

UNIVERSITA' DEGLI STUDI DI MESSINA

NETWORKING COURSE PROJECT

AutoSyncGen

March 20, 2015

Author:
Vittorio ROMEO

Professors:
Antonio PULIAFITO
Giovanni MERLINO



<http://unime.it>



<http://vittorioromeo.info>

Contents

1	Introduction	1
1.1	Problem	1
1.2	Solution	1
1.3	Technologies used	2
2	Network module	4
2.1	SFML	4
2.2	Packets	5
2.2.1	Packet types	5
2.2.2	Packet handling	6
2.3	SessionHost	6
2.3.1	SessionServer	9
2.3.2	SessionClient	11
3	Synchronization module	12
3.1	SyncObj	12
3.1.1	Fields	13
3.2	Snapshot	14
3.2.1	SnapshotTypeData	15
3.3	Diff	15
3.3.1	DiffTypeData	15
3.4	LifetimeManager	16
3.5	SyncManager	17
3.5.1	Lifetime managers tuple	17
3.5.2	Handle maps tuple	17
3.5.3	Object management functions	17
4	Example application: synchronized chat	21
4.1	Data structures	21
4.1.1	Message	21
4.2	Lifetime manager specialization	21
4.2.1	Pointers as handles	21
4.2.2	create	21
4.2.3	remove	21
4.3	Application packet types	21
4.3.1	Server to client	21
4.3.2	Client to server	21
4.4	ConsoleSessionController	21
4.4.1	Server role	21
4.4.2	Client role	21

Chapter 1

Introduction

What is *AutoSyncGen* and what is it trying to achieve?

1.1 Problem

Many networked applications require *constant synchronization* between the server and the connected clients. Software such as *multiplayer online games* completely rely on the fact that a synchronized state is maintained between the server and the players. Other applications that are not reliant on state synchronization may still benefit from such a system. A *basic chatroom program*, for example, may use state synchronization to keep track of sent messages and allow newly-connected users to browse the message history.

Implementing state synchronization is not a trivial task. It is usually required to define a simple protocol that *explicitly handles* state synchronization for all data structures.

This is usually achieved by explicitly coding serialization behavior and explicitly defining how packets should be created and read for every synchronizable objects. Even for small applications, maintaining and expanding the code using a similar approach can become difficult quickly.

1.2 Solution

The solution proposed in this paper is *AutoSyncGen*, a *C++14* library written using *modern and idiomatic code*. The library allows developers to quickly define synchronizable data structures, and automatically deals with *object lifetime management*, *serialization* and *deserialization*, *packet creation* and *reading*.

By quickly specifying how object memory should be managed and the types of the synchronizable fields, *AutoSyncGen* generates code at *compile-time* to keep track of the entire application state and generate synchronization requests and responses that send the smallest amount of data required to fully synchronize the clients to the server.

Here's an example of a synchronizable data structure defined using the AutoSyncGen library.

```
1 // Synchronizable data structure representing a chat message.
2 struct Message : syn::SyncObj
3 <
4     // The fields types are passed as a variadic type list.
5
6     int,           // messageID
7     std::string,   // author
8     std::string    // contents
9 >
10 {
11     // Field proxies are generated via macros for convenience.
12
13     SYN_PROXY(0, messageID);
14     SYN_PROXY(1, author);
15     SYN_PROXY(2, contents);
16 };
```

A shorter (but more preprocessor-heavy) construct can be used, which reduces code repetition.

```
1 // Synchronizable data structure representing a chat message.
2 SYN_DEFINE_SYNCOBJ
3 (
4     // Name of data structure.
5     Message,
6
7     // Tuple of synchronizable fields.
8     (
9         (int, messageID),
10        (std::string, author),
11        (std::string, contents)
12    )
13 );
```

The library is structured in such a way that most of the user code can be *shared between the client and the server*. It is not necessary to re-define data structures or lifetime management instructions twice.

1.3 Technologies used

C++14, which is the latest official C++ standard, released in 2 March 2014 (*paper N3936*), is the language of choice for AutoSyncGen. Like C++11, this newer standard is a huge step forward for the language. *Smarter memory managment, automatic type deduction*, and countless new programming and metaprogramming features allow developers to write much safer and powerful code. C++11 and C++14 features make AutoSyncGen possible.

On top of C++14 and its standard library, the *SSV framework* is being used as well. This framework was completely written from scratch by me, *Vittorio Romeo*, and is available under the open-source AFL3.0 license on GitHub.

