FlexibleK不同于Raft的一点在于，对于不同follower，即使Index与Term相同，它们存储的也可能是不同的数据，但是它们的数据一定来自于一个相同的raft entry。另一方面，由于我们可能使用一直变化的k, m对同一个entry进行encoding，这增加了系统在fault时可能出现故障的频率，所以我们必须要谨慎考虑对算法的修改。

根据raft的论文，`(raft_index, raft_term)`唯一确定了一个entry, 也就是：

> 对于不同的server, 具有相同index和term的entry数据一定相同。

但在FlexibleK中，一个entry可能是fragment, 更严重地是，可能是带有不同encoding参数的fragment。但是我们有：

> `(raft_index, raft_term, k, m, frag_id)`唯一确定了一个entry, 具有相同上述tuple的entry数据相同

实际上，raft提供了处理`(Index, term)` tuple的能力，就是AppendEntries RPC中follower收到一个entry的处理逻辑，通过判断index和term，follower会决定是否将这个entry append到自己的log中，或者删除已有的log。但在FlexibleK中，一个已经被replicate的entry，可能会被要求overwrite，这可能是因为两个原因：

- Leader发现k已经被修改了，它需要重新encoding这个entry
- Leader发现live server的情况发生了变化，例如原先是S1, S2, S3存活，现在是S2, S3, S4存活了。它可能需要改变fragment在这些follower上的分布。

但不管何种原因，这都导致了follower必须要overwrite一个entry(更确切地说，一个fragment)。否则它就违背了leader希望达到的局面，从而导致没有满足commit的要求：例如，leader希望S1备份fragment1, S2备份fragment2，但是S2没有照做，它可能在之前的round已经备份了fragment1, 这导致fragment1在两个follower上被replicate了，从而使得这个entry无法被提交。

因此我们需要一个`Version`来标记每个fragment, 一个tuple：`(k, m, fragment_id)`能够唯一标识一个entry的fragment。通过这样一个version，leader能够为每个要发送的fragment进行标记，在Follower收到AppendEntries RPC之后，follower会回复自己append的这些entry的version；这样leader能正确地知道fragment在所有follower上的分布。

但是仅仅使用上述`(k, m, fragment_id)`的tuple是不够的，因为网络乱序和丢包可能会使得leader无法真正跟踪follower上的数据分布情况。例如，leader连续对一个entry进行了两次encoding, 采用不同的k和m参数。然后第一轮的数据被所有follower replicate并产生了回复，leader对这些回复进行了记录。第二轮的数据被所有follower replicate了，但回复在中途丢失。此时leader会误认为第一轮encoding的fragment在follower上被replicate了，但是实际上follower上存储的是encoding的fragments。leader在这种情况下可能会做出错误的提交行为：例如一共7台机器，第一轮时leader探测到6台机器alive（包括自己），因此它计算得到k=6-3 = 3, m=3；所有fragments都replicate出去；但是第二轮时，它短暂地检测到有7台机器，于是它计算k=7-3=4, m=3.将所有fragments发送出去，但是其中一台机器fail了。所以现在是（k=4, m=3)的配置只在6台机器上存储了，但是leader可能会根据第一轮得到的结果来决定commit。

解决上述问题很简单，我们只需要要求commit时始终按照最近encoding的(k, m)参数来提交即可，因此我们引入了VersionNumber, 来标记每轮发送：

```
struct VersionNumber {
  uint32_t term;
  uint32_t seq;
}
```

VersionNumber由一个term和一个seq组成，term指的是leader在发送该entry时的term，或者leader encoding这个entry时的term。seq是一个单调递增的数字，用来区分每轮发送。

当leader对一个entry进行encoding并且将其replicate到follower时，它生成一个新的VersionNumber, 这一轮发送的所有fragments都按照这个VersionNumber标记。然后被发送出去，同时，leader记下这个VersionNumber以及对应的k，m参数，称为`CommitRequirements`。也就是，必须所有具有这个VersionNumber的fragments被存储在follower上这个entry才能提交。

实际上，只需要一个seq就能区分不同的发送批次了，引入term是为了区分不同的leader。**新的leader会覆盖之前leader发送的fragment**。这样做是为了commit考虑（后面会说）

