```cpp
// PMLL.cpp

#include <iostream>
#include <fstream>
#include <unordered_map>
#include <mutex>
#include <thread>
#include <future>
#include <chrono>
#include <string>
#include <vector>
#include <list>
#include <memory>

// Include nlohmann/json for JSON serialization
#include <nlohmann/json.hpp>

// Include spdlog for logging
#include <spdlog/spdlog.h>
#include <spdlog/sinks/basic_file_sink.h>

// Namespace declarations for convenience
using json = nlohmann::json;

// ------------------------------
// LRU Cache Implementation
// ------------------------------
template<typename Key, typename Value>
class LRUCache {
public:
    LRUCache(size_t capacity) : capacity_(capacity) {}

    bool get(const Key& key, Value& value) {
        std::lock_guard<std::mutex> lock(cache_mutex_);
        auto it = cache_map_.find(key);
        if (it == cache_map_.end()) {
            return false;
        }
        // Move the accessed item to the front of the list
        cache_list_.splice(cache_list_.begin(), cache_list_, it->second);
        value = it->second->second;
        return true;
    }

    void put(const Key& key, const Value& value) {
        std::lock_guard<std::mutex> lock(cache_mutex_);
        auto it = cache_map_.find(key);
        if (it != cache_map_.end()) {
            // Update existing item and move to front
            it->second->second = value;
            cache_list_.splice(cache_list_.begin(), cache_list_, it->second);
            return;
        }

        // Insert new item at the front
        cache_list_.emplace_front(key, value);
        cache_map_[key] = cache_list_.begin();
        // Evict least recently used item if capacity is exceeded
        if (cache_map_.size() > capacity_) {
            auto last = cache_list_.end();
            last--;
            cache_map_.erase(last->first);
            cache_list_.pop_back();
        }
    }

    size_t capacity() const {
        return capacity_;
    }

private:
    size_t capacity_;
    std::list<std::pair<Key, Value>> cache_list_;
    std::unordered_map<Key, typename std::list<std::pair<Key, Value>>::iterator> cache_map_;
    std::mutex cache_mutex_;
};

// ----------------------------
// PersistentMemory Class
// ----------------------------
class PersistentMemory {
public:
    PersistentMemory(const std::string& memory_file = "persistent_memory.json",
                    size_t cache_capacity = 1000)
        : memory_file_(memory_file),
          cache_(cache_capacity),

logger_(spdlog::basic_logger_mt(
"PMLL_logger", "pml_log.txt")) {
        logger_-
>set_level(spdlog::level::info);
        loadMemory();
    }

    ~PersistentMemory() {
        // Optional: Flush logs before
destruction

spdlog::drop("PMLL_logger");
    }

    // Add or update memory
asynchronously
    void addMemory(const
std::string& key, const json&
value) {
        {

std::lock_guard<std::mutex>
lock(memory_mutex_);
            memory_data_[key] =
value;
            cache_.put(key, value);
            logger_->info("Added/
Updated memory for key: {}", key);
        }
        saveMemoryAsync();
    }

    // Retrieve memory with caching
    json getMemory(const
std::string& key, const json&
default_value = nullptr) {
        json value;
        if (cache_.get(key, value)) {
            logger_->info("Cache hit for key: {}", key);
            return value;
        }

        {
            std::lock_guard<std::mutex> lock(memory_mutex_);
            auto it = memory_data_.find(key);
            if (it != memory_data_.end()) {
                cache_.put(key, it->second);
                logger_->info("Cache miss for key: {}. Loaded from storage.", key);
                return it->second;
            }
        }
        logger_->warn("Key not found: {}", key);
        return default_value;
    }

    // Clear all memory asynchronously
    void clearMemory() {
        {
            std::lock_guard<std::mutex> lock(memory_mutex_);
            memory_data_.clear();
            cache_ = LRUCache<std::string, json>(cache_.capacity()); // Reset cache
            logger_->info("Cleared all
memory.");
        }
        saveMemoryAsync();
    }

    // Add a memory version
    void addMemoryVersion(const
std::string& key, const json&
value) {
        {

std::lock_guard<std::mutex>
lock(memory_mutex_);

memory_versions_[key].push_bac
k(value);
            memory_data_[key] =
value;
            logger_->info("Added
memory version for key: {}", key);
        }
        saveMemoryAsync();
    }

    // Retrieve a specific memory
version
    json getMemoryVersion(const
std::string& key, size_t version) {
        std::lock_guard<std::mutex>
lock(memory_mutex_);
        if
(memory_versions_.find(key) !=
memory_versions_.end() &&