The *SSVUtils* library, used throughout the whole program, features many heterogeneous self-contained modules: an efficient and modern handwritten JSON parser, a preprocessor metaprogramming module, a template metaprogramming module, automatic console formatted output for containers and user-defined types, efficient data structures (bimap, handle vector, growable arrays), advanced memory management facilities, type-safe variadic unions, handwritten templating system, filesystem management, easy benchmarking of portions of code, and much more.

Two additional dependencies for AutoSyncGen are the *SFML* library, which offers lightweight abstraction over sockets and packets, and the *SSVStart* library, part of the SSV framework, which extends the functionality of SFML and wraps some of its C++03 abstractions with more modern and safer C++14 constructs.

Chapter 2

Network module

AutoSyncGen is divided in two main modules. The *network module* deals with client/server abstraction and communication, and will be covered in this chapter. The *synchronization module* deals with compile-time synchronization data structures generation and synchronization algorithms, and will be covered in the next chapter.

2.1 SFML

As mentioned in the previous chapter, SFML is one of the dependencies of AutoSyncGen. SFML offers convenient and type-safe wrappers over basic networking constructs such as *sockets* and *packets*.

AutoSyncGen's network module extensively uses these abstractions to allow cross-platform compatibility and additional safety.

```
1  // SFML offers an 'IpAddress' data structure that that can be initialized from strings.
2  using sf::IpAddress;
3
4  // SFML offers a 'Packet' object that can be filled/emptied using stream operators.
5  using sf::Packet;
6
7  // SFML offers an 'UdpSocket' class that properly manages and abstracts the memory
8  // and functionality of an UDP socket.
9  using sf::UdpSocket;
```

Just to clarify how convenient these abstractions are, here's a small code example that shows *sf::Packet* usage.

```
1  int x{5};
2  std::string y{"hello!"};
3  double z{2.52};
4
5  sf::Packet toSend;
6  toSend >> x >> y >> z;
7
8  // Send packet through the UDP socket...
9
```

```
10  int outX;
11  std::string outY;
12  double outZ;
13
14  sf::Packet received;
15  received << outX << outY << outZ;
```

SFML is open-source, cross-platform and free.

2.2 Packets

AutoSyncGen uses a simple but effective protocol for data synchronization, which requires some special packets.

2.2.1 Packet types

Packet types are defined using *C++11 enum classes*, which are type-safe enumeration types that cannot be implicitly converted to their underlying representation.

CtoS enumeration

This enumeration holds the packet types that are sent from the client to the server.

```
1  enum class CtoS : NType
2  {
3      // Request to establish a connection to the server
4      ConnectionRequest = 0,
5
6      // Ping to avoid timing out with the server
7      Ping = 1,
8
9      // Request to sync data with server
10     SyncRequest = 2,
11
12     // Informs the server that the synchronization was successful
13     SyncSatisfied = 3,
14
15     // Custom data packet
16     Data = 4,
17 };
```

StoC enumeration

This enumeration holds the packet types that are sent from the server to the client.

```
1  enum class StoC : NType
2  {
3      // Accept a requested client connection, assigning the client a CID
4      ConnectionAccept = 0,
```

```

5
6     // Decline a client's requested connection
7     ConnectionDecline = 1,
8
9     // Satisfy sync request (if the client's revision is behind)
10    SyncRequestSatisfy = 2,
11
12    // Sync not needed (client has the same revision as the server)
13    SyncRequestUnneeded = 3,
14
15    // Sync request declined (overload/technical issue)
16    SyncRequestDecline = 4,
17
18    // Custom data packet
19    Data = 5,
20 };

```

2.2.2 Packet handling

Packets are automatically handled by the client and the server, using a switch.

```

1  void handle(RPT mType)
2  {
3      switch(mType)
4      {
5          case RPT::ConnectionAccept:
6              handleConnectionAccept();
7              return;
8          case RPT::ConnectionDecline:
9              handleConnectionDecline();
10             return;
11             // ...
12     }
13 }

```

This implementation is fine for the current packet types AutoSyncGen uses, but can be changed to a lookup table or an array of function pointers if more packet types need to be added in the future.

2.3 SessionHost

Clients and servers are abstracted as *SessionHost* classes in the network module. The *SessionHost* classes defines data structures and functionality shared both by clients and servers.

Every host has a dedicated extra thread for data and socket management.

```

1  // 'SessionHost' is a CRTP template class, to allow compile-time generation
2  // of appropriate data structures.
3  template
4  <
5      // 'struct' containing global synchronization settings.

