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 (yet) 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, *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, out of the box, send HIPO events. What kind of messages can be sent? 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.

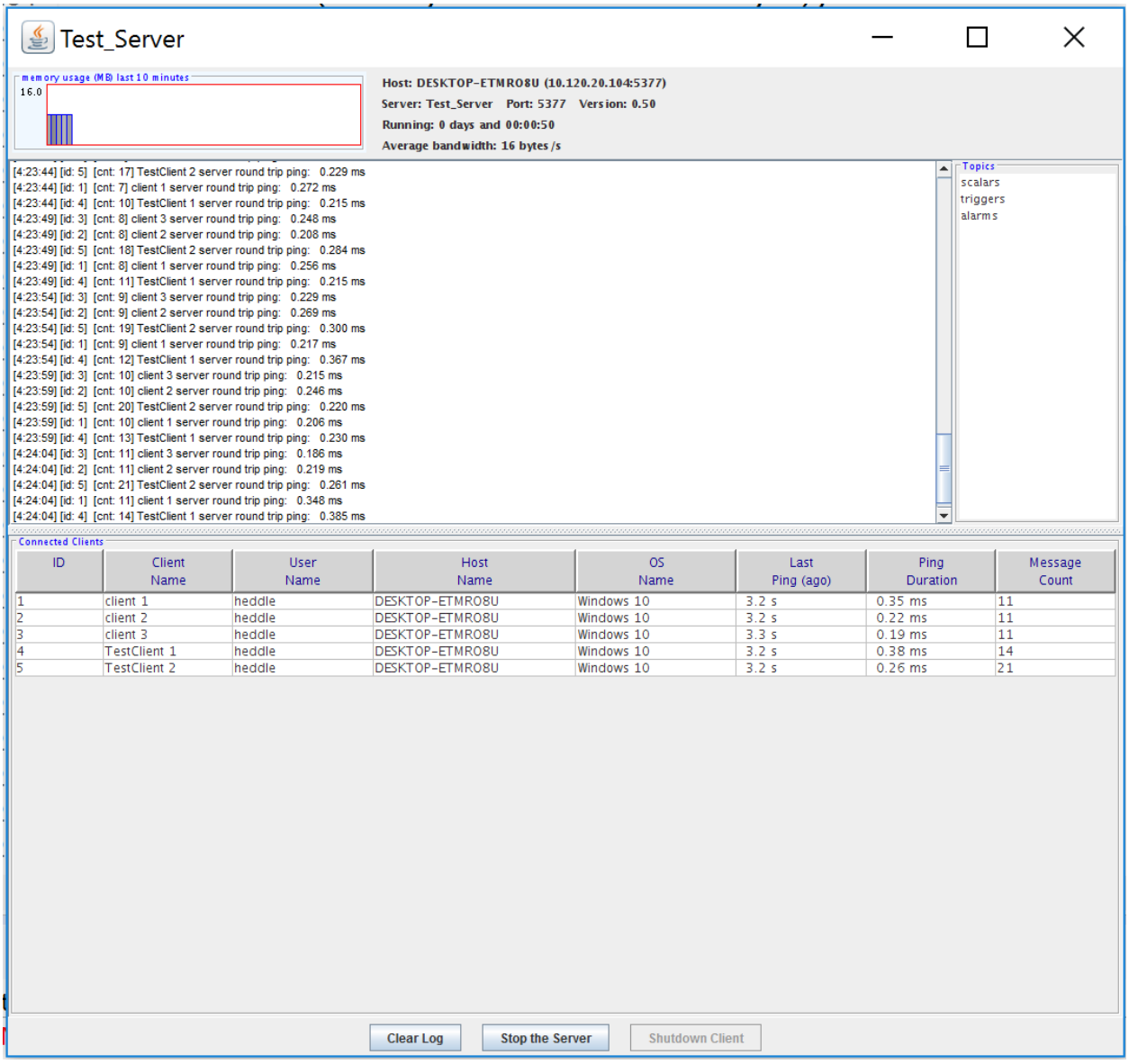


Figure 1. A server GUI just after starting the server, with five connected clients.

Stress Test

TMS has been stress tested. One test client created a topic and then, as fast as possible, sendt 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 used for administrative messages like pings and handshakes—messages that developers should not worry about. (And "server" is reserved—clients can't create a topic named "server." Really. Don't try.). 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. So "aLARms" is the same as "ALARmS".

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. A client can access its own ID (which was given by the server during validation) via its getId() method. The client ID in the message 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. They won't know who, say, client 32 is, and in any event (since there is no point to point messaging) have no way of communicating directly back to the creator client.

