Raft library

Raft is a protocol with which a cluster of nodes can maintain a replicated state machine. The state machine is kept in sync through the use of a replicated log. For more details on Raft, see "In Search of an Understandable Consensus Algorithm" (https://raft.github.io/raft.pdf) by Diego Ongaro and John Ousterhout.

This Raft library is stable and feature complete. As of 2016, it is **the most widely used** Raft library in production, serving tens of thousands clusters each day. It powers distributed systems such as etcd, Kubernetes, Docker Swarm, Cloud Foundry Diego, CockroachDB, TiDB, Project Calico, Flannel, Hyperledger and more.

Most Raft implementations have a monolithic design, including storage handling, messaging serialization, and network transport. This library instead follows a minimalistic design philosophy by only implementing the core raft algorithm. This minimalism buys flexibility, determinism, and performance.

To keep the codebase small as well as provide flexibility, the library only implements the Raft algorithm; both network and disk IO are left to the user. Library users must implement their own transportation layer for message passing between Raft peers over the wire. Similarly, users must implement their own storage layer to persist the Raft log and state.

In order to easily test the Raft library, its behavior should be deterministic. To achieve this determinism, the library models Raft as a state machine. The state machine takes a `Message` as input. A message can either be a local timer update or a network message sent from a remote peer. The state machine's output is a 3-tuple `{[]Messages, []LogEntries, NextState}` consisting of an array of `Messages`, `log entries`, and `Raft state changes`. For state machines with the same state, the same state machine input should always generate the same state machine output.

A simple example application, *raftexample*, is also available to help illustrate how to use this package in practice: https://github.com/etcd-io/etcd/tree/main/contrib/raftexample

Features

This raft implementation is a full feature implementation of Raft protocol. Features includes:

- Leader election
- Log replication
- Log compaction
- Membership changes
- Leadership transfer extension

- Efficient linearizable read-only queries served by both the leader and followers
  - leader checks with quorum and bypasses Raft log before processing read-only queries
  - followers asks leader to get a safe read index before processing read-only queries
- More efficient lease-based linearizable read-only queries served by both the leader and followers
  - leader bypasses Raft log and processing read-only queries locally
  - followers asks leader to get a safe read index before processing read-only queries
  - this approach relies on the clock of the all the machines in raft group

This raft implementation also includes a few optional enhancements:

- Optimistic pipelining to reduce log replication latency
- Flow control for log replication
- Batching Raft messages to reduce synchronized network I/O calls
- Batching log entries to reduce disk synchronized I/O
- Writing to leader's disk in parallel
- Internal proposal redirection from followers to leader
- Automatic stepping down when the leader loses quorum
- Protection against unbounded log growth when quorum is lost

Notable Users

- cockroachdb A Scalable, Survivable, Strongly-Consistent SQL Database
- dgraph A Scalable, Distributed, Low Latency, High Throughput Graph Database
- etcd A distributed reliable key-value store
- tikv A Distributed transactional key value database powered by Rust and Raft
- swarmkit A toolkit for orchestrating distributed systems at any scale.
- chain core Software for operating permissioned, multi-asset blockchain networks

Usage

The primary object in raft is a Node. Either start a Node from scratch using raft.StartNode or start a Node from some initial state using raft.RestartNode.

To start a three-node cluster

```
storage := raft.NewMemoryStorage()
c := &raft.Config{
  ID:              0x01,
  ElectionTick:    10,
  HeartbeatTick:   1,
  Storage:         storage,
  MaxSizePerMsg:   4096,
  MaxInflightMsgs: 256,
}
// Set peer list to the other nodes in the cluster.
// Note that they need to be started separately as well.
n := raft.StartNode(c, []raft.Peer{{ID: 0x02}, {ID: 0x03}})
```

Start a single node cluster, like so:

```
// Create storage and config as shown above.
// Set peer list to itself, so this node can become the leader of this single-node cluste
peers := []raft.Peer{{ID: 0x01}}
n := raft.StartNode(c, peers)
```

To allow a new node to join this cluster, do not pass in any peers. First, add the node to the existing cluster by calling `ProposeConfChange` on any existing node inside the cluster. Then, start the node with an empty peer list, like so:

```
// Create storage and config as shown above.
n := raft.StartNode(c, nil)
```

To restart a node from previous state:

```
storage := raft.NewMemoryStorage()

// Recover the in-memory storage from persistent snapshot, state and entries.
storage.ApplySnapshot(snapshot)
storage.SetHardState(state)
storage.Append(entries)

c := &raft.Config{
  ID:              0x01,
  ElectionTick:    10,
  HeartbeatTick:   1,
  Storage:         storage,
  MaxSizePerMsg:   4096,
  MaxInflightMsgs: 256,
}

// Restart raft without peer information.
// Peer information is already included in the storage.
n := raft.RestartNode(c)
```

After creating a Node, the user has a few responsibilities:

First, read from the Node.Ready() channel and process the updates it contains. These steps may be performed in parallel, except as noted in step 2.

