determine per-namespace field support.
 * @match: receives the extracted flow match information.
 * @key: Netlink attribute holding nested %OVS_KEY_ATTR_* Netlink attribute
 * sequence. The fields should of the packet that triggered the creation
 * of this flow.
 * @mask: Optional. Netlink attribute holding nested %OVS_KEY_ATTR_* Netlink
 * attribute specifies the mask field of the wildcarded flow.
 * @log: Boolean to allow kernel error logging.  Normally true, but when
 * probing for feature compatibility this should be passed in as false to
 * suppress unnecessary error logging.
 */
int ovs_nla_get_match(struct net *net, struct sw_flow_match *match,
›   ›       const struct nlattr *nla_key,
›   ›       const struct nlattr *nla_mask,
```

20 › › bool log)

Netlink related: http://lxr.free-electrons.com/source/include/uapi/linux/netlink.h#L186

datapath.c

ovs_flow_cmd_new() ---> ovs_nla_get_match(); ovs_nla_copy_actions(); ovs_flow_tbl_insert() [put flow in bucket];

ovs_flow_cmd_new() [datapath.c: add new flow to the flow table.] ---> ovs_nla_get_match() [flow_netlink.c: parses Netlink attributes into a flow key and mask]; ovs_flow_mask_key(); ovs_nla_get_identifier() [flow_netlink.c: Extract flow identifier]; ***ovs_nla_copy_actions***()[flow_netlink.c: Validate actions]; ovs_flow_tbl_insert() [flow_table.c: Put flow in bucket]; ovs_flow_cmd_fill_info(); or update actions...

ovs_vport_receive() ---> ovs_flow_key_extract(); ovs_dp_process_packet() ---> flow = ovs_flow_tbl_lookup_stats(); ovs_execute_actions() (Execute a list of actions against 'skb') ---> ***do_execute_actions***()

lib/ofp-actions.h to add new actions.

Define struct ofpact_xxx

To define new types, like struct xia_xid, we need to modify the include/openvswitch/types.h file.
E.g., struct eth_addr.

Issues:
1. sudo ovs-ofctl add-flow s1 ipv4,actions=mod_nw_ttl:5
sudo ovs-ofctl dump-flows s1

xxx. ip actions=***drop***

XIDs

lib/meta-flow.h

1. How to define a new Type, like MAC, IPv6, etc., and introduce a new formatting

```
 * Every field must specify the following key-value pairs:
 *
 *   Type:
 *
 *     The format and size of the field's value.  Some possible values are
 *     generic:
 *
 *         u8: A one-byte field.
 *         be16: A two-byte field.
 *         be32: A four-byte field.
 *         be64: An eight-byte field.
 *
 *     The remaining values imply more about the value's semantics, though OVS
 *     does not currently take advantage of this additional information:
 *
 *         MAC: A six-byte field whose value is an Ethernet address.
 *         IPv6: A 16-byte field whose value is an IPv6 address.
 *         tunnelMD: A variable length field, up to 124 bytes, that carries
 *                   tunnel metadata.
```

```
 *   Formatting:
 *
 *     Explains how a field's value is formatted and parsed for human
 *     consumption.  Some of the options are fairly generally useful:
 *
 *       decimal: Formats the value as a decimal number.  On parsing, accepts
 *         decimal (with no prefix), hexadecimal with 0x prefix, or octal
 *         with 0 prefix.
 *
 *       hexadecimal: Same as decimal except nonzero values are formatted in
 *         hex with 0x prefix.  The default for parsing is *not* hexadecimal:
 *         only with a 0x prefix is the input in hexadecimal.
 *
 *       Ethernet: Formats and accepts the common format xx:xx:xx:xx:xx:xx.
 *         6-byte fields only.
 *
 *       IPv4: Formats and accepts the common format w.x.y.z.  4-byte fields
```

```
 *        only.
 *
 *      IPv6: Formats and accepts the common IPv6 formats.  16-byte fields
 *        only.
```

```
/* How to format or parse a field's value. */
enum OVS_PACKED_ENUM mf_string {
    /* Integer formats.
     *
     * The particular MFS_* constant sets the output format.  On input, either
     * decimal or hexadecimal (prefixed with 0x) is accepted. */
    MFS_DECIMAL,
    MFS_HEXADECIMAL,

    /* Other formats. */
    MFS_CT_STATE,           /* Connection tracking state */
    MFS_ETHERNET,
    MFS_IPV4,
    MFS_IPV6,
    MFS_OFP_PORT,           /* 16-bit OpenFlow 1.0 port number or name. */
    MFS_OFP_PORT_OXM,       /* 32-bit OpenFlow 1.1+ port number or name. */
    MFS_FRAG,               /* no, yes, first, later, not_later */
    MFS_TNL_FLAGS,          /* FLOW_TNL_F_* flags */
    MFS_TCP_FLAGS,          /* TCP_* flags */
};
```

2. An important data structure to manage the fields in meta-flow.h

```
struct mf_field {
    /* Identification. */
    enum mf_field_id id;        /* MFF_*. */
    const char *name;           /* Name of this field, e.g. "eth_type". */
    const char *extra_name;     /* Alternate name, e.g. "dl_type", or NULL. */

    /* Size.
     *
     * Most fields have n_bytes * 8 == n_bits.  There are a few exceptions:
     *
     *     - "dl_vlan" is 2 bytes but only 12 bits.
     *     - "dl_vlan_pcp" is 1 byte but only 3 bits.
     *     - "is_frag" is 1 byte but only 2 bits.
     *     - "ipv6_label" is 4 bytes but only 20 bits.
     *     - "mpls_label" is 4 bytes but only 20 bits.
```

```
 *     - "mpls_tc"    is 1 byte but only 3 bits.
 *     - "mpls_bos"   is 1 byte but only 1 bit.
 */
unsigned int n_bytes;       /* Width of the field in bytes. */
unsigned int n_bits;        /* Number of significant bits in field. */
bool variable_len;          /* Length is variable, if so width is max. */
……
}
```

3./* The representation of a field's value. */

```
union mf_value {
    uint8_t tun_metadata[128];
    struct in6_addr ipv6;
    struct eth_addr mac;
    ovs_be128 be128;
    ovs_be64 be64;
    ovs_be32 be32;
    ovs_be16 be16;
    uint8_t u8;
};
```

4. Parsing and formatting

```
/* Parsing and formatting. */
char *mf_parse(const struct mf_field *, const char *,
            union mf_value *value, union mf_value *mask);
char *mf_parse_value(const struct mf_field *, const char *, union mf_value *);
void mf_format(const struct mf_field *,
            const union mf_value *value, const union mf_value *mask,
            struct ds *);
void mf_format_subvalue(const union mf_subvalue *subvalue, struct ds *s);
```

lib/meta-flow.c

```
static void nxm_do_init(void)
```

```
const struct mf_field mf_fields[MFF_N_IDS] = {
#include "meta-flow.inc"
};
```

build-aux/extract-ofp-fields: a Python file defines FORMATTING, etc.