FoundationDB project starts in 2009, when there is not a good async programming framework and memory management library. As a result, FDB invents its own, which has many nuances that may trip you if you are not aware.

The following is assuming you are already familiar with flow.

Arena Memory Management

The memory management uses `Arena` abstraction a lot, which owns some memory blocks, typically for `MutationRef`, `VectorRef` these `*Ref` types. For example, a `StringRef` object is usually a pointer and a length to one of the blocks `Arena` owns. Thus, it's very important that the `Arena` is live while accessing these `*Ref` objects.

`Arena` is internal reference counted objects. Thus, if you declare `Arena b = a;`, then both `a` and `b` share the same memory blocks that `a` owns before. Even if `a` goes out of scope, these memory blocks are still valid.

Alternatively, you can have `b.dependsOn(a)`, so that `b` also holds references to the memory blocks `a` owns before. A caveat is that memory allocated by `a` afterward may NOT be reference counted by `b`. This is because the new memory block is only referenced by `a`, not `b`.

If you are curious about the implementation, here is the explanation:

```
// For an arena A, one would expect using B.dependsOn(A) would extent A's lifecycle to B's lifecycle.
// However this is not quite true. Internally arenas are implemented as linked lists, or
//
//  A.impl =      ArenaBlock -> ArenaBlock -> ...
//
// where each ArenaBlock is a piece of allocated memory managed by reference counting. Applying
// B.dependsOn(A) would cause B refers to the first ArenaBlock in A, or
//
//            B----|
//                 V
//  A.impl =      ArenaBlock -> ArenaBlock -> ...
//
// so when A destructs, there is still at least one extra reference to the first ArenaBlock,
// preventing the release of the chained blocks.
// However, when additional memory is requested to A, A will create additional ArenaBlocks (see
// Arena.cpp:ArenaBlock::create), and the new ArenaBlock might be inserted *prior* to the
// existing ArenaBlock, i.e.
//                          B----|
//                               V
//  A.impl =      ArenaBlock -> ArenaBlock -> ArenaBlock ...
//                   ^- new created
// and when A is destructed, the unreferrenced (hereby the first one) will be destructed.
// This causes B depends on only part of A. The only way to ensure all blocks in A have
// the same life cycle to B is to let B.dependsOn(A) get called when A is stable, or no
// extra ArenaBlocks will be created.
```

Serialization with FlatBuffers

FDB implements its own FlatBuffers, originally invented by Google. FlatBuffers allow accessing serialized data without parsing/unpacking.

```
struct GetKeyValuesReply : public LoadBalancedReply {
    constexpr static FileIdentifier file_identifier = 1783066;
    Arena arena;
    VectorRef<KeyValueRef, VecSerStrategy::String> data;
    Version version; // useful when latestVersion was requested
    bool more;
    bool cached = false;

   GetKeyValuesReply() : version(invalidVersion), more(false), cached(false) {}

    template <class Ar>
    void serialize(Ar& ar) {
     serializer(ar, LoadBalancedReply::penalty, LoadBalancedReply::error, data, version, more, cached, arena
    }
};
```

The above is an example of using FlatBuffers. The function `serialize()`, surprisingly, does both serialization and deserialization! This is because:

```
template <class Archive, class Item, class... Items>
typename Archive::WRITER& serializer(Archive& ar, const Item& item, const Items&... items) {
    save(ar, item);
    if constexpr (sizeof...(Items) > 0) {
        serializer(ar, items...);
    }
    return ar;
}

template <class Archive, class Item, class... Items>
typename Archive::READER& serializer(Archive& ar, Item& item, Items&... items) {
    load(ar, item);
    if constexpr (sizeof...(Items) > 0) {
        serializer(ar, items...);
    }
    return ar;
}
```

The "Reader" (e.g., `BinaryReader`, `ObjectReader`, `ArenaReader`) calls `load(ar, item)` to deserialize all items, while the "Writer" (e.g., `BinaryWriter`, `ObjectWriter`) calls `save(ar, item)` to serialize all data items.

Sometimes, we want the serialization and deserialization to behave differently according to the value of data items. For instance, a single key clear `MutationRef` can be compressed by sending only the key once, not twice (`key` and `keyAfter(key)`). To achieve this, the `serialize()` function below checks if we are serializing `ar.isSerializing` or deserializing `ar.isDeserializing`:

```
struct MutationRef {
    uint8_t type;
    StringRef param1, param2;

    template <class Ar>
    void serialize(Ar& ar) {
     if (ar.isSerializing && type == ClearRange && equalsKeyAfter(param1, param2)) {
            StringRef empty;
            serializer(ar, type, param2, empty);
        } else {
            serializer(ar, type, param1, param2);
        }
     if (ar.isDeserializing && type == ClearRange && param2 == StringRef() && param1 != StringRef()) {
            ASSERT(param1[param1.size() - 1] == '\x00');
            param2 = param1;
            param1 = param2.substr(0, param2.size() - 1);
        }
    }
```