            version <
memory_versions_[key].size()) {
            logger_->info("Retrieved
version {} for key: {}", version,
key);
```

```
            return
memory_versions_[key][version];
        }
        logger_->warn("Version {} for
key: {} not found.", version, key);
        return nullptr;
    }

private:
    std::string memory_file_;

std::unordered_map<std::string,
json> memory_data_;
    LRUCache<std::string, json>
cache_;
    std::mutex memory_mutex_;

std::shared_ptr<spdlog::logger>
logger_;

std::unordered_map<std::string,
std::vector<json>>
memory_versions_;

    // Load memory from file
    void loadMemory() {
        std::ifstream
infile(memory_file_);
        if (infile.is_open()) {
            try {
                json j;
                infile >> j;

std::lock_guard<std::mutex>
lock(memory_mutex_);
                if
(j.contains("memory_data")) {
                    memory_data_ =
j.at("memory_data").get<std::un-
ordered_map<std::string,
json>>();
                }
                if
(j.contains("memory_versions")) {
                    memory_versions_ =
j.at("memory_versions").get<std::
unordered_map<std::string,
std::vector<json>>>();
                }
                logger_->info("Memory
loaded from file: {}",
memory_file_);
            } catch (const
json::parse_error& e) {
                logger_->error("Error
parsing memory file: {}", e.what());
            } catch (const
json::out_of_range& e) {
                logger_->error("Missing
keys in memory file: {}", e.what());
            }
            infile.close();
        } else {
            logger_->warn("Memory
file not found. Starting with empty
memory.");
        }
    }

    // Save memory to file
    void saveMemory() {
        std::lock_guard<std::mutex>
lock(memory_mutex_);
        std::ofstream
outfile(memory_file_);
        if (outfile.is_open()) {
            try {
                json j;
                j["memory_data"] =
memory_data_;
                j["memory_versions"] =
memory_versions_;
                outfile << j.dump(4);
                logger_->info("Memory
saved to file: {}", memory_file_);
            } catch (const
json::type_error& e) {
                logger_->error("Error
serializing memory data: {}",
e.what());
            }
            outfile.close();
        } else {
            logger_->error("Failed to
open memory file for writing: {}",
memory_file_);
        }
    }

    // Asynchronous save operation
    void saveMemoryAsync() {
        std::future<void> fut =
std::async(std::launch::async,
&PersistentMemory::saveMemory,
this);
        // Optionally, store futures if
you need to manage their lifetimes
    }
};

// ----------------------------
// Example Usage
// ----------------------------
int main() {
    // Initialize PersistentMemory
with default file and cache
capacity
```

```
    PersistentMemory pm;

    // Add memory entries
    pm.addMemory("user_123",
{ {"name", "Josef"}, {"last_topic",
"PMLL architecture"} });
    pm.addMemory("user_456",
{ {"name", "Alice"}, {"last_topic",
"Distributed Systems"} });

    // Retrieve memory entries
    json user1 =
pm.getMemory("user_123");
    if (!user1.is_null()) {
        std::cout << "Welcome back,
" << user1["name"]
                  << "! Last time we
discussed " << user1["last_topic"]
<< "." << std::endl;
    } else {
        std::cout << "Welcome! Let's
start our conversation." <<
std::endl;
    }

    json user2 =
pm.getMemory("user_456");
    if (!user2.is_null()) {
        std::cout << "Hello, " <<
user2["name"]
                  << "! Last time we talked
about " << user2["last_topic"] <<
"." << std::endl;
    } else {
        std::cout << "Hello! Let's
begin our interaction." <<
std::endl;
    }
    // Update a memory entry
    pm.addMemory("user_123", { {"name", "Josef"}, {"last_topic", "GPT-5 ETA"} });

    // Add a new version to a memory entry
    pm.addMemoryVersion("user_123", { {"name", "Josef"}, {"last_topic", "Advanced PMLL Features"} });

    // Retrieve a specific memory version
    json version1 = pm.getMemoryVersion("user_123", 0); // First version
    if (!version1.is_null()) {
        std::cout << "Version 1 for user_123: " << version1.dump() << std::endl;
    } else {
        std::cout << "No version 1 found for user_123." << std::endl;
    }

    // Attempt to retrieve a non-existent key
    json user3 = pm.getMemory("user_789");
    if (user3.is_null()) {
        std::cout << "No records found for user_789." << std::endl;
    }

    // Simulate some delay to allow asynchronous save operations to complete
```

```
std::this_thread::sleep_for(std::chrono::seconds(2));

    // Clear all memory
    pm.clearMemory();

    return 0;
}

// -----------------------------
