Topic 4: Microservice Architecture: Controller Implementation and Pod/Networking Implementation

Nov. 30, 2021

Y. Richard Yang
Computer Science Department, Yale University
Email: yry@cs.yale.edu

http://zoo.cs.yale.edu/classes/cs434/
Final Project planning

- Dec. 3: Checkpoint (2-page update)
- Dec. 22: Due (report ~6 pages)
Recap: Services and Microservices

Architectures Basic Idea and Background

- Basic idea: divide an application into components/services
- Microservice architecture: Instead of using virtual machines, use lighter-weight isolation mechanism (aka microservices)
  - One main isolation mechanism: container
    - Container is just a higher abstraction built on top of Linux namespace and cgroup for isolation
    - Linux has multiple types of namespaces, e.g., PID, network (NET), IPC, mount namespace (NS), IPC, identify (UTS)
    - Each process belongs to a namespace of each type, and only sees those in the same namespace

```
$ ls -l /proc/88/ns
  total 0
  lrwxrwxrwx 1 dj 0 Jan 4 04:12 ipc -> ipc:[4026531859]
  lrwxrwxrwx 1 dj 0 Jan 4 04:12 net -> net:[4026531866]
  lrwxrwxrwx 1 dj 0 Jan 4 04:12 pid -> pid:[4026531866]
  lrwxrwxrwx 1 dj 0 Jan 4 04:12 user -> user:[4026531837]
```
Recap: Basic Services of a Microservice Architecture (Kubernetes)

- **Pod**: the smallest deployable units of computing that one can create and manage
  - A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled; Pod’s contents are always co-located and co-scheduled, and run in a shared storage and network context
    - `kubectl run pingpong --image=alpine ping 1.1.1.1`

- **Deployment**: a declarative specification about a desired state
  - the Deployment Controller changes the actual state to the desired state at a controlled rate
    - `kubectl create deployment ngservers --image=nginx`
    - `kubectl scale deployment ngservers --replicas=3`

- **Service**: a stable address for a pod (or a bunch of pods)
  - If you want to connect to the pod(s), you need to create a *service*
  - Four types of services: ClusterIP, NodePort, LoadBalancer, ExternalName
    - `kubectl expose deployment ngservers --port=8888`
    - `kubectl describe endpoints ngservers`
    - `kubectl get pods -l app=ngservers -o wide`
    - `kubectl expose deploy/webui --type=NodePort --port=80 // assume allocate port 31207`
    - `http://0.0.0.0:31207/index.html`
Recap: Microservices Architecture

Implementation: Required Functions

- **Tracking state**
  - cluster resource state (e.g., nodes status)
  - container state (e.g., container running or stopped, ...)
  - user requested target state (e.g., #replicas, ...)

- **Orchestration**
  - UI
  - controller-manager: pod/container set up on node, feedback control loop between current state and user request target state
  - scheduling: pod (containers) -> node binding
Recap: Microservice Architecture Implementation: Kubernetes Design

Coordination and fault-tolerance through a reliable data store
- Different functions coordinate through the data store to divide tasks
- Replicas of same function coordinate for fault-tolerance

%kubectl -n kube-system get all
Exercise: See any services used in its own implementation?
Recap: Controller-Manager+Scheduler+Etcld Work Flow

Exercises:
- How do the components get different tasks?
- Does the controller-manager need to know who the scheduler is?
- Does the controller-manager need to know who the node controllers (kubelet) are?
- What is a possible design of having a backup scheduler?
- What may etcd API (services) look like?
- What are some key performance requirements?
etcd-v3 Services and Requirements

- Three APIs (https://etcd.io/docs/v3.5/learning/api/)
  - KV: Creates, updates, fetches, and deletes key-value pairs
  - Watch: Monitors changes to keys
  - Lease: A client can own a key subspace and need to refresh

- Details shortly
etcd Core Requirements and Implementation

- **Requirements**
  - Consistent
    - Many design points
    - One of the cleanest: *linearized* operations
      - Concurrent requests from multiple clients are executed in a sequential/linear order
  - Available
    - Available despite (some) server crashes

- **Implementation**
  - Based on RAFT, a consistent, replicated log protocol
Outline

- Admin and recap
- Microservice based architecture
  - Overview and background
  - Microservices architecture services (Kubernetes)
  - Microservices architecture design and implementation
    - Overview
    - Consistent, replicated log protocol realizing consistent, available controller data store
We focus on an asymmetric, leader-based consistent protocol
  - At any given time, one server is in charge, others accept its decisions
  - Clients communicate with the leader
  - Leaders ensure concurrent requests are serialized in a single, global order