1. Write Entries, HardState and Snapshot to persistent storage in order,

i.e. Entries first, then HardState and Snapshot if they are not empty. If persistent storage supports atomic writes then all of them can be written together. Note that when writing an Entry with Index i, any previously-persisted entries with Index >= i must be discarded.

2. Send all Messages to the nodes named in the To field. It is important that no messages be sent until the latest HardState has been persisted to disk, and all Entries written by any previous Ready batch (Messages may be sent while entries from the same batch are being persisted). To reduce the I/O latency, an optimization can be applied to make leader write to disk in parallel with its followers (as explained at section 10.2.1 in Raft thesis). If any Message has type MsgSnap, call Node.ReportSnapshot() after it has been sent (these messages may be large). Note: Marshalling messages is not thread-safe; it is important to make sure that no new entries are persisted while marshalling. The easiest way to achieve this is to serialise the messages directly inside the main raft loop.

3. Apply Snapshot (if any) and CommittedEntries to the state machine. If any committed Entry has Type EntryConfChange, call Node.ApplyConfChange() to apply it to the node. The configuration change may be cancelled at this point by setting the NodeID field to zero before calling ApplyConfChange (but ApplyConfChange must be called one way or the other, and the decision to cancel must be based solely on the state machine and not external information such as the observed health of the node).

4. Call Node.Advance() to signal readiness for the next batch of updates. This may be done at any time after step 1, although all updates must be processed in the order they were returned by Ready.

Second, all persisted log entries must be made available via an implementation of the Storage interface. The provided MemoryStorage type can be used for this (if repopulating its state upon a restart), or a custom disk-backed implementation can be supplied.

Third, after receiving a message from another node, pass it to Node.Step:

```
func recvRaftRPC(ctx context.Context, m raftpb.Message) {
    n.Step(ctx, m)
}
```

Finally, call `Node.Tick()` at regular intervals (probably via a `time.Ticker`). Raft has two important timeouts: heartbeat and the election timeout. However, internally to the raft package time is represented by an abstract "tick".

The total state machine handling loop will look something like this:

```
for {
  select {
  case <-s.Ticker:
      n.Tick()
    case rd := <-s.Node.Ready():
      saveToStorage(rd.HardState, rd.Entries, rd.Snapshot)
      send(rd.Messages)
      if !raft.IsEmptySnap(rd.Snapshot) {
        processSnapshot(rd.Snapshot)
      }
      for _, entry := range rd.CommittedEntries {
        process(entry)
        if entry.Type == raftpb.EntryConfChange {
          var cc raftpb.ConfChange
          cc.Unmarshal(entry.Data)
          s.Node.ApplyConfChange(cc)
        }
      }
      s.Node.Advance()
    case <-s.done:
      return
    }
  }
```

To propose changes to the state machine from the node to take application data, serialize it into a byte slice and call:

```
    n.Propose(ctx, data)
```

If the proposal is committed, data will appear in committed entries with type raftpb.EntryNormal. There is no guarantee that a proposed command will be committed; the command may have to be reproposed after a timeout.

To add or remove node in a cluster, build ConfChange struct 'cc' and call:

```
    n.ProposeConfChange(ctx, cc)
```

After config change is committed, some committed entry with type raftpb.EntryConfChange will be returned. This must be applied to node through:

```
    var cc raftpb.ConfChange
    cc.Unmarshal(data)
    n.ApplyConfChange(cc)
```

Note: An ID represents a unique node in a cluster for all time. A given ID MUST be used only once even if the old node has been removed. This means that for example IP addresses make poor node IDs since they may be reused. Node IDs must be non-zero.

Implementation notes

This implementation is up to date with the final Raft thesis (https://github.com/ongardie/dissertation/blob/maste
although this implementation of the membership change protocol differs somewhat from that described in chapter 4. The key invariant that membership changes happen one node at a time is preserved, but in our implementation the membership change takes effect when its entry is applied, not when it is added to the log (so the entry is committed under the old membership instead of the new). This is equivalent in terms of safety, since the old and new configurations are guaranteed to overlap.

To ensure there is no attempt to commit two membership changes at once by matching log positions (which would be unsafe since they should have different quorum requirements), any proposed membership change is simply disallowed while any uncommitted change appears in the leader's log.

This approach introduces a problem when removing a member from a two-member cluster: If one of the members dies before the other one receives the commit of the confchange entry, then the member cannot be removed any more since the cluster cannot make progress. For this reason it is highly recommended to use three or more nodes in every cluster.

Go docs

More detailed development documentation can be found in go docs: https://pkg.go.dev/go.etcd.io/etcd/raft/v3.