// src/PersistentMemory.cpp
// -----------------------------

#include "PersistentMemory.h"
#include <fstream>
#include <thread>
#include <future>

// LRUCache Implementation
template<typename Key, typename Value>
LRUCache<Key, Value>::LRUCache(size_t capacity) : capacity_(capacity) {}

template<typename Key, typename Value>
bool LRUCache<Key, Value>::get(const Key& key, Value& value) {
    std::lock_guard<std::mutex> lock(cache_mutex_);
    auto it = cache_map_.find(key);
    if (it == cache_map_.end()) {
        return false;
    }
    // Move the accessed item to the front of the list
```

```
cache_list_.splice(cache_list_.begin(), cache_list_, it->second);
    value = it->second->second;
    return true;
}

template<typename Key,
typename Value>
void LRUCache<Key,
Value>::put(const Key& key, const
Value& value) {
    std::lock_guard<std::mutex>
lock(cache_mutex_);
    auto it = cache_map_.find(key);
    if (it != cache_map_.end()) {
        // Update existing item and
move to front
        it->second->second = value;

cache_list_.splice(cache_list_.begin(), cache_list_, it->second);
        return;
    }

    // Insert new item at the front
    cache_list_.emplace_front(key,
value);
    cache_map_[key] =
cache_list_.begin();

    // Evict least recently used item
if capacity is exceeded
    if (cache_map_.size() >
capacity_) {
        auto last = cache_list_.end();
        last--;
        cache_map_.erase(last-
>first);
```

```
        cache_list_.pop_back();
    }
}

// Explicit template instantiation to
avoid linker errors
template class
LRUCache<std::string, json>;

// PersistentMemory
Implementation
PersistentMemory::Persistent-
Memory(const std::string&
memory_file, size_t
cache_capacity)
    : memory_file_(memory_file),
      cache_(cache_capacity),

logger_(spdlog::basic_logger_mt(
"PMLL_logger", "pml_log.txt")) {
    logger_-
>set_level(spdlog::level::info);
    loadMemory();
}

PersistentMemory::~Persistent-
Memory() {
    // Optional: Flush logs before
destruction
    spdlog::drop("PMLL_logger");
}

void
PersistentMemory::addMemory(c
onst std::string& key, const json&
value) {
    {
        std::lock_guard<std::mutex>
lock(memory_mutex_);
```

```
        memory_data_[key] = value;
        cache_.put(key, value);
        logger_->info("Added/Updated memory for key: {}", key);
    }
    saveMemoryAsync();
}

json PersistentMemory::getMemory(const std::string& key, const json& default_value) {
    json value;
    if (cache_.get(key, value)) {
        logger_->info("Cache hit for key: {}", key);
        return value;
    }

    {
        std::lock_guard<std::mutex> lock(memory_mutex_);
        auto it = memory_data_.find(key);
        if (it != memory_data_.end()) {
            cache_.put(key, it->second);
            logger_->info("Cache miss for key: {}. Loaded from storage.", key);
            return it->second;
        }
    }
    logger_->warn("Key not found: {}", key);
    return default_value;
}

void
```

```
PersistentMemory::clearMemory() {
    {
        std::lock_guard<std::mutex> lock(memory_mutex_);
        memory_data_.clear();
        cache_ = LRUCache<std::string, json>(cache_.capacity()); // Reset cache
        logger_->info("Cleared all memory.");
    }
    saveMemoryAsync();
}

void PersistentMemory::addMemoryVersion(const std::string& key, const json& value) {
    {
        std::lock_guard<std::mutex> lock(memory_mutex_);
        memory_versions_[key].push_back(value);
        memory_data_[key] = value;
        logger_->info("Added memory version for key: {}", key);
    }
    saveMemoryAsync();
}

json PersistentMemory::getMemoryVersion(const std::string& key, size_t version) {
    std::lock_guard<std::mutex> lock(memory_mutex_);
    if (memory_versions_.find(key) !
= memory_versions_.end() &&
        version <
memory_versions_[key].size()) {
        logger_->info("Retrieved
version {} for key: {}", version,
key);
        return
memory_versions_[key][version];
    }
    logger_->warn("Version {} for
key: {} not found.", version, key);
    return nullptr;
}

void
PersistentMemory::loadMemory()
{
    std::ifstream
infile(memory_file_);
    if (infile.is_open()) {
        try {
            json j;
            infile >> j;

std::lock_guard<std::mutex>
lock(memory_mutex_);
            if
(j.contains("memory_data")) {
                memory_data_ =
j.at("memory_data").get<std::unordered_map<std::string,
json>>();
            }
            if
(j.contains("memory_versions")) {
                memory_versions_ =
j.at("memory_versions").get<std::
unordered_map<std::string,
std::vector<json>>>();
            }
            logger_->info("Memory loaded from file: {}", memory_file_);
        } catch (const json::parse_error& e) {