A replicated operations log protocol
  - Replication => surviving leader crash (a server has state when old leader crashes and this server becomes the new leader)

Note: Assumed failure model: fail-stop (not Byzantine), delayed/lost messages
Full Raft Protocol Spec

Instead of looking at the full specification, we develop the basic ideas in steps.
Outline

- Admin and recap
- Microservice based architecture
  - Overview and background
  - Microservices architecture services (Kubernetes)
  - Microservices architecture design and implementation
    - Overview
    - Scheduler and controller-manager algorithms
    - Consistent, replicated log protocol realizing controller data store
      - Basic operations
Basic Operations

- Client sends commands to leader
- Leader appends command to its log
- Leader sends AppendEntries commands to followers (called commit)
- Leader executes command to its state machine, returns result to client

```
put deploy/nginx --image=nginx
```

```
delete deploy/nginx
```
Basic Operations: Detecting Failures

- Problem: leader can crash, need detection
- Q: How may a client detect leader crash?
  - A: Client request should have time out

- Q: How can a server detect leader crash?
  - A: Leader sends heartbeats to maintain authority
  - If no heartbeats (typically 100-500ms), assume leader crashes
  (what does a server do after detecting leader crash?)
Basic Operations: Leader Election Finite State Machine

1. **Start**: Follower
2. **Timeout, Start Election**: Candidate
3. **Timeout, New Election**: Leader
4. **Loss Election**: Follower
5. **Win Election**: Candidate
6. **“Step Down”**: Leader
Leader Election Requirements and Mechanisms

- **Safety** requirement: at any time the system has at most one leader => allow at most one winner per election
  - Mechanism: Winner must get majority vote, and each server gives out only one vote per term (persist on disk)
    - Each election is identified by a number (term)

- **Liveness** requirement: some candidate must eventually win
  - Mechanism: To avoid livelock (persistent split votes), choose election timeouts randomly in $[T, 2T]$
    - Works well if $T \gg$ broadcast time, one server usually times out and wins election before others wake up
Basic Election: Big Picture

- Time divided into terms:
  - Election
  - Normal operation under a single leader
- At most one leader per term
  - Some terms have no leader (failed election)
- Each server maintains current term value
Basic Election: Details

- Increment current term
- Change to Candidate state
- Vote for self

Send RequestVote RPCs to all other servers, retry until one of the following:

1. Receive votes from majority of servers:
   - Become leader
   - Send AppendEntries heartbeats to all other servers
2. Receive RPC from valid leader:
   - Return to follower state
3. No-one wins election (election timeout elapses):
   - Increment term, start new election
Exercises

- Why not assign an ID to each server and elect the one with the highest seen ID?

- What is a simple proof that majority vote (> N/2, where N is total servers) can ensure safety?

Invariant established so far: there is at most one node elected as the leader for a term. Given a term with a winner, there is a unique leader node associated.
Due to leader change, at beginning of new leader's term:

- Old leader may have left entries partially replicated
- Multiple crashes can leave many extraneous log entries:

  * Consistency requires that leader changes will not lead to operation commit order inconsistency.
Raft Design Decision on Leader Commit w/ Leader Changes

- Leader’s log is “the truth”
  - If a leader has decided that a log entry is committed, that entry will be present in the logs of all future leaders
  - No special steps by a new leader to revise leader log [in some alternative design]

- Leader will eventually make follower’s logs identical to leader’s
Leader Commit w/ Leader Change

Q: Can the leader (s1) commit Log[4]?

A: If only simple majority leader election, if s1 crashes right after commit, S5 may become the next leader, but S5 misses Log[4].

Exercise: Simple revision to election rule so that S5 cannot be the next leader?
Revised Leader Election Rule

- During elections, choose candidate with log from newer terms
  - Candidates include log info in RequestVote RPCs
    (index & term of last log entry)
  - Voting server V denies vote if its log is “more complete“:
    \[(\text{lastLogTerm}_V > \text{lastLogTerm}_C) \lor \quad (\text{lastLogTerm}_V == \text{lastLogTerm}_C) \land (\text{lastLogIndex}_V > \text{lastLogIndex}_C)\]
  - Leader will likely have “most complete” log among electing majority
Exercise: Leader Commit w/ Leader Change

Q: Can S5 become the next leader under revised election rule?
Outline

- Admin and recap
- **Microservice based architecture**
  - Overview and background
  - Microservices architecture services (Kubernetes)
  - Microservices architecture design and implementation
    - Overview
    - Scheduler and controller-manager algorithms
    - Consistent, replicated log protocol realizing controller data store
      - Basic operations
      - Leader election rule
      - Commitment rule
**Leader Commit: Continue**

Q: Assume revised leader election, can the leader (s1) commit Log[3] in this case?