```

```

6     typename TSettings,
7
8     // Type of sent packets.
9     typename TSPT,
10
11    // Type of received packets.
12    typename TRPT,
13
14    // CRTP (curiously recurring template pattern) derived type.
15    template<typename> class TDerivedBase
16 >
17 class SessionHost
18 {
19     // Typedefs of data structures that depend on the template parameters.
20 public:
21     // CRTP type.
22     using Derived = TDerivedBase<Settings>;
23
24     // Packet types.
25     using SPT = TSPT;
26     using RPT = TRPT;
27
28     // Synchronization classes.
29     using Settings = TSettings;
30     using SyncManager = typename Settings::SyncManager;
31     using Diff = typename SyncManager::DiffType;
32     using Snapshot = typename SyncManager::SnapshotType;
33
34 private:
35     // Name of the host.
36     std::string name;
37
38     // IP address of the host.
39     IPAddress ip;
40
41     // Port the host is listening on.
42     Port port;
43
44     // UDP socket of the host.
45     UdpSocket socket;
46
47     // Status of the host.
48     bool busy{false};
49
50     // Tries binding the socket to the specified port.
51     void tryBindSocket();
52
53     // Tries forwarding the received packets to the underlying
54     // client/server implementation.
55     void tryForwardReceivedPacket();
56

```

```

57     // Function running in the receive thread, which keeps track
58     // of the data receiving buffer and of the socket state.
59     void receiveThread();
60
61 protected:
62     // Buffers that store received data and data that will be sent.
63     Packet sendBuffer, recvBuffer;
64
65     // 'void' future offering a RAII wrapper for the host's thread.
66     std::future<void> hostFuture;
67
68     // IP of the sender of the last received data.
69     IPAddress senderIp;
70
71     // Port of the sender of the last received data.
72     Port senderPort;
73
74     // Synchronization manager.
75     SyncManager syncManager;
76
77     // Sends the content of the send buffer to the specified host.
78     void sendTo(const IPAddress& mIp, const Port& mPort);
79
80     // Builds a packet from a variadic argument list.
81     template<SPT TType, typename... TArgs> void mkPacket(TArgs&&... mArgs);
82
83     // Sets the state of the host.
84     void setBusy(bool mBusy) noexcept { busy = mBusy; }
85
86     // Pops data of type 'T' from the receive buffer.
87     template<typename T> auto popRecv();
88
89 public:
90     // Constructs the host using a specified name and port.
91     // This function tries to bind the socket and start the dedicated thread.
92     SessionHost(std::string mName, syn::Port mPort);
93
94     // Getters.
95     const auto& getName() const noexcept;
96     const auto& getIp() const noexcept;
97     const auto& getPort() const noexcept;
98     const auto& isBusy() const noexcept;
99     auto& getSyncManager() noexcept;
100 };
101
102 // Template typedef for the base 'SessionHost' used by servers.
103 template<typename TSettings>
104 using SessionServerBase = SessionHost<TSettings, PT::StoC, PT::CtoS, SessionServer>;
105
106 // Template typedef for the base 'SessionHost' used by clients.
107 template<typename TSettings>

```

```
108 using SessionClientBase = SessionHost<TSettings, PT::CtoS, PT::StoC, SessionClient>;
```

2.3.1 SessionServer

The *SessionServer* CTRP derivation of *SessionHost* requires additional data structures and algorithms to deal with client connections and storage of the client's latest synchronization state snapshots.

ClientHandler

A *ClientHandler* instance is created and managed for every client connected to the server.

```
1  class ClientHandler
2  {
3      private:
4          // Connect client ID.
5          // Assigned by the server.
6          CID cid{nullCID};
7
8          // IP and Port of the client.
9          IPAddress ip;
10         Port port;
11
12         // Seconds until the client is timed out.
13         int secondsUntilTimeout;
14
15         // State of the client handler.
16         bool busy{false};
17
18     public:
19         // Assumes the handler is not busy and binds it to a client with a
20         // specific client ID, IP and Port.
21         void bindToClient(CID mCID, const IPAddress& mIp, const Port& mPort);
22
23         // Unbinds the handler from the client connection.
24         void unbindFromClient();
25
26         // Getters
27         const auto& getCID() const noexcept;
28         const auto& getIp() const noexcept;
29         const auto& getPort() const noexcept;
30     };
```

Client handlers are stored and managed in the *ClientHandlerManager* class, of which the server has an instance.

ClientHandlerManager

The *ClientHandlerManager* manages memory and functionality of *ClientHandler* instances.

A *ssvu::MonoRecycler* is used to store the handlers on the heap and recycle their memory. The manager also stores two vector of pointers to client handlers: one with non-busy handlers, and one with busy handlers.

