The Sock Ninja Utility Class for Socket Apps

March 2020

Michael Benjamin, mikerb@mit.edu
Department of Mechanical Engineering, CSAIL
MIT, Cambridge MA 02139

1 Overview

The SockNinja is a C++ utility class for facilitating (a) socket configuration managment and (b) buffered communications I/O to and from a configured socket. As the number of apps grows, with

a similar need for socket comms, the goal of this class is to ease app development. The idea is conveyed in Figure 1 below, where each app contains an instance of the SockNinja to handle the work of comms over a socket.

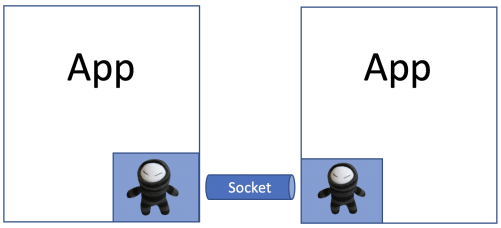

Figure 1: **The SockNinja Utility**: Applications needing to interface over a socket can use the SockNinja to facilitate socket configuration and I/O over the connected socket.

The SockNinja can be configured as either a client or server with a single configuration parameter, and the SockNinja Read/Write interface is the same in either client or server mode. This allows apps that use the SockNinja to also be easily configurable to act in a client or server mode.

This document describes the basic usage of the SockNinja targeting the application developer who may use this utility in their app.

1.1 Loose Relationship to MOOS-IvP

Although the SockNinja is used in MOOS apps, the SockNinja code has no dependency on MOOS, and may be used in any C++ application working with a socket. There are some string parsing utlities from the IvP codebase (from the `mbutils` library) used in the SockNinja. If a user wished to use the SockNinja outside the IvP context, a few utilities would need to be replaced. These functions will be listed later in the document.

1.2 Loose Relationship to NMEA

2 Using the Sock Ninja - A Simple Example

A simple example is given below to demonstrate a simple use case of the the SockNinja. This program may be invoked from the command line in either the client or server mode. It will loop until connected and then simply send a single message, looking for an incoming message, and then exit.

```
 1  #include <iostream>
 2  #include "MBUtils.h"
 3  #include "SockNinja.h"
 4
 5  using namespace std;
 6  int main(int argc, char *argv[])
 7  {
 8    if(argc != 3)
 9      return(1);
10    string ninja_type = argv[1];
11    string user_greet = argv[2];
12
13    SockNinja ninja(ninja_type, 29500);
14    ninja.setMsgFormatVerbatim();
15    while(!ninja.isConnected()) {
16      millipause(500);
17      cout << "Waiting to connect..." << endl;
18      ninja.setupConnection();
19    }
20
21    ninja.sendSockMessage(user_greet);
22    cout << "Reply:" << ninja.getSockMessage() << endl;
23  }
```

This program is compiled into the executable called `testninja`. It takes two arguments. The first is either `"client"` or `"server"`. The second argument is a message to send. To demonstrate, two instances are run, one as server and one as a client, each with a different message. Two terminals are needed. In the first terminal, launch with the `server` option chosen, and a simple message like `Hello`:

```
$ testninja server Hello
Waiting to connect...
Waiting to connect...
Waiting to connect...
Waiting to connect...
Waiting to connect...
```

```
Waiting to connect...
Reply:Bonjour              (after the second testninja is later launched)
```

The first terminal will run indefinitely until a client connects. The next step is to use a second terminal to launch a client version of `testninja`, providing another simple test message:

```
$ testninja client Bonjour
Waiting to connect...
Reply:Hello
```

The second launching of `testninja` completes almost immediately. It connects to the first version of the `testninja` allowing it to exit the while-loop in lines 15-19. After each has connected to one another, the greeting message is sent out, on line 21. The final step is to read the incoming message from the other and write to the terminal, on line 22.

2.1 SockNinja Message Format Options

In the example above, on line 14, the SockNinja is configured to send and receive messages in the *verbatim* format. There is no expectation of syntax, except newline characters are regarded as message separators. The string `"Hello\nWorld` would be regarded as two messages. The default message format mode is `"nmea"`, in which case incoming and outgoing messages are checked for proper NMEA syntax (but not semantics), as part of the read or written functions. Improperly formatted NMEA strings will be dropped but noted. More on NMEA format in Section 3.3

2.2 Sending Messages

Messages are sent with the `sendSockMessage(string)` function. One message per function call. A Boolean is returned, `true` if the message is sent, and `false` if the socket is unable to send the message or if the message is not a proper NMEA string if the message format mode is configured to be `"nmea"`

2.3 Receiving Messages

Messages are received with one of the two functions:

```
list<string> getSockMessages();
string getSockMessages();
```

In both cases, the SockNinja will read data from the socket and fill a list of received messages. The first function above retrieves all received messages. The second function only retrieves the most recent message. Both functions also clear the SockNinja's cache of received messages as part of the function call.