A: The revised leader election may still elect S5, again causing problem: If elected, it will overwrite entry 3 on s₁, s₂, and s₃!

Exercise: A simple revision to commit rule so that Log[3] cannot be committed?
Revised Commitment Rule

- For a leader to decide an entry is committed:
  - Must be stored on a majority of servers
  - At least one new entry from leader’s term must also be stored on majority of servers

- Once entry 4 stored on majority:
  - \( s_5 \) cannot be elected leader for term 5
  - Entries 3 and 4 both safe
Exercise: Revised Commitment Rule

Q: Does the revised leader election rule plus the revised commit rule ensure safety in this case?
**Log Consistency Invariant**

High level of coherency between logs:

- If log entries on different servers have same index and term:
  - They store the same command
  - The logs are identical in all preceding entries

- If a given entry is committed, all preceding entries are also committed

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>add</td>
<td>cmp</td>
<td>ret</td>
<td>mov</td>
<td>jmp</td>
<td>div</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>add</td>
<td>cmp</td>
<td>ret</td>
<td>mov</td>
<td>jmp</td>
<td>sub</td>
</tr>
</tbody>
</table>
AppendEntries Consistency Check

- Each AppendEntries RPC contains index, term of entry preceding new ones
- Follower must contain matching entry; otherwise it rejects request (see later for how this is used)
- Implements an induction step, ensures coherency

```
leader
1 add 1 cmp 1 ret 2 mov 3 jmp
follower
1 add 1 cmp 1 ret 2 mov

AppendEntries succeeds: matching entry

leader
1 add 1 cmp 1 ret 2 mov 3 jmp
follower
1 add 1 cmp 1 ret 1 shl

AppendEntries fails: mismatch
```
Details: AppendEntries to Follower Logs

- Leader keeps nextIndex for each follower:
  - Index of next log entry to send to that follower
  - Initialized to (1 + leader’s last index)
- When AppendEntries consistency check fails, decrement nextIndex and try again:

![Diagram showing log index and leader for term 7 with followers.](image-url)
Details: AppendEntries to Follower Logs, cont’d

- When follower overwrites inconsistent entry, it deletes all subsequent entries:

```
+---+---+---+---+---+---+---+---+---+---+---+
|   |   |   |   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+
| 1 | 1 | 1 | 4 | 4 | 5 | 5 | 6 | 6 | 6 |   |
+---+---+---+---+---+---+---+---+---+---+---+
| 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 |
+---+---+---+---+---+---+---+---+---+---+---+
| 1 | 1 | 1 | 4 |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+---+---+---+
```

**log index** 1 2 3 4 5 6 7 8 9 10 11

**leader for term 7**

**follower (before)**

**follower (after)**
Summary: Raft Election and Commit

- Use a combination of election rule, commitment rule, and update-follower rule to achieve safety

Commitment correctness depends on update rule

  - Client sends commands to leader
  - Leader appends command to its log
  - Leader sends **AppendEntries commands** to followers and blocks until **commit**:
    - Successful on a follower requires log consistency check
    - Must be successful on a majority of servers
    - At least one new entry from leader’s term must also be successful on majority of servers
  - Leader **executes** command to its state machine, returns result to client
Exercise

- Which logs can s1 commit?
- Who may become next leader if s1 crashes or is disconnected?
- Sanity check: If a log is committed by the leader, the next leader should have the entry committed as well.
Exercise

- Which logs can $s_1$ commit?
- Who may become next leader if $s_1$ crashes or is disconnected?
- Sanity check: If a log is committed by the leader, the next leader should have the entry committed as well.
Exercise

- Which logs can s1 commit?
- Who may become next leader if s1 crashes or is disconnected?
- Sanity check: If a log is committed by the leader, the next leader should have the entry committed as well.
Exercise

- Which logs can $s_1$ commit?
- Who may become next leader if $s_1$ crashes or is disconnected?
- Sanity check: If a log is committed by the leader, the next leader should have the entry committed as well.
Exercise

- Consider server $S_L$, who is the current leader with term $t$
  
  $S_L \begin{array}{c} \text{1} \\ \text{t} \end{array}$

- The $t$-term entries will be committed in sequential order by $S_L$ [T | F]

- Consider the last $t$-term entry committed to a set $S$ of servers. If $S_L$ crashes, only one of those in $S$ can become the next leader [T | F]
Outline

