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

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 it 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 papers 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.

```cpp
/// @brief Synchronizable data structure representing a chat message.
struct Message : syn::SyncObj
<
    // The fields types are passed as a variadic type list.

    int,            // messageID
    std::string,    // author
    std::string     // contents
>
{
    // Field proxies are generated via macros for convenience.

    SYN_PROXY(0, messageID);
    SYN_PROXY(1, author);
    SYN_PROXY(2, contents);
};
```

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

```cpp
/// @brief Synchronizable data structure representing a chat message.
SYN_DEFINE_SYNCOBJ
(
    // Name of data structure.
    Message,

    // Tuple of synchronizable fields.
    (
        (int, messageID),
        (std::string, author),
        (std::string, contents)
    )
);
```

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 managament*, *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.

```cpp
// SFML offers an `IpAddress` data structure that that can be initialized from strings.
using sf::IpAddress;

// SFML offers a `Packet` object that can be filled/emptied using stream operators.
using sf::Packet;

// SFML offers an `UdpSocket` class that properly manages and abstracts the memory
// and functionality of an UDP socket.
using sf::UdpSocket;
```

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

```cpp
int x{5};
std::string y{"hello!"};
double z{2.52};

sf::Packet toSend;
toSend >> x >> y >> z;

// Send packet through the UDP socket...

```

```cpp
int outX;
std::string outY;
double outZ;

sf::Packet received;
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.

```cpp
enum class CtoS : NType
{
    // Request to establish a connection to the server
    ConnectionRequest = 0,

    // Ping to avoid timing out with the server
    Ping = 1,

    // Request to sync data with server
    SyncRequest = 2,

    // Informs the server that the syncronization was successful
    SyncSatisfied = 3,

    // Custom data packet
    Data = 4,
};
```

StoC enumeration

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

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

    // Decline a client's requested connection
    ConnectionDecline = 1,

    // Satisfy sync request (if the client's revision is behind)
    SyncRequestSatisfy = 2,

    // Sync not needed (client has the same revision as the server)
    SyncRequestUnneeded = 3,

    // Sync request declined (overload/technical issue)
    SyncRequestDecline = 4,

    // Custom data packet
    Data = 5,
};
```

2.2.2 Packet handling

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

```
void handle(RPT mType)
{
    switch(mType)
    {
        case RPT::ConnectionAccept:
            handleConnectionAccept();
            return;
        case RPT::ConnectionDecline:
            handleConnectionDecline();
            return;
        // ...
    }
}
```

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.

```
// `SessionHost` is a CRTP template class, to allow compile-time generation
// of appropriate data structures.
template
<
    // `struct` containing global synchronization settings.
```

```cpp
    typename TSettings,

    // Type of sent packets.
    typename TSPT,

    // Type of received packets.
    typename TRPT,

    // CRTP (curiously recurring template pattern) derived type.
    template<typename> class TDerivedBase
>
class SessionHost
{
    // Typedefs of data structures that depend on the template parameters.
    public:
        // CRTP type.
        using Derived = TDerivedBase<Settings>;

        // Packet types.
        using SPT = TSPT;
        using RPT = TRPT;

        // Synchronization classes.
        using Settings = TSettings;
        using SyncManager = typename Settings::SyncManager;
        using Diff = typename SyncManager::DiffType;
        using Snapshot = typename SyncManager::SnapshotType;

    private:
        // Name of the host.
        std::string name;

        // IP address of the host.
        IpAddress ip;

        // Port the host is listening on.
        Port port;

        // UDP socket of the host.
        UdpSocket socket;

        // Status of the host.
        bool busy{false};

        // Tries binding the socket to the specified port.
        void tryBindSocket();

        // Tries forwarding the received packets to the underlying
        // client/server implementation.
        void tryForwardReceivedPacket();

        // Function running in the receive thread, which keeps track
        // of the data receiving buffer and of the socket state.
        void receiveThread();

    protected:
        // Buffers that store received data and data that will be sent.
        Packet sendBuffer, recvBuffer;

        // ‘void‘ future offering a RAII wrapper for the host’s thread.
        std::future<void> hostFuture;

        // IP of the sender of the last received data.
        IpAddress senderIp;

        // Port of the sender of the last received data.
        Port senderPort;

        // Synchronization manager.
        SyncManager syncManager;

        // Sends the content of the send buffer to the specified host.
        void sendTo(const IpAddress& mIp, const Port& mPort);

        // Builds a packet from a variadic argument list.
        template<SPT TType, typename... TArgs> void mkPacket(TArgs&&... mArgs);

        // Sets the state of the host.
        void setBusy(bool mBusy) noexcept { busy = mBusy; }

        // Pops data of type ‘T‘ from the receive buffer.
        template<typename T> auto popRecv();

    public:
        // Constructs the host using a specified name and port.
        // This function tries to bind the socket and start the dedicated thread.
        SessionHost(std::string mName, syn::Port mPort);

        // Getters.
        const auto& getName() const noexcept;
        const auto& getIp() const noexcept;
        const auto& getPort() const noexcept;
        const auto& isBusy() const noexcept;
        auto& getSyncManager() noexcept;
};

// Template typedef for the base ‘SessionHost‘ used by servers.
template<typename TSettings>
using SessionServerBase = SessionHost<TSettings, PT::StoC, PT::CtoS, SessionServer>;

// Template typedef for the base ‘SessionHost‘ used by clients.
template<typename TSettings>
```

```
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.