The lists are periodically checked by an extra thread to remove timed-out client handlers and to accept new connections.

```
1  class ClientHandlerManager
2  {
3      private:
4          // Next available unique client ID.
5          CID nextCID{0};
6
7          // Memory management and recycling data structure for 'ClientHandler' instances.
8          ssvu::MonoManager<ClientHandler> clientHandlers;
9
10         // Vector of non-busy client handlers.
11         std::vector<ClientHandler*> chAvailable
12
13         // Vector of busy client handlers.
14         std::vector<ClientHandler*> chBusy;
15
16         // Map of client handlers, by client ID.
17         std::map<CID, ClientHandler*> chMap;
18
19         // 'void' future that wraps the thread which checks timeouts for client handlers.
20         std::future<void> timeoutFuture;
21
22         // Vector of client handler IDs that need to be disconnected.
23         std::vector<CID> toDisconnect;
24
25         // State of the client handler manager.
26         bool busy{true};
27
28         // Creates a client handler instance from the 'MonoManager'
29         // and inserts it into the available vector.
30         void createClientHandler();
31
32         // Method ran in the timeout thread.
33         // Periodically checks if clients have timed out.
34         void runTimeout();
35
36     public:
37         // Accepts a new client connection.
38         // Uses an existing free handler, if available, otherwise allocates a new one.
39         // Also removes all handlers that have to be disconnected and updates the ID map.
40         auto& acceptClient(const IPAddress& mIp, const Port& mPort);
41
42         // Returns true if an handler exists, by ID.
43         bool has(CID mCID) const noexcept;
44
45         // Gets a client handler by ID.
46         auto& operator[](CID mCID) noexcept;
47
48         // Refreshes the timeout duration when a ping is received for a specific client.
```

```
49         void pingReceived(CID mCID) noexcept;  
50     };
```

2.3.2 SessionClient

The *SessionClient* CTRP derivation of *SessionHost* does not require any particular additional data structure. The only additions, compared to the *SessionHost* base class, are the IP and Port of the current server.

Chapter 3

Synchronization module

3.1 SyncObj

The *SyncObj* class represents a synchronizable data structure. It is composed of synchronizable fields that automatically get serialized/deserialized and sent/received by the *SyncManager*.

SyncObj is a template class that derives from a polymorphic base class. It takes a variadic amount of template arguments that are the types of the fields stored by the *SyncObj*.

JSON is currently used for serialization/deserialization of objects. The code architecture makes it easy to implement and use alternative serialization methods.

```
1  template
2  <
3      // Variadic type list of field types.
4      typename... TArgs
5  >
6  class SyncObj : public Impl::ObjBase
7  {
8      public:
9          // Type of the tuple of fields.
10         using TplFields = ssvu::Tpl<TArgs...>;
11
12     private:
13         // Count of fields.
14         static constexpr SizeT fieldCount{sizeof...(TArgs)};
15
16         // Tuple of fields.
17         TplFields fields;
18
19         // Dirty/clean bits for the fields.
20         std::bitset<fieldCount> fieldFlags;
21
22     public:
23         // Type of the field at index 'TI'.
24         template<TypeIdx TI> using TypeAt = ssvu::TplElem<TI, decltype(fields)>;
25
26         // Type of the field proxy at index 'TI'.
27         template<TypeIdx TI> using ProxyAt = FieldProxy<TI, SyncObj<TArgs...>>;
```

```

28
29     private:
30         // Non-const reference to the field at index 'TI'.
31         template<TypeIdx TI> auto& getFieldAt() noexcept;
32
33         // Sets the bit at index 'TI' to true.
34         template<TypeIdx TI> void setBitAt() noexcept;
35
36         // Sets the bit at index 'mI' to false.
37         void unsetBitAt(TypeIdx mI) noexcept;
38
39     public:
40         // Returns a proxy for the field at index 'TI'.
41         template<TypeIdx TI> auto get() noexcept;
42
43         // Sets the object state from an 'mX' json value.
44         void setFromJson(const ssvj::Val& mX);
45
46         // Serializes all the fields to json.
47         auto toJsonAll();
48
49         // Serializes all dirty fields to json.
50         auto toJsonDirty();
51 };

```

3.1.1 Fields

Field tuple

The field are stored in an *std::tuple*, generated at compile-time from the passed variadic type list.

Field flags

Every field has a corresponding bit that keeps track of whether it has been modified or not. This bit is used to quickly determine the fields that have to be sent again from the server, that were changed since the last sent revision.

The bits are stored in a *std::bitset*, generated at compile-time with size equal to the number of fields.

FieldProxy

The *FieldProxy* class is a convenient abstraction to help the developer get and set synchronizable object fields.

Since the fields are stored into a tuple, it is impossible to assign a name to them, unlike traditional C++ structs and classes. The *FieldProxy* abstraction allows the developer to use a name to refer to synchronizable fields.