- Admin and recap
- Microservice based architecture
  - Overview and background
  - Microservices architecture services (Kubernetes)
  - Microservices architecture design and implementation
    - Overview
    - Scheduler and controller-manager algorithms
    - Consistent, replicated log protocol realizing controller data store
      - Basic operations
      - Leader election rule
      - Commitment rule
      - Client protocol issue and fix
Introducing Clients: Raft as Client/Server Protocol

- Client sends commands to leader
  - If leader unknown, contact any server
  - If contacted server not leader, it will redirect to leader
  - If request times out (e.g., leader crash):
    - Client reissues command to some other server
    - Eventually redirected to new leader
    - Retry request with new leader

- Leader response
  - Leader does not respond until command has been logged, committed, and executed by leader’s state machine
Introducing Clients: Raft as Client/Server Protocol

- Problem: a leader crashes after executing command, but before responding
  - Must not execute command twice
- Solution: client embeds a unique id in each command
  - Server includes id in log entry
  - Before accepting command, leader checks its log for entry with that id
  - If id found in log, ignore new command, return response from old command
- Result: exactly-once semantics as long as client doesn’t crash
Outline

- Admin and recap
- Microservice based architecture
  - Overview and background
  - Microservices architecture services (Kubernetes)
  - Microservices architecture design and implementation
    - Overview
    - Scheduler and controller-manager algorithms
    - Consistent, replicated log protocol realizing controller data store
      - Basic operations
      - Leader election rule
      - Commitment rule
      - Client protocol issue and fix
      - Configuration change [optional]
**Offline: Configuration Changes**

- **System configuration:**
  - ID, address for each server
  - Determines what constitutes a majority

- **Consensus mechanism must support changes in the configuration:**
  - Replace failed machine
  - Change degree of replication
Cannot switch directly from one configuration to another: conflicting majorities could arise
Offline: Joint Consensus

- Raft uses a 2-phase approach:
  - Intermediate phase uses joint consensus (need majority of both old and new configurations for elections, commitment)
  - Configuration change is just a log entry; applied immediately on receipt (committed or not)
  - Once joint consensus is committed, begin replicating log entry for final configuration
Offline: Joint Consensus, cont’d

- Additional details:
  - Any server from either configuration can serve as leader
  - If current leader is not in $C_{new}$, must step down once $C_{new}$ is committed.

![Diagram showing offline joint consensus process](image-url)
### Raft Protocol Summary

#### Followers
- Respond to RPCs from candidates and leaders.
- Convert to candidate if election timeout elapses without either:
  - Receiving valid AppendEntries RPC, or
  - Granting vote to candidate.

#### Candidates
- Increment currentTerm, vote for self.
- Reset election timeout.
- Send RequestVote RPCs to all other servers, wait for either:
  - Votes received from majority of servers: become leader.
  - AppendEntries RPC received from new leader: step down.
  - Election timeout elapses without election resolution: increment term, start new election.
  - Discover higher term: step down.

#### Leaders
- Initialize nextIndex for each to last log index + 1.
- Send initial empty AppendEntries RPCs (heartbeat) to each follower; repeat during idle periods to prevent election timeouts.
- Accept commands from clients, append new entries to local log.
- Whenever last log index ≥ nextIndex for a follower, send AppendEntries RPC with log entries starting at nextIndex, update nextIndex if successful.
- If AppendEntries fails because of log inconsistency, decrement nextIndex and retry.
- Mark log entries committed if stored on a majority of servers and at least one entry from current term is stored on a majority of servers.
- Step down if currentTerm changes.

#### Persistent State
Each server persists the following to stable storage synchronously before responding to RPCs:
- `currentTerm`: latest term server has seen (initialized to 0 on first boot).
- `votedFor`: candidateId that received vote in current term (or null if none).
- `log[]`: log entries.

#### Log Entry
- `term`: term when entry was received by leader.
- `index`: position of entry in the log.
- `command`: command for state machine.

#### RequestVote RPC
Invoked by candidates to gather votes.

**Arguments:**
- `candidateId`: candidate requesting vote.
- `term`: candidate's term.
- `lastLogIndex`: index of candidate's last log entry.
- `lastLogTerm`: term of candidate's last log entry.

**Results:**
- `term`: currentTerm, for candidate to update itself.
- `voteGranted`: true means candidate received vote.

**Implementation:**
1. If `term > currentTerm`, `currentTerm ← term` (step down if leader or candidate).
2. If `term == currentTerm`, `votedFor` is null or `candidateId`, and candidate's log is at least as complete as local log, grant vote and reset election timeout.