```
class ClientHandler
{
    private:
        // Connect client ID.
        // Assigned by the server.
        CID cid{nullCID};

        // IP and Port of the client.
        IpAddress ip;
        Port port;

        // Seconds until the client is timed out.
        int secondsUntilTimeout;

        // State of the client handler.
        bool busy{false};

    public:
        // Assumes the handler is not busy and binds it to a client with a
        // specific client ID, IP and Port.
        void bindToClient(CID mCID, const IpAddress& mIp, const Port& mPort);

        // Unbinds the handler from the client connection.
        void unbindFromClient();

        // Getters
        const auto& getCID() const noexcept;
        const auto& getIp() const noexcept;
        const auto& getPort() const noexcept;
};
```

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.

```cpp
class ClientHandlerManager
{
    private:
        // Next available unique client ID.
        CID nextCID{0};

        // Memory management and recycling data structure for `ClientHandler` instances.
        ssvu::MonoManager<ClientHandler> clientHandlers;

        // Vector of non-busy client handlers.
        std::vector<ClientHandler*> chAvailable

        // Vector of busy client handlers.
        std::vector<ClientHandler*> chBusy;

        // Map of client handlers, by client ID.
        std::map<CID, ClientHandler*> chMap;

        // `void` future that wraps the thread which checks timeouts for client handlers.
        std::future<void> timeoutFuture;

        // Vector of client handler IDs that need to be disconnected.
        std::vector<CID> toDisconnect;

        // State of the client handler manager.
        bool busy{true};

        // Creates a client handler instance from the `MonoManager`
        // and inserts it into the available vector.
        void createClientHandler();

        // Method ran in the timeout thread.
        // Periodically checks if clients have timed out.
        void runTimeout();

    public:
        // Accepts a new client connection.
        // Uses an existing free handler, if available, otherwise allocates a new one.
        // Also removes all handlers that have to be disconnected and updates the ID map.
        auto& acceptClient(const IpAddress& mIp, const Port& mPort);

        // Returns true if an handler exists, by ID.
        bool has(CID mCID) const noexcept;

        // Gets a client handler by ID.
        auto& operator[](CID mCID) noexcept;

        // Refreshes the timeout duration when a ping is received for a specific client.
        void pingReceived(CID mCID) noexcept;
};
```

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.

```cpp
template
<
    // Variadic type list of field types.
    typename... TArgs
>
class SyncObj : public Impl::ObjBase
{
    public:
        // Type of the tuple of fields.
        using TplFields = ssvu::Tpl<TArgs...>;

    private:
        // Count of fields.
        static constexpr SizeT fieldCount{sizeof...(TArgs)};

        // Tuple of fields.
        TplFields fields;

        // Dirty/clean bits for the fields.
        std::bitset<fieldCount> fieldFlags;

    public:
        // Type of the field at index ‘TI‘.
        template<TypeIdx TI> using TypeAt = ssvu::TplElem<TI, decltype(fields)>;

        // Type of the field proxy at index ‘TI‘.
        template<TypeIdx TI> using ProxyAt = FieldProxy<TI, SyncObj<TArgs...>>;

    private:
        // Non-const reference to the field at index `TI`.
        template<TypeIdx TI> auto& getFieldAt() noexcept;

        // Sets the bit at index `TI` to true.
        template<TypeIdx TI> void setBitAt() noexcept;

        // Sets the bit at index `mI` to false.
        void unsetBitAt(TypeIdx mI) noexcept;

    public:
        // Returns a proxy for the field at index `TI`.
        template<TypeIdx TI> auto get() noexcept;

        // Sets the object state from an `mX` json value.
        void setFromJson(const ssvj::Val& mX);

        // Serializes all the fields to json.
        auto toJsonAll();

        // Serializes all dirty fields to json.
        auto toJsonDirty();
};
```

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.