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 usually 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: getDataLength(), getTag(), etc.

Creating Messages

The general method for creating a CLIENT message is:

Message message = Message.createMessage(int clientID, short tag, String topic)

This creates a message the sender's clientID, a tag, and a topic.

This method has no payload. The next step is to set one.

Setting the Payload

After creating a message, the client will add a payload. In this section, we describe setting a payload for all data types except STREAMED (used for mixed data) which is described below.

For any array or String simple **call message.addPayload(object)** where **object** is the array or string. The method will set the data type based on the object type. So to set a byte array payload:

byte bytes[] = new byte[1000]
(fill in array)
message.setPayload(bytes)

Note that we are *setting* the payload, not adding to the payload. If you call two **setPayoad** methods, the first payload will be toast and the actual payload will be the result of the *second* **setPayload**; they are not added together.

There are convenience methods for creating payloads of single atomic types, such as setPayload(short) or setPayload(double). These will handle the mundane task of creating corresponding arrays of one element.

Streamed Messages

This is the most complicated data type, and is used for sending mixed types. It is not, out of the box, self-described. If the sender sent, in order, a short, a String, and an array of floats then the receiving clients have to know to read, in order a short, a String, and an array of floats. Now—a given application might develop a format string (e.g., "sS[f") to describe the data and send it first, so that clients can parse it and know how to read it—but TMS does not do that for you.

To send a STREAMED payload, the client creates a StreamedOutputPayload object

StreamedOutputPayload so = new StreamedOutputPayload();

The writes to it like a JAVA Data Stream, e.g.,

so.writeFloat(f);

so.write(byteArray);

When done writing *it is important to close the stream:*

so.close();

After that, just set the payload in the normal way:

message.setPayload(so);

The Server

The TMS server is started either by creating a **TinyMessageServer** object or by calling the main program in the **TinyMessageServer** class.

The most general constructor call is:

public TinyMessageServer(String name, int port)

This will attempt to start a server, giving it the name *name*, (the name is just a decoration; it can be anything) on the local machine which listens to client connections on the given port.

There is another constructor that doesn't take a port argument. It will try to start a server on a list of default ports: 5377, 2953, 3320, and 22724.

The main program in TinyMessageServer will attempt to start a server on a default port with the name "TinyMessageServer".

The Client

The base class for clients is the DefaultClient class. The most general constructor is

DefaultClient(String clientName, String hostName, int port)

The *clientName* is whatever name (just a decoration) you want to have associated with this particular client. The *hostName* is the name of the server (it can be an IP address) and *port* is the port the server is listening on. Using this constructor, the client has to know, a priori, the host name and port. For clients on the same local machine (common, since it is so hard to have open ports) there is a simpler constructor:

DefaultCliet(String clientName)

This looks for a TinyMesageServer using one of the default ports on the same local machine.

All real clients should extend the **DefaultClient** class.

They first thing they should do is override the initialize() method to subscribe to topics. The second thing they want to do is override processClientMessage(Message message) to deal with incoming CLIENT messages. Here is an example:

```
public class TestClient extends DefaultClient {

	/**
	 * Create a TestClient
	 * @throws BadSocketException
	 */
	public TestClient() throws BadSocketException {
		super("TestClient”);
	}

	/**
	 * This is a good place for clients to do custom initializations such as
	 * subscriptions. It is not necessary, but convenient. It is called at the
	 * end of the constructor.
	 */
	@Override
	public void initialize() {
		//subscribe to some topics
		subscribe("Scalars");
		subscribe("Triggers");
		subscribe("Alarms");



	@Override
	public void processClientMessage(Message message) {
		if (message != null) {
			switch (message.getDataType()) {

			case NO_DATA:
				break;

			case BYTE_ARRAY:
				byte[] barray = message.getByteArray();
				System.err.print("\nClient: " + getClientName() + " got byte array [");
				for (byte b : barray) {
					System.err.print(b + " ");
				}
				System.err.println("]");
				break;

			case SHORT_ARRAY:
				break;

			case INT_ARRAY:
				int[] iarray = message.getIntArray();
				System.err.print("\nClient: " + getClientName() + " got int array [");
				for (int i : iarray) {
					System.err.print(i + " ");
				}
				System.err.println("]");
				break;

			case LONG_ARRAY:
				long[] larray = message.getLongArray();
				int len = larray.length;
				System.err.print("\nClient: " + getClientName() + " got long array of len = " + len +
						" with 1st elem = " + larray[0] + " last elem = " + larray[len-1]);
				break;

			case FLOAT_ARRAY:
				break;

			case DOUBLE_ARRAY:
```

```
                        double[] darray = message.getDoubleArray();
                        System.err.print("\nClient: " + getClientName() + " got double array [");
                        for (double i : darray) {
                                System.err.print(i + " ");
                        }
                        System.err.println("]");
                        break;

                case STRING_ARRAY:
                        String[] sarray = message.getStringArray();
                        System.err.print("\nClient: " + getClientName() + " got string array [");
                        for (String s : sarray) {
                                System.err.print("\"" + s + "\" ");
                        }
                        System.err.println("]");
                        break;

                case STRING:
                        String s = message.getString();
                        System.err.print("\nClient: " + getClientName() + " got string \"" + s + "\"");
                        break;

                case SERIALIZED_OBJECT:
                        System.err.print("\nClient: " + getClientName() + " got serialized object\n");
                        Object obj = message.getSerializedObject();
                        System.err.println(obj.toString());
                        break;

                case STREAMED:
                        System.err.print("\nClient: " + getClientName() + " got streamed object\n");
                        StreamedInputPayload si = message.getStreamedInputPayload();
                        try {
                                System.err.println("Got byte: " + si.readByte());
                                System.err.println("Got double: " + si.readDouble());
                                System.err.println("Got long: " + si.readLong());
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                        break;

                }
        }
    }
}
```