#### AppendEntries RPC
Invoked by leader to replicate log entries and discover inconsistencies; also used as heartbeat.

**Arguments:**
- `term`: leader's term.
- `leaderId`: so follower can redirect clients.
- `prevLogIndex`: index of log entry immediately preceding new ones.
- `prevLogTerm`: term of `prevLogIndex` entry.
- `entries[]`: log entries to store (empty for heartbeat).
- `commitIndex`: last entry known to be committed.

**Results:**
- `term`: currentTerm, for leader to update itself.
- `success`: true if follower contained entry matching `prevLogIndex` and `prevLogTerm`.

**Implementation:**
1. Return if `term < currentTerm`.
2. If `term == currentTerm`, `currentTerm ← term`.
3. If candidate or leader, step down.
4. Reset election timeout.
5. Return failure if log doesn't contain an entry at `prevLogIndex` whose term matches `prevLogTerm`.
6. If existing entries conflict with new entries, delete all existing entries starting with first conflicting entry.
7. Append any new entries not already in the log.
8. Advance state machine with newly committed entries.
Etcd KV API: Range/Put/Delete

Exercise: How may the API be implemented on top of RAFT?
message TxnRequest {
    repeated Compare compare = 1;
    repeated RequestOp success = 2;
    repeated RequestOp failure = 3;
}

message RequestOp {
    // request is a union of request types accepted
    oneof request {
        RangeRequest request_range = 1;
        PutRequest request_put = 2;
        DeleteRangeRequest request_delete_range = 3;
    }
}

message TxnResponse {
    ResponseHeader header = 1;
    bool succeeded = 2;
    repeated ResponseOp responses = 3;
}
Etcd Watch API

```java
message WatchCreateRequest {
  bytes key = 1;
  bytes range_end = 2;
  int64 start_revision = 3;
  bool progress_notify = 4;
}

enum FilterType {
  NOPUT = 0;
  NODELETE = 1;
}

message WatchResponse {
  ResponseHeader header = 1;
  int64 watch_id = 2;
  bool created = 3;
  bool canceled = 4;
  int64 compact_revision = 5;

  repeated mvccpb.Event events = 11;
}

message Event {
  enum EventType {
    PUT = 0;
    DELETE = 1;
  }
  EventType type = 1;
  KeyValuе kv = 2;
  KeyValuе prev_kv = 3;
}

message WatchCancelRequest {
  int64 watch_id = 1;
}
```
Etcd Lease API

```protobuf
message LeaseGrantRequest {
  int64 TTL = 1;
  int64 ID = 2;
}
```

```protobuf
message LeaseGrantResponse {
  ResponseHeader header = 1;
  int64 ID = 2;
  int64 TTL = 3;
}
```

```protobuf
message LeaseKeepAliveRequest {
  int64 ID = 1;
}
```

```protobuf
message LeaseKeepAliveResponse {
  ResponseHeader header = 1;
  int64 ID = 2;
  int64 TTL = 3;
}
```

```protobuf
message LeaseRevokeRequest {
  int64 ID = 1;
}
```
Additional Details

- https://blog.containership.io/etcd/
  for etcd and RAFT, in particular the visual guide
- https://github.com/etcd-io/etcd
- http://thesecretlivesofdata.com/raft/
- https://etcd.io/docs/v3.5/learning/
Outline

- Admin and recap
- Microservice based architecture
  - Overview and background
  - Microservices architecture services (Kubernetes)
  - Microservices architecture design and implementation
    - Overview
    - Scheduler and controller-manager implementation
    - Consistent, replicated log protocol implementation
    - Pod implementation
Recap: Node, Pod, and Container
Recall: Namespace

- Linux has multiple types of namespaces, e.g., PID, network (NET), IPC, mount namespace (NS), IPC, identify (UTS)
- Each process belongs to a namespace of each type, and only sees those in the same namespace

```c
int clone(int (*child_func)(void *), void *child_stack, int flags, void *arg);
```
execv(child_args[0], child_args);
printf("Doops\n");
return 1;
}

int main()
{
    // init sync primitive
    pipe(checkpoint);
    printf(" - [%5d] Hello ?\n", getpid());

    int child_pid = clone(child_main, child_stack+STACK_SIZE, 
                        CLONE_NEWUTS | CLONE_NEWIPC | CLONE_NEWPID | CLONE_NEWNS | CLONE_NEWNET | 
                        SIGCHLD, NULL);