```
template
<
    // Index of the field in the tuple.
    TypeIdx TI,

    // Type of attached `SyncObj`.
    typename TObj
>
class FieldProxy
{
    private:
        // Reference to the attached object.
        TObj& syncObj;

    public:
        // Constructs the `FieldProxy` from a reference to its parent object.
        FieldProxy(TObj& mSyncObj) noexcept;

        // Returns a non-const reference to the underlying field, and sets the dirty bit.
        auto& edit() noexcept;

        // Returns a const reference to the underlying field.
        const auto& view() const noexcept;
};
```

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*).

```
// Class representing the snapshot of the entire state of the `SyncManager`.
template<typename TManager> struct Snapshot
{
    // Alive/dead bitset storage per object type.
    BitsetStorage bitsetIDs;

    // Tuple that stores a snapshot type data for every synchronizable type.
    ssvu::TplRepeat<TypeData, TManager::typeCount> typeDatas;

    // Serializes the snapshot to JSON.
    auto toJson() const;

    // Initializes the snapshot from JSON.
    void initFromJson(const ssvj::Val& mX);

    // Returns the difference between this snapshot and `mX`.
    auto getDiffWith(const Snapshot& mX);
};
```

3.2.1 SnapshotTypeData

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

```cpp
// Class representing the snapshot for a specific synchronizable type.
struct SnapshotTypeData
{
    // Items contained in the snapshot type data.
    std::map<ID, ssvj::Val> items;

    // Serializes the snapshot type data to JSON.
    auto toJson() const;

    // Initializes the snapshot type data from JSON.
    void initFromJson(const ssvj::Val& mX);
};
```

‘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.

```cpp
// Class representing a diff for all the types handled by the ‘SyncManager‘.
template<typename TManager> struct Diff
{
    // Tuple containing the diff type data for every synchronizable type.
    ssvu::TplRepeat<TypeData, TManager::typeCount> typeDatas;

    // Serializes the diff to a JSON value.
    auto toJson() const;

    // Initializes the diff from a JSON value.
    void initFromJson(const ssvj::Val& mX);

    // Returns true if there is no difference between snapshots.
    bool isEmpty() const noexcept;
};
```

3.3.1 DiffTypeData

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

```cpp
// Class representing the diff for a specific type managed by the ‘SyncManager‘.
struct DiffTypeData
{
    // Map of objects that have been created since last snapshot.
    std::map<ID, ssvj::Val> toCreate;

    // Map of objects that have been updated since last snapshot.
    std::map<ID, ssvj::Val> toUpdate;

    // Vector of objects that have been removed since last snapshot.
    std::vector<ID> toRemove;

    // Serializes the diff type data to JSON.
    auto toJson() const;

    // Initializes the data from JSON.
    void initFromJson(const ssvj::Val& mX);

    // Returns true if there are no differences.
    bool isEmpty() const noexcept;
};
```

‘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¿* 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*.

```cpp
// ‘LifetimeManager‘ specialization for ‘Message‘.
template<> struct LifetimeManager<Message>
{
    // Required typedef of the handle used by the ‘SyncManager‘.
    using Handle = Message*;

    // Required function that returns a null handle.
    Handle getNullHandle() noexcept;

    // Required function that creates an object and returns an handle to it.
    Handle create();

    // Required function that removes an object attached to a specific handle.
    void remove(Handle mHandle);

    // ...user data...
};
```

3.5 SyncManager

3.5.1 Lifetime managers tuple

3.5.2 Handle maps tuple

3.5.3 Object management functions

Chapter 4

Example application: synchronized chat

4.1 Data structures

4.1.1 Message

4.2 Lifetime manager specialization

4.2.1 Pointers as handles

4.2.2 create

4.2.3 remove

4.3 Application packet types

4.3.1 Server to client

4.3.2 Client to server

4.4 ConsoleSessionController

4.4.1 Server role

4.4.2 Client role