It also automatically sets the corresponding field *dirty bit* to true when the underlying data is modified.

```

1     template
2     <
3         // Index of the field in the tuple.
4         TypeIdx TI,

```

```

5
6     // Type of attached 'SyncObj'.
7     typename TObj
8
9 >
10 class FieldProxy
11 {
12     private:
13         // Reference to the attached object.
14         TObj& syncObj;
15
16     public:
17         // Constructs the 'FieldProxy' from a reference to its parent object.
18         FieldProxy(TObj& mSyncObj) noexcept;
19
20         // Returns a non-const reference to the underlying field, and sets the dirty bit.
21         auto& edit() noexcept;
22
23         // Returns a const reference to the underlying field.
24         const auto& view() const noexcept;
25 };

```

3.2 Snapshot

A *Snapshot* is a complete representation of the current synchronizable state.

The *Snapshot* stores the state of all types of objects, separating them by type in appropriate data structures. It also stores a bitset for every object type, where the n -th bit represents the state of the n -th object of a specific type (0 for *dead*, 1 for *alive*).

```

1 // Class representing the snapshot of the entire state of the 'SyncManager'.
2 template<typename TManager> struct Snapshot
3 {
4     // Alive/dead bitset storage per object type.
5     BitsetStorage bitsetIDs;
6
7     // Tuple that stores a snapshot type data for every synchronizable type.
8     ssvu::TplRepeat<TypeData, TManager::typeCount> typeDatas;
9
10    // Serializes the snapshot to JSON.
11    auto toJson() const;
12
13    // Initializes the snapshot from JSON.
14    void initFromJson(const ssvj::Val& mX);
15
16    // Returns the difference between this snapshot and 'mX'.
17    auto getDiffWith(const Snapshot& mX);
18 };

```

3.2.1 SnapshotTypeData

A *Snapshot* is composed of *SnapshotTypeData* objects. Every single type data object represents the synchronization state for a specific type.

```
1  // Class representing the snapshot for a specific synchronizable type.
2  struct SnapshotTypeData
3  {
4      // Items contained in the snapshot type data.
5      std::map<ID, ssvj::Val> items;
6
7      // Serializes the snapshot type data to JSON.
8      auto toJson() const;
9
10     // Initializes the snapshot type data from JSON.
11     void initFromJson(const ssvj::Val& mX);
12 }
```

‘items’ map

The *items* map simply contains all the objects of a specific type, serialized to JSON. The keys are the unique IDs of the objects, the values are their JSON representation.

3.3 Diff

A *Diff* is a representation of the difference between two snapshot instances.

```
1  // Class representing a diff for all the types handled by the ‘SyncManager’.
2  template<typename TManager> struct Diff
3  {
4      // Tuple containing the diff type data for every synchronizable type.
5      ssvu::TplRepeat<TypeData, TManager::typeCount> typeDatas;
6
7      // Serializes the diff to a JSON value.
8      auto toJson() const;
9
10     // Initializes the diff from a JSON value.
11     void initFromJson(const ssvj::Val& mX);
12
13     // Returns true if there is no difference between snapshots.
14     bool isEmpty() const noexcept;
15 }
```

3.3.1 DiffTypeData

A *Diff* is composed of *DiffTypeData* objects. Every single type data object represents the synchronization difference for a specific type.

```

1  // Class representing the diff for a specific type managed by the 'SyncManager'.
2  struct DiffTypeData
3  {
4      // Map of objects that have been created since last snapshot.
5      std::map<ID, ssvj::Val> toCreate;
6
7      // Map of objects that have been updated since last snapshot.
8      std::map<ID, ssvj::Val> toUpdate;
9
10     // Vector of objects that have been removed since last snapshot.
11     std::vector<ID> toRemove;
12
13     // Serializes the diff type data to JSON.
14     auto toJson() const;
15
16     // Initializes the data from JSON.
17     void initFromJson(const ssvj::Val& mX);
18
19     // Returns true if there are no differences.
20     bool isEmpty() const noexcept;
21 };

```

'toCreate' map

The *toCreate* map stores all the objects of a specific type that have been created since the last snapshot. The keys are the unique IDs of the objects, the values are their JSON representation.

'toUpdate' map

The *toUpdate* map stores all the objects of a specific type that have been modified since the last snapshot. The keys are the unique IDs of the objects, the values are their JSON difference representation.

'toRemove' vector

The *toRemove* vector stores all the unique IDs of objects of a specific type that have been deleted since the last snapshot. No JSON values are required, since the objects will be completely removed from the synchronization manager.