    // further init: create a veth pair
    char* cmd;
asprintf(&cmd, "ip link set veth1 netns %d", child_pid);
system("ip link add veth0 type veth peer name veth1");
system(cmd);
system("ip link set veth0 up");
system("ip addr add 169.254.1.1/30 dev veth0");
free(cmd);

    // signal "done"
    close(checkpoint[1]);

    waitpid(child_pid, NULL, 0);
    return 0;
}
Pod Structure

- Demo (MacOS)
  - kubectl get pod
  - docker ps

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#define STRINGIFY(s) #s
#define VERSION_STRING(s) STRINGIFY(s)

#ifndef VERSION
#define VERSION "0.0.1"
#endif

static void sigwarn(int sig) {
    perror(sig, "Shutting down, got signal");
    exit(0);
}

static void sigresp(int sig) {
    while (waitpid(-1, NULL, WNOHANG) > 0) {
    }
}

int main(int argc, char **argv) {
    int i;
    for (i = 1; i < argc; ++i) {
        if (strcmp(argv[i], "-v")) {
            printf("%s", VERSION_STRING(VERSION));
            return 0;
        }
    }
    if (getpid() == 1) {
        /* Not an error because pause sees use outside of infra containers. */
        fprintf(stderr, "Warning: pause should be the first process'utile;);
    }
    if (sigaction(SIGINT, &struct sigaction{.sa_handler = sigwarn}, NULL) < 0)
        return 1;
    if (sigaction(SIGTERM, &struct sigaction{.sa_handler = sigresp}, NULL) < 0)
        return 2;
    if (sigaction(SIGCHLD, &struct sigaction{.sa_handler = sigresp,
                                             .sa_flags = SA_RESTART},
                 NULL) < 0)
        return 3;
    for (;;) {
        pause();
        fprintf(stderr, "Error: infinite loop terminated\n");
        return 42;
    }
}
```
Simulate Pod using Docker (Structure)

```
# docker run -d --name pause gcr.io/google_containers/pause-amd64:3.0

# cat <<EOF >> nginx.conf
>    error_log stderr;
>    events { worker_connections 1024; }
>    http {
>        access_log /dev/stdout combined;
>        server {
>            listen 80 default_server;
>            server_name example.com www.example.com;
>            location / {
>                proxy_pass http://127.0.0.1:2368;
>            }
>        }
>    }
> EOF
```

# docker run -d --name ghost --net = container: pause --ipc = container: pause --pid = container: pause ghost

```
# docker run -d --name nginx -v `pwd`/nginx.conf:/etc/nginx/nginx.conf -p 8080:80 --net=container:pause --ipc=container:pause --pid=container:pause nginx
```
Pod Construction (Structure)

- Initialize Pause network namespace
- Init Pause cont. net device
- Cross-node networking
Outline

- Admin and recap
- Microservice based architecture
  - Overview and background
  - Microservices architecture services (Kubernetes)
  - Microservices architecture design and implementation
    - Overview
    - Scheduler and controller-manager implementation
    - Consistent, replicated log protocol implementation
    - Pod implementation
    - Networking implementation
      - Network namespaces
        » Basic network namespace
Basic Network Namespace: Demo

- Show network namespaces
  - `ip netns list`
- Create a network namespace
  - `ip netns add netns1`
- Show devices in the network namespace
  - `ip netns exec netns1 ip link list`
  - `ip netns exec netns1 ifconfig <dev>`
- Show routing tables/iptables/sockets
  - `ip netns exec netns1 route // or netstat -rn`
  - `ip netns exec netns1 iptables -L // or netstat -rn`
  - `ip netns exec netns1 netstat -p tcp`
- Ping localhost
  - `ip netns exec netns1 ping 127.0.0.1`
- Contrast w/ default name space
Basic Network Namespace: Demo

- Turn up lo
  - `ip netns exec netns1 ip link set dev lo up` // or `ifconfig` 

- Ping
  - `ip netns exec netns1 ping 127.0.0.1`
  - `ip netns exec netns1 ifconfig lo`

- Delete network namespace [if you want, but not now]
  - `ip netns delete netns1`
Outline

- Admin and recap
- Microservice based architecture
  - Overview and background
  - Microservices architecture services (Kubernetes)
  - Microservices architecture design and implementation
    - Overview
    - Scheduler and controller-manager implementation
    - Consistent, replicated log protocol implementation
    - Pod implementation
    - Networking implementation
      - Network namespaces
        » Basic network namespace
        » Network namespace “networking”
Out of Network Namespace “Jail”: veth

- A pair of virtual ethernet devices
  - Packet enters one end will appear in the other end
  - A common way to allow a namespace to communicate outside

Source code: drivers/net/veth.c
Linux veth Pair driver: Demo

- **Create veth pair**
  - `ip link add veth0 type veth peer name veth1`
  - `ip link list // list devices in default namespace`

- **Move one of them to different namespace**
  - `ip link set veth1 netns netns1`
  - `ip link list // devices in default namespace`
  - `ip netns exec netns1 ip link list // devices in netns1`

- **Assign IP addresses and turn on devices**
  - `ip netns exec netns1 netstat -rn // see routing table`
  - `ip netns exec netns1 ifconfig veth1 10.1.1.1/24 up // check routing again`
  - `ip netns exec netns1 ifconfig veth1`
  - **Exercise**: assign veth0 10.1.1.2 and turn on; compare routing tables of netns1 and root

- **Exercise**: in netns1 ping 10.1.1.2, and host 10.1.1.1
Exercise

- If there are N network namespaces (e.g., those inside a k8s Pod) on a single node, how many veth pairs do we need to set up so that they all can communicate?
Linux Bridge: Demo

- Create a linux bridge
  - `ip link add name br0 type bridge // or brctl addbr br0`
- Add veth0 to the bridge
  - `ip link set dev veth0 master br0 // or brctl addif br0 veth0`
- Display devices on br0
  - `brctl show`
Offline: Docker Exercise

- **Start a container using docker**
  - `ip link show // or ifconfig` see current devices
  - `docker run -d nginx` // `ifconfig` to see devices again
  - `docker inspect <container id> // find PID; see IP address, gateway`
  - `ls -l /proc/<pid>/ns // see the namespaces`

- **Docker networking**
  - `brctl show // Show all bridges`
  - `ifconfig <bridge>`

- **See docker container network namespace**
  - `ip netns attach <name> <pid>`
  - `ip netns exec <name> ip link show`

