Vehicle Maintenance and Tracking System (VMTS)

John Newcomb
jnewcomb@gmu.edu

Abstract

Vehicle Maintenance and Tracking System (VMTS) is a vehicle monitoring and tracking system that enables users to track and monitor vehicle operating conditions, be alerted on configurable conditions, and edit and maintain vehicle information. The system is directed towards companies with fleet vehicles that need to monitor vehicle usage and operational alert conditions to ensure that vehicles are being operated safely and that vehicle error conditions are identified early so that damage to the vehicle can be minimized. The system can also be utilized by individual consumers that require detailed information about their vehicle usage as well, such as parents with teenage drivers. The system consists of a simulated Vehicle Monitoring Component (VMC), Vehicle Monitoring Component Server (VMC Server), Vehicle Maintenance and Tracking System Server (VMTS Server) and a database server. The VMC is installed on a vehicle and connects to the on-board computer system of the vehicle. It monitors vehicle events and sends information to the VMC Server based on configured event settings. The VMC Server receives VMC events and sends them to the VMTS Server and it monitors configuration requests from the VMTS Server and sends to the appropriate VMC. The VMTS Server handles VMC events as well as sends configuration updates to the VMC via the VMC Server. The VMTS Server will add the VMC events the database and forward event notifications to the owner based on configured settings.

1. Introduction

The idea for VMTS originated from the OnStar commercials touting the capability to deliver monthly vehicle diagnostic reports to owners via Email. This seems like a good idea on the surface, but realistically most normal vehicle owners do not monitor vehicle diagnostic information at a very detailed level. Also, if an error condition is detected it would not get distributed to the owner until the monthly diagnostic report is distributed, and the monitoring and tracking options are not configurable to the owner’s requirements. I used to be a mechanic for two airlines where I was responsible for the maintenance of the fleet of ground equipment and vehicles. I have always been interested in creating a system to manage and track vehicle operational conditions and system maintenance schedules.

Companies that operate a fleet of vehicles would benefit greatly from an on-board vehicle monitoring system. All of the vehicle’s operational conditions can be monitored and recorded. This will enable the company to track vehicle abuse and be notified quickly when there is some type of system error that requires attention. The system can also be used to track required maintenance based on maintenance schedules specific to the vehicle. Vehicle operators are less likely to abuse the vehicle when the vehicle is equipped with an on-board tracking system. Ultimately I believe the VMTS will promote safe vehicle usage and will reduce fleet operating expenses.

The VMTS project attempts to simulate the Vehicle Monitoring and Tracking System environment. My goal was to implement the components utilizing a variety of distributed computing technologies. The VMTS project consists of the following components:

- **Simulated Vehicle Monitoring Component (VMC)**
  The foundation of the system is the Vehicle Monitoring Component (VMC) that is installed on the vehicle and interfaces with to the on-board computer so that all vehicle conditions may be monitored. Each VMC is identified by a unique identifier and routable address. The communication mechanism is flexible to support Wi-Fi hotspots typical in large metropolitan areas, satellite or GSM networks.

- **Vehicle Monitoring Component Server (VMC Server)**
The VMC Server is the conduit between the VMCs and the VMTS Server. It receives VMC events and adds them to a MSMQ for the VMTS to process. It also receives VMC configuration events from the VMTS and is responsible for distributing them to the specified VMC.

- **Vehicle Monitoring and Tracking Servers (VMTS)**
  The VMTS Server reads VMC event data from the MSMQ and stores them in the database. It will also alert the owner based on pre-configured options. The VMTS Server also distributes VMC configuration data to the VMC Server to be applied to the specified VMC.

- **Vehicle Monitoring and Tracking Database Servers (VMTD)**
  The Vehicle Monitoring and Tracking System will utilize one or more independent database servers to store and maintain vehicle configuration information as well as monitoring information received from the Vehicle Monitoring Component.

- **Vehicle Information Warehouse (VIW) (Proposed)**
  The Vehicle Information Warehouse is a simulated service that provides vehicle service and other information to VMTS for the monitoring and tracking process.

![Diagram of Vehicle Monitoring and Tracking System]

**Figure 1 - Vehicle Maintenance and Tracking System**

The VMTS project was developed in C# and the Microsoft.NET 2.0 framework. The database is Microsoft SQL Server 2005. The distributed computing technologies implemented in the VMTS project include the following:

- Asynchronous sockets
- .NET Remoting
- Microsoft Message Queues (MSMQ)

### 2. Related Project Works

There are some similar products on the market, including the OnStar system mentioned previously. OnStar utilizes a satellite and integrated equipment in GM vehicles to implement the technology. The OnStar system is not configurable – the on-board system can not be configured to match specific monitoring settings. GuardMagic is a satellite vehicle guard and navigation system. It can track pre-configured parameters, such as vehicle mileage, speed, idle time and such. I have not seen a system that is a configurable to the level of what VMTS is.
3. Solutions/Analysis

3.1. Development Overview
My goal was to use a variety of distributed computing technologies to implement VMTS. I wanted to use synchronous and asynchronous sockets, .NET remoting, MSMQ and Web Services. I discovered that the synchronous sockets did not work well with the implementation of the system, so only asynchronous sockets are implemented. I did not have time to implement functionality utilizing web services.