            logger_->error("Error parsing memory file: {}", e.what());
        } catch (const json::out_of_range& e) {
            logger_->error("Missing keys in memory file: {}", e.what());
        }
        infile.close();
    } else {
        logger_->warn("Memory file not found. Starting with empty memory.");
    }
}

void PersistentMemory::saveMemory() {
    std::lock_guard<std::mutex> lock(memory_mutex_);
    std::ofstream outfile(memory_file_);
    if (outfile.is_open()) {
        try {
            json j;
            j["memory_data"] = memory_data_;
            j["memory_versions"] = memory_versions_;
            outfile << j.dump(4);
            logger_->info("Memory saved to file: {}", memory_file_);
        } catch (const json::type_error& e) {
            logger_->error("Error serializing memory data: {}", e.what());
        }
        outfile.close();
    } else {
        logger_->error("Failed to open memory file for writing: {}", memory_file_);
    }
}

void PersistentMemory::saveMemoryAsync() {
    std::future<void> fut = std::async(std::launch::async, &PersistentMemory::saveMemory, this);
    // Optionally, store futures if you need to manage their lifetimes
} // x/ibc/memory/module.go

package memory

import (
    "github.com/cosmos/cosmos-sdk/codec"
    sdk "github.com/cosmos/cosmos-sdk/types"
    "github.com/cosmos/cosmos-sdk/types/module"
)

type AppModule struct {
    AppModuleBasic
    keeper Keeper
```

```
}

func NewAppModule(keeper
Keeper) AppModule {
    return AppModule{
        AppModuleBasic:
AppModuleBasic{},
        keeper:        keeper,
    }
}

func (AppModule) Name() string {
    return "memory"
}

func (AppModule)
RegisterInvariants(_
sdk.InvariantRegistry) {}

func (AppModule) Route()
sdk.Route {
    // Define message routing if
needed
    return sdk.Route{}
}

func (AppModule) QuerierRoute()
string {
    return "memory"
}

func (AppModule)
LegacyQuerierHandler(*codec.-
Codec) sdk.Querier {
    // Implement querier if needed
    return nil
}

func (AppModule) BeginBlock(_
sdk.Context, _
abci.RequestBeginBlock) {}

func (AppModule) EndBlock(_
sdk.Context, _
abci.RequestEndBlock)
[]abci.ValidatorUpdate {
   return []abci.ValidatorUpdate{}
}

// AppModuleBasic implements the
basic application module
type AppModuleBasic struct{}

func (AppModuleBasic) Name()
string {
   return "memory"
}

func (AppModuleBasic)
RegisterCodec(*codec.Codec) {}

func (AppModuleBasic)
DefaultGenesis()
json.RawMessage {
   return nil
}

func (AppModuleBasic)
ValidateGenesis(json.RawMessage)
error {
   return nil
}

func (AppModuleBasic)
RegisterRESTRoutes(ctx
client.Context, r *mux.Router) {}

func (AppModuleBasic)
```

```
RegisterGRPCGatewayRoutes(clientCtx client.Context, mux *runtime.ServeMux) {}

func (AppModuleBasic) GetTxCmd() *cobra.Command {
    return nil
}

func (AppModuleBasic) GetQueryCmd() *cobra.Command {
    return nil
}

// x/ibc/memory/module.go

package memory

import (
    "github.com/cosmos/cosmos-sdk/codec"
    sdk "github.com/cosmos/cosmos-sdk/types"
    "github.com/cosmos/cosmos-sdk/types/module"
)

type AppModule struct {
    AppModuleBasic
    keeper Keeper
}

func NewAppModule(keeper Keeper) AppModule {
    return AppModule{
        AppModuleBasic: AppModuleBasic{},
        keeper:       keeper,
    }
}

func (AppModule) Name() string {
    return "memory"
}

func (AppModule) RegisterInvariants(_ sdk.InvariantRegistry) {}

func (AppModule) Route() sdk.Route {
    // Define message routing if needed
    return sdk.Route{}
}

func (AppModule) QuerierRoute() string {
    return "memory"
}

func (AppModule) LegacyQuerierHandler(*codec.Codec) sdk.Querier {
    // Implement querier if needed
    return nil
}

func (AppModule) BeginBlock(_ sdk.Context, _ abci.RequestBeginBlock) {}

func (AppModule) EndBlock(_ sdk.Context, _ abci.RequestEndBlock) []abci.ValidatorUpdate {
    return []abci.ValidatorUpdate{}
```

```
}

// AppModuleBasic implements the basic application module
type AppModuleBasic struct{}

func (AppModuleBasic) Name() string {
   return "memory"
}

func (AppModuleBasic) RegisterCodec(*codec.Codec) {}

func (AppModuleBasic) DefaultGenesis() json.RawMessage {
   return nil
}

func (AppModuleBasic) ValidateGenesis(json.RawMessage) error {
   return nil
}

func (AppModuleBasic) RegisterRESTRoutes(ctx client.Context, r *mux.Router) {}

func (AppModuleBasic) RegisterGRPCGatewayRoutes(clientCtx client.Context, mux *runtime.ServeMux) {}

func (AppModuleBasic) GetTxCmd() *cobra.Command {
   return nil
}
```

```
func (AppModuleBasic) GetQueryCmd() *cobra.Command {
    return nil // x/ibc/memory/module.go

package memory

import (
    "github.com/cosmos/cosmos-sdk/codec"
    sdk "github.com/cosmos/cosmos-sdk/types"