```bash
# docker inspect 6f5 | grep Pid
"Pid": 15818,
# ll /proc/15818/ns/
total 0
lrwxrwxrwx 1 root root 0 Jan 10 16:16 ipc -> ipc:[4026535715]
lrwxrwxrwx 1 root root 0 Jan 10 16:16 mnt -> mnt:[4026535789]
lrwxrwxrwx 1 root root 0 Jan 10 16:16 net -> net:[4026535718]
lrwxrwxrwx 1 root root 0 Jan 10 16:16 pid -> pid:[4026535791]
lrwxrwxrwx 1 root root 0 Jan 10 16:18 user -> user:[4026531887]
 lrwxrwxrwx 1 root root 0 Jan 10 16:16 uts -> uts:[4026535790]
```
Offline: Docker Exercise

- Start a container using docker with port mapping
  - docker run -p 6789:80 -d nginx
  - docker inspect <container id> // find PID: see IP address, gateway

- Docker networking addition
  - iptables -t nat -nL

- Source rewrite (to visit outside)
- Dest rewrite (port map)

```
# docker inspect 6f5 | grep Pid
"Pid": 16818,
# ll /proc/16818/ns/
total 0
lrwxrwxrwx 1 root root 0 Jan 10 16:16 ipc -> ipc:[4026535715]
lrwxrwxrwx 1 root root 0 Jan 10 16:16 mnt -> mnt:[4026535799]
lrwxrwxrwx 1 root root 0 Jan 10 16:16 net -> net:[4026535718]
lrwxrwxrwx 1 root root 0 Jan 10 16:16 pid -> pid:[4026535791]
lrwxrwxrwx 1 root root 0 Jan 10 16:16 user -> user:[4026531837]
lrwxrwxrwx 1 root root 0 Jan 10 16:16 uts -> uts:[4026535790]
```
Outline

- Admin and recap
- Microservice based architecture
  - Overview and background
  - Microservices architecture services (Kubernetes)
  - Microservices architecture design and implementation
    - Overview
    - Scheduler and controller-manager implementation
    - Consistent, replicated log protocol implementation
    - Pod implementation
    - Networking implementation
      - Network namespaces
      - Cross-node networking
Recall: Node, Pod, Networking Requirements

- **IP-Per-Pod**: Each Pod has a single IP from the range
  - Pods are allocated IP addresses from a given range
  - Pods run in a direct-connect, flat address space
    - Containers in the same Pod share IP, same Linux network stack, and can connect using localhost
Setting

- **IP-Per-Pod:** Each Pod has a single IP from the range

Diagram showing Node1 and Node2 with Pods, containers, and networking details.

Discussion: Key Issues?
Key Issues (Cross Nodes)

- IP address management (ipam): potentially overlapping addresses if not careful
  - Key solution: Address allocation: in the whole Kubernetes cluster, Pod IP allocation should have no conflict

- Routing: Virtual IP and virtual MAC may not be recognized in the physical network

Good read: Container Networking Interface:
https://github.com/containernetworking/cni/blob/master/SPEC.md#container-network-interface-cni-specification
Outline

- Admin and recap
- Microservice based architecture
  - Overview and background
  - Microservices architecture services (Kubernetes)
  - Microservices architecture design and implementation
    - Overview
    - Scheduler and controller-manager implementation
    - Consistent, replicated log protocol implementation
    - Pod implementation
    - Networking implementation
      - Network namespaces
      - Cross-node networking
      » flannel
Flannel Overview

- No server
  - There is one agent on each Node