3.4 LifetimeManager

In order to allow maximum flexibility in object storage, a *LifetimeManager* T_i template class specialization must be provided to decide how to allocate/deallocate memory for the synchronizable objects, and how to refer to specific objects through *handles*.

Here's an example of an user-defined *LifetimeManager*, for the type *Message*.

```

1  // 'LifetimeManager' specialization for 'Message'.
2  template<> struct LifetimeManager<Message>
3  {
4      // Required typedef of the handle used by the 'SyncManager'.
5      using Handle = Message*;

```

```

6
7 // Required function that returns a null handle.
8 Handle getNullHandle() noexcept;
9
10 // Required function that creates an object and returns an handle to it.
11 Handle create();
12
13 // Required function that removes an object attached to a specific handle.
14 void remove(Handle mHandle);
15
16 // ...user data...
17 };

```

3.5 SyncManager

The *SyncManager* is the core of the AutoSyncGen library. The *SyncManager* is a template class that takes a variadic list of unique synchronizable types and creates the appropriate synchronization data-structures for them at compile time.

Every object has a specific *type ID*, which corresponds to the position of the type in the passed variadic list, and a *unique ID* per instance.

An instance of the *SyncManager* is used both in the client and in the server code. The manager can serialize its entire state and serialize differences between the current state and the previous state.

Serialized difference data structures received from the server can be applied to the client's *SyncManager* to synchronize its state.

The manager also contains data structures and convenient functions to retrieve an handle to an alive object by unique ID, or check the state of a specific object.

3.5.1 Lifetime managers tuple

Creation and destruction of objects is defined in *LifetimeManager<T>* specializations, by the user of AutoSyncGen. During instantiation of the *SyncManager*, a tuple containing an instance of the specializations, per type, is generated at compile-time.

When creating or destroying an object after applying a *Diff* or explicitly modifying the state from the server, the appropriate *LifetimeManager* implementation will be used.

3.5.2 Handle maps tuple

It may be necessary to retrieve an object of a specific type through its unique ID. The current implementation of *SyncManager* offers this possibility to the user through key-value maps, where the key is the unique ID of the object and the value is an handle to the object itself.

It is guaranteed that the unique IDs of the objects are synchronized between client and server.

3.5.3 Object management functions

Since objects need to be created, removed and updated at run-time, when the *Diff* instance is received, some dispatch data structures need to be created to make sure the correct constructors/destructors are called.

This is done through the compile-time generation of three *std::array* of member function pointers, that point to implicit specialization of templated creation/removal/update member functions of the *SyncManager* itself.

This way, explicit polymorphic type erasure a-la *std::function* is avoided, to optimize run-time performance and memory usage.

```

1  template
2  <
3      // LifetimeManager template template parameter.
4      template<typename> class TLFManager,
5
6      // Variadic list of synchronizable types.
7      typename... TTypes
8  >
9  class SyncManager
10 {
11     public:
12         // Count of the types managed by this 'SyncManager'.
13         static constexpr SizeT typeCount{sizeof...(TTypes)};
14
15         // Lifetime manager for the type 'T'.
16         template<typename T> using LFManagerFor = TLFManager<T>;
17
18         // Handle for the type 'T'.
19         template<typename T> using HandleFor = typename LFManagerFor<T>::Handle;
20
21         // Handle map for the type 'T'.
22         template<typename T> using HandleMapFor = std::map<ID, HandleFor<T>>;
23
24         // Type of this manager.
25         using ThisType = SyncManager<TLFManager, TTypes...>;
26
27         // Type of snapshot.
28         using SnapshotType = Impl::Snapshot<ThisType>;
29
30         // Type of diff.
31         using DiffType = Impl::Diff<ThisType>;
32
33         // Type of bitset that keeps track of alive objects.
34         using ObjBitset = std::bitset<maxObjs>;
35
36         // Type of alive/dead bitset storage per type.
37         using BitsetStorage = std::array<ObjBitset, typeCount>;
38
39     private:
40         // Type of lifetime managers tuple.
41         using TplLFManagers = ssvu::Tpl<LFManagerFor<TTypes>...>;
42
43         // Type of handle maps tuple.
44         using TplHandleMaps = ssvu::Tpl<HandleMapFor<TTypes>...>;
45
46         // Type of next available ID-per-type tuple.
47         using TplIDs = ssvu::TplRepeat<ID, typeCount>;
48
49         // Type of member function that creates objects.
50         using MemFnCreate = void(ThisType::*)(ID, const ssvj::Val&);