    "github.com/cosmos/cosmos-sdk/types/module"
)

type AppModule struct {
    AppModuleBasic
    keeper Keeper
}

func NewAppModule(keeper Keeper) AppModule {
    return AppModule{
        AppModuleBasic: AppModuleBasic{},
        keeper:        keeper,
    }
}

func (AppModule) Name() string {
    return "memory"
}

func (AppModule) RegisterInvariants(_ sdk.InvariantRegistry) {}
func (AppModule) Route() sdk.Route {
    // Define message routing if needed
    return sdk.Route{}
}

func (AppModule) QuerierRoute() string {
    return "memory"
}

func (AppModule) LegacyQuerierHandler(*codec.Codec) sdk.Querier {
    // Implement querier if needed
    return nil
}

func (AppModule) BeginBlock(_ sdk.Context, _ abci.RequestBeginBlock) {}

func (AppModule) EndBlock(_ sdk.Context, _ abci.RequestEndBlock) []abci.ValidatorUpdate {
    return []abci.ValidatorUpdate{}
}

// AppModuleBasic implements the basic application module
type AppModuleBasic struct{}

func (AppModuleBasic) Name() string {
    return "memory"
}
```

```
func (AppModuleBasic) RegisterCodec(*codec.Codec) {}

func (AppModuleBasic) DefaultGenesis() json.RawMessage {
    return nil
}

func (AppModuleBasic) ValidateGenesis(json.RawMessage) error {
    return nil
}

func (AppModuleBasic) RegisterRESTRoutes(ctx client.Context, r *mux.Router) {}

func (AppModuleBasic) RegisterGRPCGatewayRoutes(clientCtx client.Context, mux *runtime.ServeMux) {}

func (AppModuleBasic) GetTxCmd() *cobra.Command {
    return nil
}

func (AppModuleBasic) GetQueryCmd() *cobra.Command {
    return nil
}

// x/ibc/memory/module.go

package memory
```

```go
import (
    "github.com/cosmos/cosmos-sdk/codec"
    sdk "github.com/cosmos/cosmos-sdk/types"
    "github.com/cosmos/cosmos-sdk/types/module"
)

type AppModule struct {
    AppModuleBasic
    keeper Keeper
}

func NewAppModule(keeper Keeper) AppModule {
    return AppModule{
        AppModuleBasic: AppModuleBasic{},
        keeper:         keeper,
    }
}

func (AppModule) Name() string {
    return "memory"
}

func (AppModule) RegisterInvariants(_ sdk.InvariantRegistry) {}

func (AppModule) Route() sdk.Route {
    // Define message routing if needed
    return sdk.Route{}
}
```

```
func (AppModule) QuerierRoute() string {
    return "memory"
}

func (AppModule) LegacyQuerierHandler(*codec.Codec) sdk.Querier {
    // Implement querier if needed
    return nil
}

func (AppModule) BeginBlock(_ sdk.Context, _ abci.RequestBeginBlock) {}

func (AppModule) EndBlock(_ sdk.Context, _ abci.RequestEndBlock) []abci.ValidatorUpdate {
    return []abci.ValidatorUpdate{}
}

// AppModuleBasic implements the basic application module
type AppModuleBasic struct{}

func (AppModuleBasic) Name() string {
    return "memory"
}

func (AppModuleBasic) RegisterCodec(*codec.Codec) {}

func (AppModuleBasic) DefaultGenesis() json.RawMessage {
    return nil
```

```
}

func (AppModuleBasic) ValidateGenesis(json.RawMessage) error {
	return nil
}

func (AppModuleBasic) RegisterRESTRoutes(ctx client.Context, r *mux.Router) {}

func (AppModuleBasic) RegisterGRPCGatewayRoutes(clientCtx client.Context, mux *runtime.ServeMux) {}

func (AppModuleBasic) GetTxCmd() *cobra.Command {
	return nil
}

func (AppModuleBasic) GetQueryCmd() *cobra.Command {
	return nil
}

// x/ibc/memory/module.go

package memory

import (
	"github.com/cosmos/cosmos-sdk/codec"
	sdk "github.com/cosmos/cosmos-sdk/types"
	"github.com/cosmos/cosmos-sdk/types/module"
```

```
)

type AppModule struct {
    AppModuleBasic
    keeper Keeper
}

func NewAppModule(keeper Keeper) AppModule {
    return AppModule{
        AppModuleBasic: AppModuleBasic{},
        keeper:         keeper,
    }
}

func (AppModule) Name() string {
    return "memory"
}

func (AppModule) RegisterInvariants(_ sdk.InvariantRegistry) {}

func (AppModule) Route() sdk.Route {
    // Define message routing if needed
    return sdk.Route{}
}

func (AppModule) QuerierRoute() string {
    return "memory"
}

func (AppModule) LegacyQuerierHandler(*codec.Codec) sdk.Querier {
    // Implement querier if needed
    return nil
}

func (AppModule) BeginBlock(_ sdk.Context, _ abci.RequestBeginBlock) {}

func (AppModule) EndBlock(_ sdk.Context, _ abci.RequestEndBlock) []abci.ValidatorUpdate {
    return []abci.ValidatorUpdate{}
}

// AppModuleBasic implements the basic application module
type AppModuleBasic struct{}

func (AppModuleBasic) Name() string {
    return "memory"
}

func (AppModuleBasic) RegisterCodec(*codec.Codec) {}

func (AppModuleBasic) DefaultGenesis() json.RawMessage {
    return nil
}

func (AppModuleBasic) ValidateGenesis(json.RawMessage) error {
    return nil
}
```

```
func (AppModuleBasic) RegisterRESTRoutes(ctx client.Context, r *mux.Router) {}