- Coordinating using etcd to solve the ipam and routing issue (how?)
**Flannel: Workflow (ipam)**

- Use etcd to coordinate
  - Share a large network address space and post to etcd
    ```bash
    # etcdctl set /coreos.com/network/config '{ "Network": "172.17.0.0/16" }'
    ```
  - Partition the space and get a subnet
    ```bash
    # etcdctl ls /coreos.com/network/subnets
    /coreos.com/network/subnets/172.17.18.0-24
    /coreos.com/network/subnets/172.17.19.0-24
    /coreos.com/network/subnets/172.17.20.0-24
    ```

- Modify node docker start parameter to allocate pod IP in the range, e.g.,
  - `--bip=172.17.18.0/24`
Flannel: Workflow (Routing)

- How does a flannel agent/node know the routing
  - From etcd knows the prefix of each node
    - Build routing using the etcd data

```bash
# etcdctl ls /coreos.com/network/subnets
/coreos.com/network/subnets/172.17.18.0-24
/coreos.com/network/subnets/172.17.19.0-24
/coreos.com/network/subnets/172.17.20.0-24

# etcdctl get /coreos.com/network/subnets/172.17.18.0-24
{"PublicIP":"192.168.14.97"}
# etcdctl get /coreos.com/network/subnets/172.17.19.0-24
{"PublicIP":"192.168.14.98"}
# etcdctl get /coreos.com/network/subnets/172.17.20.0-24
{"PublicIP":"192.168.14.100"}
```
Flannel Routing (Backend.Type)

- Overlay (tunnel)
  - UDP
  - VXLAN
- Non overlay
  - Host-Gateway (must be a L2 network)
Overlay Networking

- **Tunnel**: Allows user space applications to process packets
- **Example**: Design a VPN server that encapsulates packets to a given destination set (e.g., 192.168.1.0/24) using an external encapsulation
#include <net/if.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <linux/if_tun.h>
#include <stdlib.h>
#include <stdio.h>

int tun_alloc(int flags)
{
    struct ifreq ifr;
    int fd, err;
    char *clonedev = "/dev/net/tun";

    if ((fd = open(clonedev, O_RDWR)) < 0) {
        return fd;
    }

    memset(&ifr, 0, sizeof(ifr));
    ifr.ifr_flags = flags;

    if ((err = ioctl(fd, TUNSETIFF, (void *) &ifr)) < 0) {
        close(fd);
        return err;
    }

    printf("Open tun/tap device: %s for reading...\n", ifr.ifr_name);

    return fd;
}

int main()
{
    int tun_fd, nread;
    char buffer[1500];

    /* Flags: IFF_TUN  - TUN device (no Ethernet headers)
     *         IFF_TAP  - TAP device
     *         IFF_NO_PI - Do not provide packet information
     */
    tun_fd = tun_alloc(IFF_TUN | IFF_NO_PI);

    if (tun_fd < 0) {
        perror("Allocating interface");
        exit(1);
    }

    while (1) {
        nread = read(tun_fd, buffer, sizeof(buffer));
        if (nread < 0) {
            perror("Reading from interface");
            close(tun_fd);
            exit(1);
        }
        printf("Read %d bytes from tun/tap device\n", nread);
    }
    return 0;
}
Offline Tunnel Example: Execution and Configuration

- gcc tun.c -o my_tun
- ./my_tun // assume created tunnel 0 tun0; ip addr
- ip addr add 192.168.1.2/24 dev tun0
- ip link set tun0 up

Exercise: packet flow to ping 192.168.1.3

More detail on using command line (mknod) to create, on frame format, see https://www.kernel.org/doc/html/latest/networking/tuntap.html
Calico: BGP vRouter