```

```

51
52 // Type of member function that removes objects.
53 using MemFnRemove = void(ThisType::*)(ID);
54
55 // Type of member function that updates objects.
56 using MemFnUpdate = void(ThisType::*)(ID, const ssvj::Val&);
57
58 // Tuple containing all the lifetime managers.
59 TplLFManagers lfManagers;
60
61 // Tuple containing the handle maps.
62 TplHandleMaps handleMaps;
63
64 // Tuple containing the last ID for every type.
65 TplIDs lastIDs;
66
67 // Array containing the creation functions per type.
68 std::array<MemFnCreate, typeCount> funcsCreate;
69
70 // Array containing the removal functions per type.
71 std::array<MemFnRemove, typeCount> funcsRemove;
72
73 // Array containing the update functions per type.
74 std::array<MemFnUpdate, typeCount> funcsUpdate;
75
76 // Array containing the alive/dead bitset for per type.
77 BitsetStorage bitsetIDs;
78
79 // Returns the alive/dead bitset for type 'T'.
80 template<typename T> const auto& getBitsetFor() const noexcept;
81
82 // Returns true if the object 'mID' of type 'T' is alive.
83 template<typename T> bool isAlive(ID mID) const noexcept;
84
85 // Sets the alive/dead for the object 'mID' of type 'T'.
86 template<typename T> void setAlive(ID mID, bool mX) noexcept;
87
88 // Creates an object of type 'T' with id 'mID' from the JSON value 'mVal'.
89 // Member function pointers in 'funcsCreate' point to instantiations of
90 // this template member function.
91 template<typename T> void createImpl(ID mID, const ssvj::Val& mVal);
92
93 // Removes an object of type 'T' with id 'mID' from the JSON value 'mVal'.
94 // Member function pointers in 'funcsRemove' point to instantiations of
95 // this template member function.
96 template<typename T> void removeImpl(ID mID);
97
98 // Updates an object of type 'T' with id 'mID' from the JSON value 'mVal'.
99 // Member function pointers in 'funcsUpdate' point to instantiations of
100 // this template member function.
101 template<typename T> void updateImpl(ID mID, const ssvj::Val& mVal);

```

```

102
103 public:
104     // Returns the unique type ID for the type 'T'.
105     template<typename T> static constexpr ID getTypeID() noexcept;
106
107     // Returns the first available unique ID for the type 'T'.
108     template<typename T> ID getFirstFreeID() noexcept;
109
110     // Returns a new null handle for the type 'T'.
111     template<typename T> auto getNullHandleFor() noexcept;
112
113     // Returns a reference for the 'LifetimeManager' for the type 'T'.
114     template<typename T> auto& getLFManagerFor() noexcept;
115
116     // Returns a the ID<->Handle map for the type 'T'.
117     template<typename T> auto& getHandleMapFor() noexcept;
118
119     // Returns a new handle for the object of type 'T' with ID 'mID'.
120     template<typename T> auto& getHandleFor(ID mID) noexcept;
121
122     // Explicitly creates an object of type 'T'.
123     // Intended to be used only from the server.
124     template<typename T> auto serverCreate(const ssvj::Val& mVal);
125
126     // Applies a 'Diff' to the current state.
127     void applyDiff(const DiffType& mX);
128
129     // Returns the current state of the manager as a 'Snapshot'.
130     auto getSnapshot();
131 };

```

Chapter 4

Example application: synchronized chat

The current version of the *AutoSyncGen* library comes with a very simple example application to show basic functionality of the library.

The application is a *client-server chat* with persistent message history. Clients can connect to the server, send messages and edit existing messages. Clients maintain their state synchronized with the server thanks to periodic synchronization requests.

4.1 Data structures

4.1.1 Message

The *Message* data structure is a simple synchronizable object containing an unique ID for the message, the name of the author and the contents of the message.

```
1  // Synchronizable data structure representing a chat message.
2  struct Message : syn::SyncObj
3  <
4      int,           // messageID
5      std::string,   // author
6      std::string    // contents
7  >
8  {
9      SYN_PROXY(0, messageID);
10     SYN_PROXY(1, author);
11     SYN_PROXY(2, contents);
12 };
```

