Tiny Message Server

(Version 0.50)

# Introduction

The TinyMessageServer (TMS) is a small footprint message server. (Idle, it uses something like 8 MB of heap.) It is written in Java and has a Java client base class for applications to extend. If this project proves useful and someone would like a C++ client, that wouldn’t be hard. If someone would like a Fortran client: the middle of the last century is calling, it would like its programming language back.

TMS has a publish and subscribe post-office paradigm. Clients subscribe to topics. All messages are sent to the server which then routes them to subscribers. There is no point-to-point communication.

Here is what TMS is not: *it is not intended as an enterprise message server*. It is not a replacement for *SmartSockets* or *ActiveMQ*. It has no redundancy or fault-tolerance. It is not secure. It is not encrypted. It does not reconnect to clients if it dies and is restarted. It is not intended for sustained high-bandwidth operation. It was written, selfishly, so that *ced* could communicate with other Java applications which could then send CLAS12 events, on-demand one-at-a-time, for display without the necessity of going through a file or a HIPO or ET ring. To accomplish this, the *ced* extends TMS server so that it can understand HIPO events—which requires including the *coatjava* jar. This manual, however, describes a completely stand-alone TMS which cannot send HIPO events. What kind of messages can be sent out-of-the-box? That is described in the Messages section below.

# GUI

TMS starts up a monitoring GUI. You get it whether you want it or not. Figure 1 is what the GUI looks like for a test server that has just started and has five clients connected. Along the top is a memory-usage strip chart and some information about the server. Below that is the log, and to the right of the log is a list of topics that clients have created. Below the log is a table with one row of information for each connected client.