2.4 Imagining the SockNinja in a MOOS App

Note this simple program and the SockNinja have no dependency or relationship to MOOS, but it's easy to see how this can and will be used in MOOS apps.

- The SockNinja is a member variable of the MOOS app, with the client/server mode, port number and IP address, all configurable from the `.moos` file.
- In the `Iterate()` method, the SockNinja is repeatedly allowed to try to connect, until finally it succeeds.
- The MOOS app may register for mail to send out with the SockNinja over the open socket.
- The MOOS app may receive messages over the socket and publish the messages as mail Notifications to the local MOOSDB.

3 Configuring the Sock Ninja

The SockNinja has four primary configuration parameters: the client/server mode, the port number, the IP address, and the message handling mode.

3.1 Configuring the Client or Server Mode, and Port Number

The SockNinja can be configured as either a client or server, and this is declared at the time of construction through a constructor parameter.

```
SockNinja ninja("server");    // Configured as server
or
SockNinja ninja("client");    // Configured as client
or
SockNinja ninja();            // Configured as server
or
SockNinja ninja("blah");      // Configured as server
```

The default type is server, if not specified in the constructor, or if the argument is anything but `"client"`. The socket port number may be set in the constructor, or after instantiation:

```
SockNinja ninja("client", 29500);
or
SockNinja ninja("client");
bool ok_port = ninja.setPortNumber(29500);
```

The default port number is 29500. The `setPortNumber(int)` function will return `true` for any postitive integer, `false` otherwise.

3.2 Configuring the IP Address

When the SockNinja is configured as a client, the IP address may be configured to represent the address where the client should try to find the server. It is configured with:

```
SockNinja ninja("server");
bool ok_port = ninja.setPortNumber(29500);
bool ok_addr = ninja.setIPAddress(12.34.56.78);
```

The default IP address is `"localhost"`. The `setIPAddress(string)` function will return `false` if:

- The SockNinja type is ”server”, or
- The SockNinja client has already connected to a server, or
- The IP address is not a valid IPv4 address.

When the SockNinja is configured as *server*, the IP address cannot be set. However, for a server, the IP address will represent the address of the client when it connects to the server:

```
SockNinja ninja("server");
...
string ip = ninja.getIPAddress();   // IP Addr of connected client, null if not connected
```

```
SockNinja ninja("client");
...
string ip = ninja.getIPAddress();   // IP Addr to look for server
```

3.3 Configuring the Message Handling Mode

Message handling can be configured to operate in one of two modes. In the *verbatim* mode, no message syntax is checked for, and messages are separated by the newline (\n) character.

In the default, NMEA mode, incoming and outgoing messages are expected to be valid NMEA strings. A valid NMEA string is of the form:

```
$ABCDE,one,two,three,four,five*HH\r\n
```

The first character is a dollar-sign, followed by a five character header, followed by some number of comma-separated fields, terminated by an asterisk. The first two characters after the asterisk are a two-digit hex checksum of the first part of the message in between the dollar sign and asterisk. The NMEA message is terminated by CRLF, carriage-return and line-feed characters, ASCII values 13 and 10 respectively.

In the NMEA mode, the SockNinja will reject messages passed to `sendSockMessage(string)` if the string argument is not a valid NMEA message. Incoming messages read from the socket will also be checked for NMEA compliance or else rejected. More will be said about reading from the socket later in section **??**. The *semantics* of the message, e.g., the comma-separated content, are not checked for by the SockNinja. Semantics are application/user specfic and endless in variety. The SockNinja is intended to be application agnostic.

4 Internally Generated Status Messages

As the SockNinja is used, during configuration, connection and operation, it may generate event or warning messages as simple strings. The SockNinja maintains a list for each, finite in length, eventually dropping older messages for newer messages. The user of SockNinja has the option of retrieving these messages to use in whatever manner is appropriate for the application. There is no harm in *not* retrieving them, as the internal lists are bounded in size/memory. In the case of a MOOS app using a SockNinja, the messages are posted to the AppCast/terminal output, as discussed in Section 4.4.

4.1 Types of Status Messages: Events, Warnings, and Retractions

There are three types of messages:

- *Event* messages typically indicate changes of status, or things that should not be raised to the level of a warning. For example the successfuly connection to a socket.
- *Warning* messages are events that are intended to be brought to the attention of a user. For example a disconnected socket.
- *Retraction* messages are events that indicate a warning has been resolved. For example a reconnected socket, previously disconnected.

An example event:

```
Listening for connection on port 29500
```

An example warning:

```
Disconnected
```

An example retraction:

```
Disconnected
```

Retractions are strings designed to "match" a previous warning with an identical string value. This convention also happens to be consistent with how AppCasting apps handle warnings and retractions.

4.2 Retrieving the Status Messages

The SockNinja implements three methods for retrieving status messages:

```
list<string> getEvents();
list<string> getWarnings();
list<string> getRetractions();
```

In all three methods, the internal SockNinja list of status messages is cleared as a side effect of the function call. Presumably these functions are called on a regular basis. In MOOS apps they likely would be called in the `Iterate()` loop.