Append new fields in `serializer()`

FlatBuffer has a requirement that new fields should be added to the end of the list. During the 7.1.4 release process, we encountered an interesting bug: suddenly the Python or Java clients can no longer issues API calls to the 7.1.2 cluster. On the other hand, the same client that uses the 7.1.4 `libfdb_c.so` can work perfectly fine with the 7.1.4 cluster. After several hours, a group of engineers noticed there were incompatible changes:

```
struct OpenDatabaseCoordRequest {
...
        void serialize(Ar& ar) {
-          serializer(ar, issues, supportedVersions, traceLogGroup, knownClientInfoID, clusterKey, coordina
+               serializer(ar,
+                          issues,
+                          supportedVersions,
+                          traceLogGroup,
+                          knownClientInfoID,
+                          clusterKey,
+                          hostnames,
+                              coordinators,
+                              reply);
        }
```

Note `hostnames` is inserted into the middle of the list of items. This change breaks compatibility between the client and the server, because the 7.1.2 server would interpret `hostnames` as `coordinators`—flat buffer use the relative position of items for serialization. The fix is to move `hostnames` to the end of the list.

A side note is that within a major version, such as 6.3 or 7.1, FDB binaries should be compatible. More specifically using FDB's term, they should be protocol compatible. In this way, a client uses 6.3.9 C library can connect to 6.3.24 cluster without any issues. And the server cluster can upgrade to new minor release versions, without the upgrade requirement for the client C library.

BinaryReader has an `Arena`

A surprising fact is that `BinaryReader` has an internal `Arena` object, which is used in `BinaryReader::arenaRead()`:

```
    const uint8_t* arenaRead(int bytes) {
        // Reads and returns the next bytes.
        // The returned pointer has the lifetime of this.arena()
      // Could be implemented zero-copy if [begin,end) was in this.arena() already; for now is a copy
        if (!bytes)
            return nullptr;
        uint8_t* dat = new (arena()) uint8_t[bytes];
        serializeBytes(dat, bytes);
        return dat;
    }
```

Note `arenaRead()` allocates memory `new (arena()) uint8_t[bytes]`, and then copies bytes into the newly allocated memory. This is important to know, because for `StringRef`, its serialization (defined in Arena.h) is:

```
template <class Archive>
inline void load(Archive& ar, StringRef& value) {
    uint32_t length;
    ar >> length;
    value = StringRef(ar.arenaRead(length), length);
}
```

So if a `BinaryReader` is used to deserialize a `StringRef`, then the returned value's life cycle is the same as the `BinaryReader` object. As a result, if the `BinaryReader` object goes out of the scope, the `MutationRef` returned by the `BinaryReader` object will points to invalid memory:

```
StringRef serialized = some_valid_string_ref;
```

```
MutationRef m;
  {
    BinaryReader reader(serialized, IncludeVersion());
    reader >> m;
  }
// m.param1 now points to invalid memory
```

`AsyncVar::set()` may not "set" the value

The `AsyncVar::set()` compares the value with its internal state and only set if they are different:

```
    void set(V const& v) {
        if (v ≠ value)
            setUnconditional(v);
    }
```

This usually works. However, if the ≠ operator for the value `v` is overloaded, there could be unintended consequences. For example, the `ClientDBInfo` has the overloaded operator as:

```
bool operator≠(ClientDBInfo const& r) const { return id ≠ r.id; }
```

And in `extractClientInfo()`, we had code like:

```
        ClientDBInfo ni = db->get().client;
     shrinkProxyList(ni, lastCommitProxyUIDs, lastCommitProxies, lastGrvProxyUIDs, lastGrvProxies);
        info->set(ni);
```

This looks correct, but is wrong! The reason is that `shrinkProxyList()` modifies the `ni` object, but does not change the `id` field. As a result, even if `ni` has changed, `info->set(ni)` has no effect! This bug has caused infinite retrying GRVs at the client side. Specifically, the version vector feature introduces a change that the client side compares the returned proxy ID with its known set of GRV proxies and will retry GRV if the returned proxy ID is not in the set. Due to the above bug, GRV returned by a proxy is not within the client set, because the change was not applied to the "info". The fix is in PR #6877.

`addr2line` can be inaccurate

In the trace log, we often see backtrace like: `addr2line -e fdbserver.debug -p -C -f -i 0x7fae93006630 0x26b5f98 0x26eff9b 0x26dc132`. There are two things to keep in mind: * The backtrace can be inaccurate. For instance, the backtrace of a segmentation fault is not obvious. When this happens, use `gdb` to rerun the seed and the backtrace within `gdb` is more reliable. * The `fdbserver.debug` refers to the symbol file stripped from the binary. This `fdbserver.debug` file can be replaced with the unstripped `fdbserver` binary.

Debugging Techniques

- See this doc for general techniques and samples
- https://github.com/apple/foundationdb/wiki/Debug-Out-Of-Memory-(OOM)-Errors-in-Simulation
- https://github.com/apple/foundationdb/wiki/Debug-Performance-Issues-Using-Perf
- https://github.com/apple/foundationdb/wiki/Debugging-in-FDB
- https://github.com/apple/foundationdb/wiki/How-to-reproduce-a-restart-test-failure
- Use Valgrind/ASAN for FDB.

"Mostly" Single Threaded

The `fdbserver` runs almost everything in a single "network" thread (sometimes called "main" thread in the code). Particularly, all "flow" related stuff, i.e., actors, run on the network thread.

There are other threads for not so important stuff like profiling, printing stack traces, etc., if you are curious.

TODOs: flow lock, actor's synchronous callback invocation