The Solutions/Analysis section consists of summaries of communications between components, an overview of core technologies implemented in VMTS and a detailed description of the VMTS project components.

3.2. Communications Summary

3.2.1. VMC to VMC Server
Communication between VMC and the VMC Server consists of three types of vehicle events; errors, informational events, and monitored events which are sent using a sockets. The VMC Server will communicate with many VMC devices simultaneously, so it is critical that the events be received and handled as quickly as possible. An event is raised; the VMC creates a socket and opens a connection to the VMC Server. The VMC event is serialized and sent over the socket. The VMC Server receives the VMC event in an asynchronous manner, de-serializes the data and places it in the VMC Event MSMQ. The socket is closed after the data is sent.

If communication can not be established between the VMC and VMC Server the event is persisted so that it can be sent when the connection is restored.

3.2.2. VMC Server to VMC
Communications between the VMC Server and VMC consists of VMC configuration updates originating from the VMTS Server. The VMC Server receives VMC configuration data and attempts to send it to the VMC via a socket. If the VMC is not reachable, the VMC configuration data is persisted in the VMC Configuration MSMQ where a background process attempts to deliver at a configurable interval. If the VMC is reachable, the VMC Server creates a socket and opens a connection to the specific VMC, serializes the VMC Configuration data and sends it over the socket. The VMC receives the data asynchronously, de-serializes it and applies the new configuration. The socket is closed by the VMC Server once the send is complete.

3.2.3. VMC Server to VMTS Server
There is no direct communication between the VMC Server and VMTS Server. Data received from the VMC Server is placed in the VMC Event MSMQ. The VMTS Server periodically polls the VMC Event MSMQ and inserts the data to the database.

3.2.4. VMTS Server to VMC Server
Communications between the VMTS server and VMC Server consists of VMC Configuration update data and is implemented using .NET Remoting. The .NET Remoting is configured to use a server-activated singleton object and a TCP channel.

The VMC Server receives the VMC configuration data by a method call on a remotable object. When a message is received the VMC Server attempts to send to the specific VMC. If a connection can not be made the data is added to the VMC Configuration Queue where a background process will attempt to deliver when the connection can be made.
3.2.5. VMTS Server to VMTS Database Server

The VMTS Server polls the VMC Event MSMQ periodically for messages. The VMTS Server receives messages from the VMC Event MSMQ and inserts the data into the database.

3.3. Core Technology Overview

The following is a detailed description of the core technology implemented in the VMTS project.

3.3.1. Sockets

The client socket code in VMC and VMC Server is very similar. The only difference is in the object that is serialized and sent over the socket. The sending logic is pretty straight forward; the socket is created, the data to be sent is serialized into a byte array and is sent over the socket, and then the socket is closed. See the Client socket code snippet in Appendix A.

The server socket asynchronously handles client connections. The server socket code fires a new thread to listen for socket connections. When a connection is established it is asynchronously handled calling the BeginAccept method, which creates a new event handler for the connection. The event handler for the connection calls the EndAccept method, which returns the data socket. A state object is to persist the data thread and data read from the socket. The BeginReceive method is called, which creates a new event handler to receive the data from the socket, passing in the state object. The receive event handler reads data from the socket and de-serializes it in the proper object. See the Server socket code snippet in Appendix A.

3.3.2. .NET Remoting

.NET Remoting is a technology provided in the .NET Framework to build distributed applications. To use .NET Remoting in an application you need a remotable object, a host application to listen to request for the object, and a client application that makes requests for the object. The remotable object runs on the server and accepts method calls from the client. The remotable object can be implemented either as a Server-activated object or a Client-activated object. The server-activated object’s instantiation is controlled by the server and can be implemented as either a Singleton or SingleCall type. With Singleton objects, one instance serves the requests of all clients in a multi-threaded manner. With SingleCall objects, a new object is created for each client request and is destroyed afterwards. Client-activated objects are created on the server for each client request and persist until the object is destroyed.

Messages are transported between applications by Channels. Two types of channels exist; HttpChannel and TcpChannel.

For the VMTS project the VMC Server is the host application, the VMTS Server is the client application and the remotable object is the VMC Configuration object, which is implemented as a Server-activated, Singleton object. I had to use a TcpChannel because the VMC Configuration object contained data types that could not be serialized with the HttpChannel. See Appendix B for the remoting code snippets.