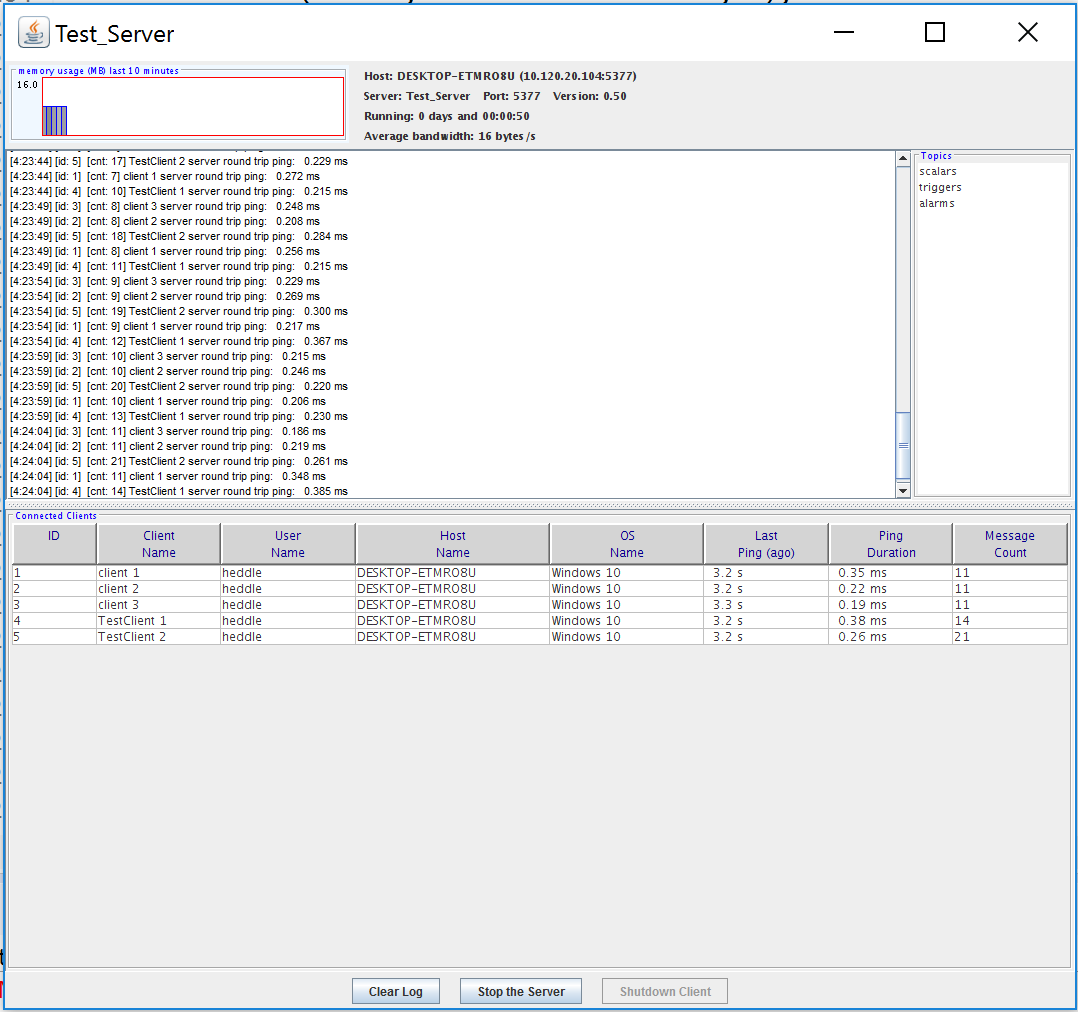


Figure 1. A server GUI just after starting the server, with five connected clients.

# Stress Test

TMS has been stress tested. One test client created a topic and then, as fast as possible, send 25,000 messages to other clients who subscribed. The messages contained arrays of 1000 long ints. The receivers verified the arrays. While running this test, the server achieved rates of 8MB/s, which was probably slowed by a print statement after each message was sent.

# Messages

TMS can send arrays of primitive types: bytes, shorts, ints, longs, floats, doubles, Strings. It can send Serializable objects (not recommended if you ever want a C++ client to connect to your server.) And it can send arbitrary mixed primitive types in the form of a “streamed” message.

Headers and Payloads

Messages have *headers* and *payloads*. Clients never have to deal with the headers directly, and can consider them as part of the message. The headers have a message type, a data type, a topic, a client Id, a tag and a length. They are described below.

## Message Types

Every message has a type. Clients never have to deal with them, but here they are:

CLIENT, HANDSHAKE, LOGOUT, PING, SERVERLOG, SHUTDOWN, SUBSCRIBE, UNSBSCRIBE

The reason that clients don’t have to deal with them is that all messages created with the correct type already in place. You never have to set the message type, it is only used internally. In case you want to know, all messages bound for other clients (i.e., sent to a topic) are CLIENT messages. The other message types are either created by the server or are consumed by the server.

## Message Data Types

Every message has a data type. Generally, clients don’t have to be aware of these either. Here they are:

NO\_DATA, BYTE\_ARRAY, SHORT\_ARRAY, INT\_ARRAY, LONG\_ARRAY, FLOAT\_ARRAY, DOUBLE\_ARRAY, STRING\_ARRAY, STRING, SERIALIZED\_OBJECT, STREAMED

For the most part clients will use convenience methods (described below) that will preclude the need for explicitly setting the data type.

## Message Topics

Message Topics are strings. There are no predefined topics, except “server” (which is reserved—clients can’t create a topic named “server”) and is used for administrative messages like pings and handshakes—messages that developers should not worry about. There will be examples of this later, but a topic is created the first time a client subscribes to it. The topics are not case sensitive, and in fact will be converted to lower case.

## Message Tag

Every message has an optional tag (a short int). The tag is not used internally. Clients can use it for any purpose, such as a sort of subtopic. Its default value is zero.

## Message Client Id

This is the ID of the client that created the message. Generally, (when using convenience methods to create messages) it is filled in automatically. It is not a source or destination ID, is a creator ID. It will contain the ID of the client that created and posted the message. This is only of use to the server—the other clients will have no use (that I can think of) for the numerical ID of the creator of the message.

Message Length

This is the length of the payload in bytes. It does not include the size of the header data. In general this is of little use, since the payloads are arrays and you can just ask them what their length is. It is of potential use for streamed messages (see below) when preparing space for the data. But even there if one is clever one can read all the data without knowing the amount ahead of time.

The Message class has obvious getters for all these fields: etdataLength()

Creating Messages

Adding the Payload

Streamed Messages

The Server

The Client

## Client ID

Every client gets a numerical ID as part of the validation process. Here is how it works. When the server is sniffed on its open port, it sends a message to the sniffer (a potential client) which includes an ID. It then gives the potential client a certain amount of time to respond with a validation message. If the potential client is not validated it is discarded. That is the extent of the TMS security.

Availablility