Note that this is a pretty dumb **processClientMessage**, but it does demonstrate how to get the payload out of the message. A real **processClientMessage** might, for example, will certainly check the message topic and perhaps the message tag:

```
@Override
    public void processClientMessage(Message message) {
        if (message != null) {
            if (message.getTopic().equals(“scalars”) {
                if (message.getTag() == SCALY_SCALARS {
```

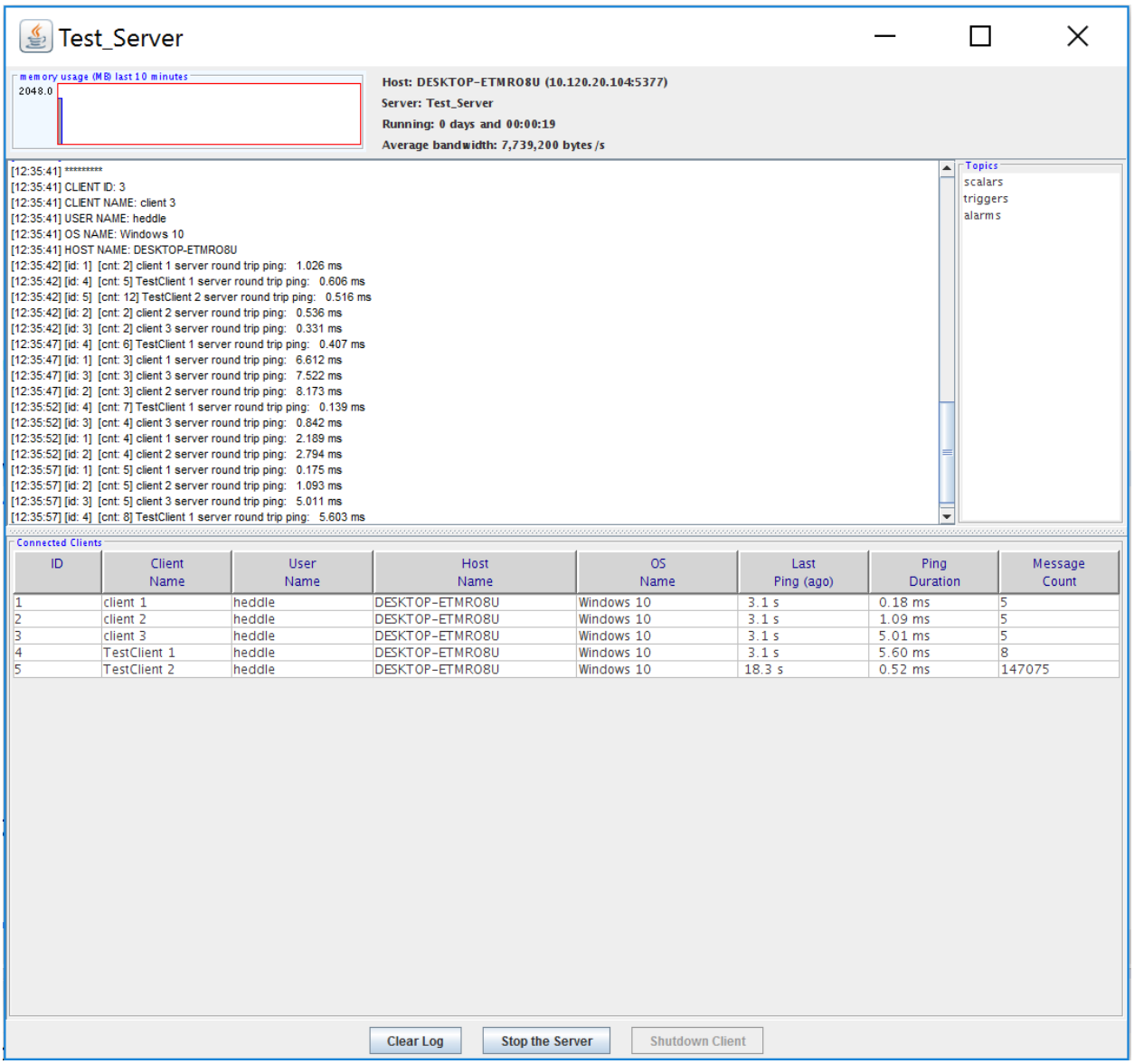
etc.

