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.

```
/// @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.

```
/// @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.

```
// 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.

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

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

// Send packet through the UDP socket...

```

```
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.

```
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.

```
enum class StoC : NType
{
    // Accept a requested client connection, assigning the client a CID
    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.

ClientHandlerManager

2.3.2 SessionClient

The *SessionClient* CTRP derivation of *SessionHost*

Chapter 3

Synchronization module

3.1 ssvu

3.2 SerializationHelper

3.3 SyncObj

3.3.1 Fields

Field tuple

Field flags

FieldProxy

3.4 Snapshot

3.4.1 SnapshotTypeData

Item map

3.5 Diff

3.5.1 DiffTypeData

toCreate map

toUpdate map

toRemove vector

3.6 LifetimeManager

3.7 SyncManager

3.7.1 Lifetime managers tuple

3.7.2 Handle maps tuple

3.7.3 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