3.3.3. VMTS Project Components

The main components of the VMTS project are the VMC, VMC Server and VMTS Server. The following is a brief description of the VMTS C# projects and objects (Reference Appendix C for the VMTS class diagrams):
**VMC.Common**
VMC.Common is a class library that contains common objects utilized by other projects, specifically the VMCMessage and VMCVehicle objects that are passed between all components – either via sockets or remoting.

**VMTS.Common**
VMTS.Common is a class library that contains the interface for the VMCUpdater remotable object used by the VMC Server and VMTS Server

**VMC Projects**
**VMC.Client**
VMC.Client is a class library that implements the VMC functionality. It is implemented as a class library so that it can be encapsulated in any type of application – windows server, windows application or console application.

**VMC.Transport**
VMC.Transport is a class library that contains IVMCTransport interface, which defines methods and events required for the transport implementation. The interface allows for different types of transports to be defined and implemented in order to send data to the VMC Server.

**VMC.SocketTransport**
VMC.SocketTransport is a class library that contains the socket implementation of the IVMCTransport interface.

**VMC.Client.GUI**
VMC.Client.GUI is a windows application that provides the user interface for the VMC.

**VMC Server**
**VMC.Server**
VMC.Server is a class library that implements the VMC Server functionality. It is implemented as a class library so that it can be encapsulated in any type of application – windows server, windows application or console application.

The VMC.Server contains separate object that handle communication from the VMC and VMTS Server independently.

**VMC.Server.GUI**
VMC.Server.GUI is a windows application that provides the user interface for the VMC Server.

**VMTS Server**
**VMTS.Server**
VMTS.Server is a class library that implements the VMTS Server functionality. It is implemented as a class library so that it can be encapsulated in any type of application – windows server, windows application or console application.

**VMTS.Server.GUI**
VMTS.Server.GUI is a windows application that provides the user interface for the VMC Server.
Figure 2 - VMTS Database Schema
4. Issues Encountered and Resolutions

Socket Communications – most socket examples show sending string data, which is pretty simple to implement. I wanted to send object instances over, which was a little more difficult to do. The object must be Serializable because it is serialized to a byte array in order to send over the socket. Sending object is hard to determine if the full object was received when implementing the asynchronous socket.

I had to implement a lot of threading to ensure that the application was responsive, and I had to be careful to ensure that all worker threads were stopped when shutting application down. The application would not shut down otherwise.

MSMQ – The MSMQ implementation in VMTS does not totally fit the usage of MSMQ. I used it as a reliable data store for vehicle update messages from the VMC Server to the VMC. There is a background thread that tries to deliver messages that originally failed, and if they can’t go they stay on the queue. The problem is that the methods available to walk the queue do not work well with this implementation. The Receive method blocks until a message arrives on the queue. When one is available, it is pulled off of the queue. If the message delivery fails it is put back on the queue, which will cause it to be read again. I wanted to walk all items in the queue one time and then sleep for some period of time. To resolve the issue I had to receive all messages at the beginning, and then walk them and put back on the queue if the delivery fails.

5. Summary

The VMTS project was a lot of fun to work on and I learned a lot, especially on sockets and MSMQ. I have worked with .NET Remoting quite a bit in the past and this project confirmed my belief that .NET Remoting is much easier to work with and is a very powerful tool for implementing distributed processing.

I did not get as much done on the VMC program as I would have liked. I wanted to get the events wired to automatically fire the error, information and monitored events. I also didn’t get a chance to implement any of the web services functionality.
Appendix A Socket Examples

Client Socket Code

```csharp
// Sends VMC Events to the VMC Server
public void Send(VMTS.VMC.Common.VMCMessage message)
{
    try {
        // Create a TCP/IP socket.
        vmcServerSocket.Connect(_VMCServerEndPoint);

        // Encode the data string into a byte array.
        System.IO.MemoryStream ms = new System.IO.MemoryStream();
        bf.Serialize(ms, message);
        byte[] data = ms.GetBuffer();

        // Send the data through the socket.
        int bytesSent = vmcServerSocket.Send(data);

        // Shutdown socket
        vmcServerSocket.Shutdown(SocketShutdown.Both);
        vmcServerSocket.Close();
    } catch (Exception ex) {
        Console.WriteLine(ex.Message); // TODO: Error handling
    }
}
```