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

The TinyMessageServer source code is in the cnuphys area of the CLAS12 ofline git repository. If you just want the jar file and the documentation, it is available at https://userweb.jlab.org/~heddle/tms/builds/


Test_Server
memory usage (MB) last 10 minutes
2048.0
Host: DESKTOP-ETMRO8U (10.120.20.104:5377)
Server: Test_Server
Running: 0 days and 00:00:19
Average bandwidth: 7,739,200 bytes/s
[12:35:41] *********
[12:35:41] CLIENT ID: 3
[12:35:41] CLIENT NAME: client 3
[12:35:41] USER NAME: heddle
[12:35:41] OS NAME: Windows 10
[12:35:41] HOST NAME: DESKTOP-ETMRO8U
[12:35:42] [id: 1] [cnt: 2] client 1 server round trip ping: 1.026 ms
[12:35:42] [id: 4] [cnt: 5] TestClient 1 server round trip ping: 0.606 ms
[12:35:42] [id: 5] [cnt: 12] TestClient 2 server round trip ping: 0.516 ms
[12:35:42] [id: 2] [cnt: 2] client 2 server round trip ping: 0.536 ms
[12:35:42] [id: 3] [cnt: 2] client 3 server round trip ping: 0.331 ms
[12:35:47] [id: 4] [cnt: 6] TestClient 1 server round trip ping: 0.407 ms
[12:35:47] [id: 1] [cnt: 3] client 1 server round trip ping: 6.612 ms
[12:35:47] [id: 3] [cnt: 3] client 3 server round trip ping: 7.522 ms
[12:35:47] [id: 2] [cnt: 3] client 2 server round trip ping: 8.173 ms
[12:35:52] [id: 4] [cnt: 7] TestClient 1 server round trip ping: 0.139 ms
[12:35:52] [id: 3] [cnt: 4] client 3 server round trip ping: 0.842 ms
[12:35:52] [id: 1] [cnt: 4] client 1 server round trip ping: 2.189 ms
[12:35:52] [id: 2] [cnt: 4] client 2 server round trip ping: 2.794 ms
[12:35:57] [id: 1] [cnt: 5] client 1 server round trip ping: 0.175 ms
[12:35:57] [id: 2] [cnt: 5] client 2 server round trip ping: 1.093 ms
[12:35:57] [id: 3] [cnt: 5] client 3 server round trip ping: 5.011 ms
[12:35:57] [id: 4] [cnt: 8] TestClient 1 server round trip ping: 5.603 ms
Topics
scalars
triggers
alarms
Connected Clients
ID
Client Name
User Name
Host Name
OS Name
Last Ping (ago)
Ping Duration
Message Count
1 client 1 heddle DESKTOP-ETMRO8U Windows 10 3.1 s 0.18 ms 5
2 client 2 heddle DESKTOP-ETMRO8U Windows 10 3.1 s 1.09 ms 5
3 client 3 heddle DESKTOP-ETMRO8U Windows 10 3.1 s 5.01 ms 5
4 TestClient 1 heddle DESKTOP-ETMRO8U Windows 10 3.1 s 5.60 ms 8
5 TestClient 2 heddle DESKTOP-ETMRO8U Windows 10 18.3 s 0.52 ms 147075
Clear Log
Stop the Server
Shutdown Client