通过VersionNumber，leader能区分开不同轮的发送；Follower也能避免网络乱序带来的困扰，如果一个VersionNumber更小的entry到达，那么它不会使用这个entry来overwrite。使用VersionNumber，leader也能确切地知道follower上被存储的到底是哪个版本的数据，例如，如果leader通过AppendEntries的回复看到follower的VersionNumber是{term=1, seq=1}, 它查看`CommitRequirements`, 如果这个entry的commitRequirements的VersionNumber也是`{term=1, seq=1}`,那么可以看看能不能提交(还要看k的要求)，这是因为，在看到AppendEntriesRPC回复时，不可能有新的AppendEntries覆盖了对应follower上的这个entry, 否则leader的CommitRequirements会被修改。

综上，我们可以修改Raft的算法：

```
struct VersionNumber {
  uint32_t term;
  uint32_t seq;
};
struct Version {
  VersionNumber v_number;
  int k, m;
  raft_frag_id_t fragment_id;
}
struct LogEntry {
  raft_index_t raft_index;
  raft_term_t raft_term;
  Version version;
  // Data
}
```

Leader进行replicate entries时的操作：

```
void replicate_entries() {
  int k, m = calculate_alive_servers();
  auto version_num = NextVersionNum();
  // Step1: Encoding or re-encoding entries:
  for (idx := commit_index + 1; idx <= last_index; ++idx) {
    if (log[idx]->NotEncoded() || log[idx]->GetK() != k || log[idx]->GetM() != m)
{
      Encode(log[idx]);
    }
    log[idx]->UpdateVersion(version_num, k, m);
```

```
    log[idx]->UpdateCommitRequirements(version_num, k, m);
  }
  // Step2: Construct a map to decide which fragments will each server receive
  auto frag_map = ConstructFragmentToFollowerMap();

  // Step3: For each server, send them corresponding fragments
  for (follower : peers) {
    auto fragment_id = frag_map[follower->Id()];
    AppendEntriesArgs args;
    for (entry in [commit_index + 1, last_index]) {
      args.Add(entry->Fragments(fragment_id));
    }
    rpc->SendAppendEntries(follower->Id(), args);
  }
}
```

Leader会对还没commit的所有entry检查是否需要重新encoding: k和m是否改变。然后更新这些log的version和CommitRequirements.注意，即使不需要re-encoding, 也需要新的version_num, 因为映射关系可能发生了变化。

`ConstructFragmentToFollowerMap`会构建一个map来决定每个follower收到哪个fragment。然后就是将所有fragment发送给follower。

Follower收到AppendEntries的操作：

```
void OnReceiveAppendEntries(AppendEntriesArgs args) {
  for (entry in args) {
    if (entry.version < logs[idx].version) {
      continue;
    }
    if (NeedOverwrite) {
      log[idx]->Overwrite(entry);
    } else {
      log[idx]->UpdateVersion(entry.version);
    }
  }
  reply version;
}
```

这里`NeedOverwrite`只需要检查VersionNumber是否更新，以及`(k, m, fragment_id)`是否与该follower存储的log相同，如果不同则覆盖写，如果相同则只更新version为新的version。

Leader收到AppendEntriesReply的动作：

```
void OnReceiveAppendEntriesReply(AppendEntriesReply reply) {
  for ([index, version] in reply.versions) {
    if (peer.stored_versions[index] < version) {
      peer.store_versions[index] = version;
    }
  }
  MaybeUpdateCommitIndex();
}
```

Leader会对每个follower保存每个index的reply version，并且只保存最高的那个，然后leader检查是否能对某个entry进行提交：

```
void MaybeUpdateCommitIndex() {
  for (index = commit_index + 1; index <= last_index; ++index) {
    int agree_cnt = 1;
    for (peer : peers) {
      if (peer.store_version[index].version_num ==
commit_requirements[index].version_num) {
        agree_cnt += 1;
      }
    }
    if (agree_cnt >= commit_requirements[index].k + F) {
      UpdateCommitIndex(index);
    }
  }
}
```

上面使用了伪代码的方式描述了FlexibleK的算法。

在实现上仍然有一些细节的问题：

- leader在发送entry时是将所有CommitIndex()到LastIndex的entry encoding之后进行发送的，如果一个follower缺少CommitIndex之前的entry怎么办？例如在一个5台机器的cluster中，一个follower fail了，原先的leader正常replicate entry并且成功commit了一些entry，这个时候follower回到集群，那么它会缺少leader的commit index之前的entry。如果按照上面的代码，leader不会给它发送这些entry，这将导致follower的entry并不是连续增长的。
- leader发送fragments时对每个follower都是发送CommitIndex()到LastIndex()之间的entry, 因此它可能忽略了原始Raft中的NextIndex的信息。这可能会导致一些不必要的重发，例如一个fragment已经在follower上被replicate了，但是还没来得及提交，下一轮leader仍然给它发送了这个entry。