func (AppModuleBasic) RegisterGRPCGatewayRoutes(clientCtx client.Context, mux *runtime.ServeMux) {}

func (AppModuleBasic) GetTxCmd() *cobra.Command {
    return nil
}

func (AppModuleBasic) GetQueryCmd() *cobra.Command {
    return nil
}
```

```
// cosmossdkibc.cpp

#include <iostream>
#include <fstream>
#include <unordered_map>
#include <mutex>
#include <thread>
#include <future>
#include <chrono>
#include <string>
#include <vector>
#include <list>
#include <memory>

// Include nlohmann/json for JSON serialization
#include <nlohmann/json.hpp>

// Include spdlog for logging
#include <spdlog/spdlog.h>
```

```
#include <spdlog/sinks/
basic_file_sink.h>

// Include cURL for HTTP requests
#include <curl/curl.h>

// Namespace declarations for
convenience
using json = nlohmann::json;

// -----------------------------
// LRU Cache Implementation
// -----------------------------
template<typename Key,
typename Value>
class LRUCache {
public:
    LRUCache(size_t capacity) :
capacity_(capacity) {}

    bool get(const Key& key, Value&
value) {
        std::lock_guard<std::mutex>
lock(cache_mutex_);
        auto it =
cache_map_.find(key);
        if (it == cache_map_.end()) {
            return false;
        }
        // Move the accessed item to
the front of the list

cache_list_.splice(cache_list_.begin(), cache_list_, it->second);
        value = it->second->second;
        return true;
    }

    void put(const Key& key, const
Value& value) {
        std::lock_guard<std::mutex> lock(cache_mutex_);
        auto it = cache_map_.find(key);
        if (it != cache_map_.end()) {
            // Update existing item and move to front
            it->second->second = value;
            cache_list_.splice(cache_list_.begin(), cache_list_, it->second);
            return;
        }

        // Insert new item at the front
        cache_list_.emplace_front(key, value);
        cache_map_[key] = cache_list_.begin();

        // Evict least recently used item if capacity is exceeded
        if (cache_map_.size() > capacity_) {
            auto last = cache_list_.end();
            last--;
            cache_map_.erase(last->first);
            cache_list_.pop_back();
        }
    }

    size_t capacity() const {
        return capacity_;
    }
```

```
private:
    size_t capacity_;
    std::list<std::pair<Key, Value>>
cache_list_;
    std::unordered_map<Key,
typename std::list<std::pair<Key,
Value>>::iterator> cache_map_;
    std::mutex cache_mutex_;
};

// ------------------------------
// PersistentMemory Class
// ------------------------------
class PersistentMemory {
public:
    PersistentMemory(const
std::string& memory_file =
"persistent_memory.json",
                    size_t cache_capacity
= 1000)
        : memory_file_(memory_file),
          cache_(cache_capacity),

logger_(spdlog::basic_logger_mt(
"PMLL_logger", "pml_log.txt")) {
        logger_-
>set_level(spdlog::level::info);
        loadMemory();
    }

    ~PersistentMemory() {
        // Optional: Flush logs before
destruction

spdlog::drop("PMLL_logger");
    }

    // Add or update memory
```

```
asynchronously
    void addMemory(const
std::string& key, const json&
value) {
        {

std::lock_guard<std::mutex>
lock(memory_mutex_);
            memory_data_[key] =
value;
            cache_.put(key, value);
            logger_->info("Added/
Updated memory for key: {}", key);
        }
        saveMemoryAsync();
    }

    // Retrieve memory with caching
    json getMemory(const
std::string& key, const json&
default_value = nullptr) {
        json value;
        if (cache_.get(key, value)) {
            logger_->info("Cache hit
for key: {}", key);
            return value;
        }

        {

std::lock_guard<std::mutex>
lock(memory_mutex_);
            auto it =
memory_data_.find(key);
            if (it !=
memory_data_.end()) {
                cache_.put(key, it-
>second);
                logger_->info("Cache
miss for key: {}. Loaded from
storage.", key);
                return it->second;
            }
        }
        logger_->warn("Key not
found: {}", key);
        return default_value;
    }

    // Clear all memory
asynchronously
    void clearMemory() {
        {

std::lock_guard<std::mutex>
lock(memory_mutex_);
            memory_data_.clear();
            cache_ =
LRUCache<std::string,
json>(cache_.capacity()); // Reset
cache
            logger_->info("Cleared all
memory.");
        }
        saveMemoryAsync();
    }

    // Add a memory version
    void addMemoryVersion(const
std::string& key, const json&
value) {
        {

std::lock_guard<std::mutex>
lock(memory_mutex_);

memory_versions_[key].push_bac
k(value);
```

```
            memory_data_[key] = value;