4.3 Using the Status Messages in a Vanilla C++ App

For a vanilla C++ app, the below is an example how this information may be retrieved and used:

```
SockNinja ninja("client");
. . .
list<string>::iterator p;
list<string> events = ninja.getEvents();
list<string> warnings = ninja.getWarnings();
list<string> retractions = ninja.getRetractions();

cout << "Events: " << endl;
for(p=events.begin(); p!=events.end(); p++)
  cout << "  " << *p << endl;

cout << "Warnings: " << endl;
for(p=warnings.begin(); p!=warnings.end(); p++)
  cout << "  " << *p << endl;

cout << "Retractions: " << endl;
for(p=retractions.begin(); p!=retractions.end(); p++)
  cout << "  " << *p << endl;
```

4.4 Using the Status Messages in an AppCasting MOOS App

In a MOOS app, the events, warnings, and retractions line up nicely with the features of AppCasting. The below snippet of code would likely be found at the end of the `Iterate()` loop:

```
// m_ninja is a SockNinja member variable of the MOOS App

list<string>::iterator p;
list<string> events      = m_ninja.getEvents();
list<string> warnings    = m_ninja.getWarnings();
list<string> retractions = m_ninja.getRetractions();

for(p=events.begin(); p!=events.end(); p++)
  reportEvent(*p);
for(p=warnings.begin(); p!=warnings.end(); p++)
  reportWarning(*p);
for(p=retractions.begin(); p!=retractions.end(); p++)
  reportRetraction(*p);
```

The `reportEvent()`, `reportWarning()`, and `reportRetraction()` methods are AppCastingMOOSApp methods that ensure this information is updated to the next-generated AppCast.

5 Obtaining Sock Ninja Config State and Statistics

The SockNinja supports an interface for obtaining the (a) the current state of configuration and connection, and (b) the total messages sent and received, along with the most recent of each.

5.1 Obtaining SockNinja Config and Connection State

The SockNinja config and connection state is comprised of the following pieces of information:

- *Ninja Type*: A string, either "server" or "Client"
- *Format*: A string, either "verbatim" or "nmea"
- *IP address*: A string holding a IPv4 formatted addres, e.g., 12.34.56.78
- *Port number*: An integer value, e.g., 29500
- *State*: A string, either "unconnected", "listening", or "connected".

The coresponding SockNinja functions are:

```
string SockNinja::getType();
string SockNinja::getFormat();
string SockNinja::getIPAddr();
string SockNinja::getState();
int    SockNinja::getPort();
bool   SockNinja::isConnected();  // convenience function
```

A further pair of convenience functions are provided for getting all five values. Information can be obtained in either a verbose or terse format. The terse format will return a single string.

```
  list<string> SockNinja::getSummaryStatComms(bool terse=false);
```

The non-terse mode will provide six strings:

```
Status:
  Type:   server
  Format: nmea
  State:  connected
  Port:   29500
  IPAddr: 12.34.56.78 (client)
```

The terse mode will provide a single string:

```
server/nmea/connected/29500/29500/12.34.56.78 (of client)
```

When the SockNinja is of type *server*, the IP address shown will be of the connected client if it is connected. When it is of type *client*, the IP address will be of the server to which is connected, or trying to connect to.

5.2 Obtaining SockNinja Send/Receive Stats

The SockNinja send/received stats are comprised of the following pieces of information:

- *Total Sent*: An unsigned int total for each message sent,
- *Total Received*: An unsigned int total for each message received,
- *Most Recent Sent*: A string containing the most recent message successfully sent.
- *Most Recent Received*: A string containing the most recent message successfully sent.

When operating in the NMEA mode, the SockNinja also stores this information *per message type*, keyed on the 5-digit NMEA header string. The SockNinja functions for getting this info are below. When a key is not provided, it will return the info related to all messages. When a key is provided and found, it will return the information related only to that key.

```
unsigned int SockNinja::getTotalMsgsSent(string key="");
unsigned int SockNinja::getTotalMsgsRcvd(string key="");
string SockNinja::getLastMsgSent(string key="");
string SockNinja::getLastMsgRcvd(string key="");
```

A further convenience function is provided for getting a summary of all incoming and outgoing messages, in the form of a list of strings with one function call:

```
  list<string> SockNinja::getSummaryStatMsgs();
```

This will provide a list of strings like the example below. The length of the list depends on how many message types exist.

```
NMEA sentences:
<--R      27  $PYDIR,45,55*57
 S-->     27  $CPNVG,134213.686,0000.00,N,00000.00,W,1,,,0,,,134213.686*64
 S-->     27  $CPRBS,134213.686,1,15.2,15.1,15.3,0*5B
 S-->     27  $GPRMC,134213.686,A,0000.00,N,00000.00,W,0,0,291263,0,E*67
```

The first column indicates whether the message was sent or received. The second column shows the number of messages for that message type, and the remainder of the string shows the most recent message for that type.