4.2 Lifetime manager specialization

To allow creation, removal and update of *Message* instances, a lifetime manager specialization must be provided. In this case, the specialization uses heap-allocation and raw pointers as handles for simplicity.

```

1  // 'LifetimeManager' specialization for 'Message'.
2  template<> struct LifetimeManager<Message>
3  {
4      // Type of the handle used by the 'SyncManager'.
5      using Handle = Message*;
6
7      // Required function that returns a null handle.
8      Handle getNullHandle() noexcept
9      {
10         // Since raw pointers are being used, 'nullptr' will be
11         // our null handle.
12         return nullptr;
13     }
14
15     // Required function that creates an object and returns an handle to it.
16     Handle create()
17     {
18         // To create a object, we simply use 'std::make_unique' and
19         // emplace the resultant 'std::unique_ptr' in the 'storage'
20         // vector. Afterwards, we just return a raw pointer to the
21         // newly created object.
22         return &ssvu::getEmplaceUPtr<Message>(storage);
23     }
24
25     // Required function that removes an object attached to a specific handle.
26     void remove(Handle mHandle)
27     {
28         // To deallocate and destroy a message, we simply look for the
29         // message with the same handle in the 'storage', then remove
30         // it. Since it's being allocated using 'std::unique_ptr', its
31         // memory will be automatically released.
32         ssvu::eraseRemoveIf(storage, [this, mHandle](const auto& mUPtr)
33         {
34             return mUPtr.get() == mHandle;
35         });
36     }
37
38     // Internal memory storage for 'Message' instances.
39     std::vector<ssvu::UPtr<Message>> storage;
40 };

```

4.3 Application packet types

The example application defines some custom packet types for client-server communication.

```

1  // Custom unique types for server-to-client chat packets.
2  enum class DP_StoC : int
3  {
4      // Send a specific message to the client.
5      DisplayMsg = 0

```



```

6  };
7
8  // Custom unique types for client-to-server chat packets.
9  enum class DP_CtoS : int
10 {
11     // Send a message to the server.
12     SendMsg = 0,
13
14     // Edit a message on the server.
15     EditMsg = 1
16 };

```

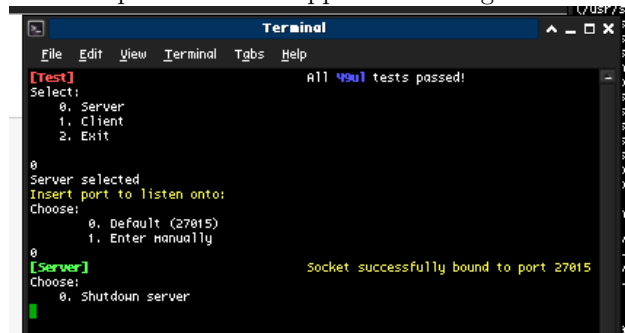
4.4 ConsoleSessionController

The console is used to control the chat application. Interaction with the user is handled thanks to the *stdin* input stream. Information is displayed to the user via the *stdout* output stream.

When the application is started, the role of the current process can be chosen.

4.4.1 Server role

Figure 4.1: Example of the chat application being ran in server mode.



When the application is running in server mode, the only possible user command is the shutdown of the application itself. Everything else is handled automatically.

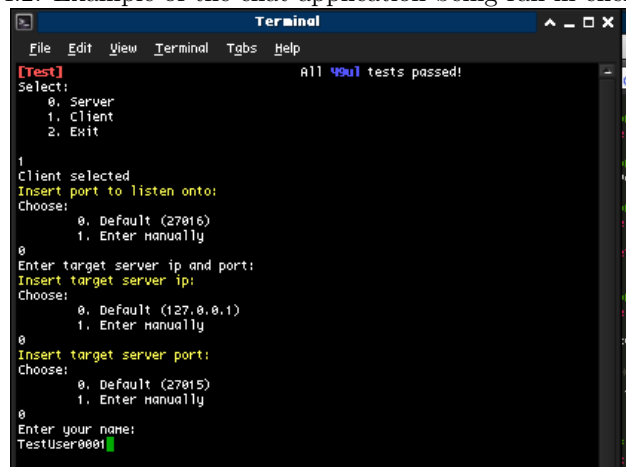
Information is displayed on sent/received data and client connections/disconnections.

4.4.2 Client role

When the application is running in client mode, after successfully connecting to the server, the user can interact in several ways.

- Messages can be created and sent to the server.
- A request to edit an existing message on the server can be sent.
- The current local state of the client can be displayed.

Figure 4.2: Example of the chat application being ran in client mode.



```
Terminal
File Edit View Terminal Tabs Help
[Test] All 4901 tests passed!
Select:
  0. Server
  1. Client
  2. Exit

1
Client selected
Insert port to listen onto:
Choose:
  0. Default (27016)
  1. Enter manually

0
Enter target server ip and port:
Insert target server ip:
Choose:
  0. Default (127.0.0.1)
  1. Enter manually

0
Insert target server port:
Choose:
  0. Default (27015)
  1. Enter manually

0
Enter your name:
TestUser0001
```

HOW TO GET, COSE, GITHUB, WEBSITE, DEPENDENCIES