            logger_->info("Added memory version for key: {}", key);
        }
        saveMemoryAsync();
    }

    // Retrieve a specific memory version
    json getMemoryVersion(const std::string& key, size_t version) {
        std::lock_guard<std::mutex> lock(memory_mutex_);
        if (memory_versions_.find(key) != memory_versions_.end() &&
            version < memory_versions_[key].size()) {
            logger_->info("Retrieved version {} for key: {}", version, key);
            return memory_versions_[key][version];
        }
        logger_->warn("Version {} for key: {} not found.", version, key);
        return nullptr;
    }

private:
    std::string memory_file_;
    std::unordered_map<std::string, json> memory_data_;
    LRUCache<std::string, json> cache_;
    std::mutex memory_mutex_;
```

```
std::shared_ptr<spdlog::logger>
logger_;

std::unordered_map<std::string,
std::vector<json>>
memory_versions_;

    // Load memory from file
    void loadMemory() {
        std::ifstream
infile(memory_file_);
        if (infile.is_open()) {
            try {
                json j;
                infile >> j;

std::lock_guard<std::mutex>
lock(memory_mutex_);
                if
(j.contains("memory_data")) {
                    memory_data_ =
j.at("memory_data").get<std::unordered_map<std::string,
json>>();
                }
                if
(j.contains("memory_versions")) {
                    memory_versions_ =
j.at("memory_versions").get<std::
unordered_map<std::string,
std::vector<json>>>();
                }
                logger_->info("Memory
loaded from file: {}",
memory_file_);
            } catch (const
json::parse_error& e) {
                logger_->error("Error
parsing memory file: {}", e.what());
```

```
            } catch (const
json::out_of_range& e) {
                logger_->error("Missing
keys in memory file: {}", e.what());
            }
            infile.close();
        } else {
            logger_->warn("Memory
file not found. Starting with empty
memory.");
        }
    }

    // Save memory to file
    void saveMemory() {
        std::lock_guard<std::mutex>
lock(memory_mutex_);
        std::ofstream
outfile(memory_file_);
        if (outfile.is_open()) {
            try {
                json j;
                j["memory_data"] =
memory_data_;
                j["memory_versions"] =
memory_versions_;
                outfile << j.dump(4);
                logger_->info("Memory
saved to file: {}", memory_file_);
            } catch (const
json::type_error& e) {
                logger_->error("Error
serializing memory data: {}",
e.what());
            }
            outfile.close();
        } else {
            logger_->error("Failed to
open memory file for writing: {}",
memory_file_);
        }
    }

    // Asynchronous save operation
    void saveMemoryAsync() {
        std::future<void> fut =
std::async(std::launch::async,
&PersistentMemory::saveMemory,
this);
        // Optionally, store futures if
you need to manage their lifetimes
    }
};

// -----------------------------
// IBC Client Implementation
// -----------------------------

class IBCClient {
public:
    IBCClient(const std::string&
base_url)
        : base_url_(base_url),

logger_(spdlog::basic_logger_mt(
"IBC_logger", "ibc_log.txt")) {
        logger_-
>set_level(spdlog::level::info);

curl_global_init(CURL_GLOBAL_DEFAULT);
    }

    ~IBCClient() {
        curl_global_cleanup();
        spdlog::drop("IBC_logger");
    }
    // Function to perform GET request
    json getRequest(const std::string& endpoint) {
        CURL* curl = curl_easy_init();
        json response_json;
        if(curl) {
            std::string read_buffer;
            curl_easy_setopt(curl, CURLOPT_URL, (base_url_ + endpoint).c_str());
            curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteCallback);
            curl_easy_setopt(curl, CURLOPT_WRITEDATA, &read_buffer);
            CURLcode res = curl_easy_perform(curl);
            if(res != CURLE_OK) {
                logger_->error("cURL GET request failed: {}", curl_easy_strerror(res));
            } else {
                try {
                    response_json = json::parse(read_buffer);
                    logger_->info("GET request to {} successful.", endpoint);
                } catch (const json::parse_error& e) {
                    logger_->error("JSON parse error: {}", e.what());
                }
            }
            curl_easy_cleanup(curl);
        }
        return response_json;
    }

    // Function to perform POST request with JSON payload
    json postRequest(const std::string& endpoint, const json& payload) {
        CURL* curl = curl_easy_init();
        json response_json;
        if(curl) {
