The global configuration framework is an eventually consistent configuration mechanism to efficiently make runtime changes to all clients and servers. It works by broadcasting updates made to the global configuration key space, relying on individual machines to store existing configuration in-memory.

The global configuration framework provides a key-value interface to all processes in a FoundationDB cluster. The first part of this document describes how to use the global configuration framework to read and write data. The second part goes into implementation details.

Using the global configuration framework

Reading data

The global configuration framework is exposed through the `GlobalConfig::globalConfig()` static function. The API is defined in `fdbclient/GlobalConfig.actor.h`. There are separate ways to read a value, depending on if it is an object or a primitive.

Reading a primitive The recommended way to read a primitive from the global configuration is to use the templated get function (note that this requires you to know the type of the value). A default value must be supplied in case the global configuration framework cannot find the specified key.

```
auto& config = GlobalConfig::globalConfig();
double value = config.get<double>("path/to/key", 1.0);
```

Instead of hardcoding a string literal for the key, it is recommended to store your global configuration keys in the global configuration implementation file (at the top).

Reading an object Reading an object returns both the object as well as a reference to the arena where the objects memory lives. As long as the client keeps the arena around, the object's memory is guaranteed to stick around.

```
auto& config = GlobalConfig::globalConfig();
auto configValue = config.get("path/to/key");
```

The type returned is `ConfigValue`, defined as

```
struct ConfigValue {
    Arena arena;
    std::any value;
};
```

If the global configuration framework does not have a value for the given key, `configValue.value.has_value()` will return false. Otherwise, the value can be cast to the appropriate type with `std::any_cast`.

```
ASSERT(configValue.value.has_value());
auto str = std::any_cast<StringRef>(configValue.value);
```

Range read Range reads of global configuration data are also available. See `get(KeyRangeRef range)` in the global configuration header file.

Writing data

Writing data is done using transactions written to the special key space range `\xff\xff/global_config/ - \xff\xff/global_config/0`. Data must always be encoded as tuples. See `fdbclient/Tuple.h` for the encode/decode API.

You are responsible for avoiding conflicts with other global configuration keys. For most uses, it is recommended to create a new directory. For example, an application that wants to write configuration data should use the `global_config/config/` namespace, instead of storing keys in the top level. Global configuration keys should be stored as constants in `flow/GlobalConfig.actor.cpp`, and you can check this file to avoid naming conflicts.

Here is an example of writing to the global configuration.

```
// In GlobalConfig.actor.h
extern const KeyRef myGlobalConfigKey;
// In GlobalConfig.actor.cpp
const KeyRef myGlobalConfigKey = LiteralStringRef("config/key");

// When you want to set the value..
Tuple value = Tuple().appendDouble(1.5);

FDBTransaction* tr = ...;
tr->setOption(FDBTransactionOptions::SPECIAL_KEY_SPACE_ENABLE_WRITES);
tr->set(GlobalConfig::prefixedKey(myGlobalConfigKey), value.pack());
// commit transaction
```

Clearing data Clears and clear ranges are supported and behave in the same way as normal transaction clears.

Implementation details

The global configuration framework stores key-value pairs using FDB itself, with some abstraction between storage and retrieval. Durable configuration data lives in the `\xff/globalConfig/` key space and is maintained in the same way as all other key-value pairs in FDB.

Storage Internals

The `\xff/globalConfig/` key space is broken up into three ranges.

- `\xff/globalConfig/ - \xff/globalConfig/k0`: storage of the key-value pairs that make up the global configuration
- `\xff/globalConfig/h/ - \xff/globalConfig/h0`: a truncated list of the last three updates made to the global configuration. Keys in this range represent the commit version the update was made at, and the value is a serialized string of `MutationRef`s representing the update
- `\xff/globalConfig/v`: serialized versionstamp of the last update made to the global configuration, followed by the protocol version

Updates

The cluster controller creates a watch on `\xff/globalConfig/v` and updates its `AsyncVar<ClientDBInfo>` with the most recent mutations when the version changes. `ClientDBInfo` is broadcast to all nodes in the system on any change, and eventually each individual node will receive `ClientDBInfo` containing the three most recent updates made to the global configuration. If the node does not know about the oldest mutation in the mutation list, it must re-read the entire global configuration from the `\xff/globalConfig/k/` range. Otherwise, it can use the history in `ClientDBInfo` to update its global configuration. This is the mechanism through which all nodes eventually converge on the most recent configuration.

Special Key Space

The special key space performs extra work when writing to the global configuration. The following actions are performed when a transaction making at least one write to `\xff\xff/global_config/` is committed.

1. Reads the `\xff/globalConfig/h/` range and deletes the oldest key-value pairs (according to version) to make room for the new update.
2. Transforms writes made to `\xff\xff/global_config/` into writes made to `\xff/globalConfig/k/` for durable storage.
3. Writes the newest set of mutations into a new `\xff/globalConfig/h/<version>` key, where `<version>` is the versionstamp of the commit.
4. Updates `\xff/globalConfig/v` with the versionstamp and protocol version used for the transaction.

The special key space also provides functionality for reading global configuration values. A transaction may read `\xff\xff/global_config/<your-key>` to read values set in the global configuration namespace. This is for information and debugging purposes only, and should not be used by client applications (see Reading Data above). The special key space read functionality will return all values as strings.