            std::string read_buffer;
            std::string json_payload = payload.dump();
            struct curl_slist* headers = NULL;
            headers = curl_slist_append(headers, "Content-Type: application/json");
            curl_easy_setopt(curl, CURLOPT_URL, (base_url_ + endpoint).c_str());
            curl_easy_setopt(curl, CURLOPT_POST, 1L);
            curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers);
            curl_easy_setopt(curl, CURLOPT_POSTFIELDS, json_payload.c_str());
            curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, WriteCallback);
            curl_easy_setopt(curl, CURLOPT_WRITEDATA, &read_buffer);
            CURLcode res = curl_easy_perform(curl);
            if(res != CURLE_OK) {
                logger_->error("cURL
POST request failed: {}",
curl_easy_strerror(res));
            } else {
                try {
                    response_json =
json::parse(read_buffer);
                    logger_->info("POST
request to {} successful.",
endpoint);
                } catch (const
json::parse_error& e) {
                    logger_->error("JSON
parse error: {}", e.what());
                }
            }

curl_slist_free_all(headers);
            curl_easy_cleanup(curl);
        }
        return response_json;
    }

private:
    std::string base_url_;

std::shared_ptr<spdlog::logger>
logger_;

    // cURL write callback
    static size_t
WriteCallback(void* contents,
size_t size, size_t nmemb, void*
userp)
    {
        ((std::string*)userp)-
>append((char*)contents, size *
nmemb);
        return size * nmemb;
    }
```

```cpp
};

// -----------------------------
// Example Usage
// -----------------------------
int main() {
    // Initialize PersistentMemory with default file and cache capacity
    PersistentMemory pm;

    // Initialize IBCClient with Cosmos SDK node's base URL (adjust as needed)
    IBCClient ibc_client("http://localhost:1317"); // Default REST port for Cosmos SDK

    // Example: Query IBC client states
    json client_states = ibc_client.getRequest("/ibc/core/client/v1/client_states");
    std::cout << "IBC Client States: " << client_states.dump(4) << std::endl;

    // Add memory entries
    pm.addMemory("user_123", { {"name", "Josef"}, {"last_topic", "PMLL architecture"} });
    pm.addMemory("user_456", { {"name", "Alice"}, {"last_topic", "Distributed Systems"} });

    // Retrieve memory entries
    json user1 = pm.getMemory("user_123");
    if (!user1.is_null()) {
        std::cout << "Welcome back,
" << user1["name"]
                  << "! Last time we
discussed " << user1["last_topic"]
<< "." << std::endl;
    } else {
        std::cout << "Welcome! Let's
start our conversation." <<
std::endl;
    }

    json user2 =
pm.getMemory("user_456");
    if (!user2.is_null()) {
        std::cout << "Hello, " <<
user2["name"]
                  << "! Last time we talked
about " << user2["last_topic"] <<
"." << std::endl;
    } else {
        std::cout << "Hello! Let's
begin our interaction." <<
std::endl;
    }

    // Update a memory entry
    pm.addMemory("user_123",
{ {"name", "Josef"}, {"last_topic",
"GPT-5 ETA"} });

    // Add a new version to a
memory entry

pm.addMemoryVersion("user_123
", { {"name", "Josef"},
{"last_topic", "Advanced PMLL
Features"} });

    // Retrieve a specific memory
```

```
version
    json version1 = pm.getMemoryVersion("user_123", 0); // First version
    if (!version1.is_null()) {
        std::cout << "Version 1 for user_123: " << version1.dump() << std::endl;
    } else {
        std::cout << "No version 1 found for user_123." << std::endl;
    }

    // Attempt to retrieve a non-existent key
    json user3 = pm.getMemory("user_789");
    if (user3.is_null()) {
        std::cout << "No records found for user_789." << std::endl;
    }

    // Example: Initiate an IBC transfer (pseudo-code, requires proper setup)
    /*
    json transfer_payload = {
        {"source_port", "transfer"},
        {"source_channel", "channel-0"},
        {"sender", "cosmos1..."},
        {"receiver", "cosmos1..."},
        {"token", {
            {"denom", "atom"},
            {"amount", "100"}
        }},
        {"timeout_height", {
            {"revision_number", "1"},
            {"revision_height", "200"}
        }},
        {"timeout_timestamp", "0"}
    };
    json transfer_response = ibc_client.postRequest("/ibc/core/transfer/v1beta1/transfer", transfer_payload);
    std::cout << "IBC Transfer Response: " << transfer_response.dump(4) << std::endl;
    */

    // Simulate some delay to allow asynchronous save operations to complete
    std::this_thread::sleep_for(std::chrono::seconds(2));

    // Clear all memory
    pm.clearMemory();

    return 0;
}
```