Server Socket Code

```csharp
// Receive Updates from VMC Server
private void StartListener(object obj) {
    IPEndPoint localEndPoint = (IPEndPoint)obj;

    // Create a TCP/IP socket.
    Socket listener = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

    // Bind the socket to the local endpoint and listen for incoming connections.
    try {
        listener.Bind(localEndPoint);
        listener.Listen(5); // Place in listening state, allow backlog of 100 connections
        while (!_stop) {
            // Set the event to nonsignaled state.
            _receiveEvent.Reset();

            // Begin async operation to accept connection
        }
    }
```
listener.BeginAccept(new AsyncCallback(AcceptCallback), listener);

    // Wait until a connection is made before continuing, timeout of 1 second
    _receiveEvent.WaitOne();
}

listener.Close(); //close down socket listener
listener = null;
}
catch (Exception e)
{
    Console.WriteLine(e.ToString()); //TODO: error handling
}

private void AcceptCallback(IAsyncResult ar)
{
    if (!_stop) return;

    // Signal the main thread to continue.
    _receiveEvent.Set();

    Socket listener = (Socket)ar.AsyncState; // Get the socket that handles the client request.
    Socket handler = listener.EndAccept(ar); // returns data socket
    StateObject state = new StateObject(); // Create the state object to persist socket info
    state.workSocket = handler;
    //Start async receive of data
    handler.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0, new AsyncCallback(ReadCallback), state);
}

private void ReadCallback(IAsyncResult ar)
{
    // Retrieve the state object and the handler socket from the asynchronous state object.
    StateObject state = (StateObject)ar.AsyncState;
    Socket handler = state.workSocket;

    int bytesRead = handler.EndReceive(ar); // Read data from the client socket.
    if (bytesRead < state.buffer.Length)
    {
        //All data has been read. Create message object
        System.IO.MemoryStream ms = new System.IO.MemoryStream(state.buffer, 0, bytesRead);
        VMTS.Common.VMCVehicle vehicle = (VMTS.Common.VMCVehicle)bf.Deserialize(ms);
        ms.Close();

        //Raise event that data was received
        if (VehicleInfoUpdated != null)
            VehicleInfoUpdated(this, vehicle);
        handler.Shutdown(SocketShutdown.Both);
        handler.Close();
    }
state = null;
}
else
{
    // There might be more data, so store the data received so far.
    state.data.Add(state.buffer);

    // Not all data received. Get more.
    handler.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0,
                        new AsyncCallback(ReadCallback), state);
}

Appendix B .NET Remoting Samples

Remotable Object
The Server and Client applications reference a common library that contains the interface for the remotable object. The interface is as follows:

```csharp
public interface IVMCUpdater
{
    void UpdateVMC(VMCVehicle vehicle);
}
```

Server
The Server creates an object that implements the interface. The following code sets remoting for the Server application:

```
_channel = new TcpChannel(port);       //creates new channel for a specified port
ChannelServices.RegisterChannel(_channel, false);  //the channel is registered

//this publishes _update manager, which is the implementation of the IVMCUpdate interface.  VMCUpdateManager.rem is the URI for the remotable object
RemotingServices.Marshal(_updateManger, "VMCUpdateManager.rem");
```

Client
The following code creates a remotable object instance on the client:

```
TcpChannel chnl = new TcpChannel();
ChannelServices.RegisterChannel(chnl, false);

string uri = "tcp://" + host + ":" + port.ToString() + "/VMCUpdateManager.rem";
_vmcmCUpdator = (IVMCUpdater)Activator.GetObject(typeof(IVMCUpdater), uri);
```
## Appendix C VMTS Class Diagrams

### VMC.Common

<table>
<thead>
<tr>
<th>VMCMonitor</th>
<th>VMCVehicle</th>
</tr>
</thead>
<tbody>
<tr>
<td>_monitorID : Guid</td>
<td>_vehicleID : Guid</td>
</tr>
<tr>
<td>_name : string</td>
<td>_vmcID : Guid</td>
</tr>
<tr>
<td>_type : string</td>
<td>_host : string</td>
</tr>
<tr>
<td>_condition : string</td>
<td>_port : int</td>
</tr>
<tr>
<td>_value : string</td>
<td>_pollInterval : double</td>
</tr>
<tr>
<td>_unit : string</td>
<td>_enabled : bool</td>
</tr>
<tr>
<td>+MonitorID() : Guid</td>
<td>_vin : string</td>
</tr>
<tr>
<td>+Name() : string</td>
<td>_make : string</td>
</tr>
<tr>
<td>+Type() : string</td>
<td>_model : string</td>
</tr>
<tr>
<td>+Condition() : string</td>
<td>_year : string</td>
</tr>
<tr>
<td>+Value() : string</td>
<td>_fuelCapacity : double</td>
</tr>
<tr>
<td>+Unit() : string</td>
<td>_redline : short</td>
</tr>
<tr>
<td>+VMCMonitor()</td>
<td>_monitors : List&lt;VMTS.VMC.Common.VMCMonitor&gt;</td>
</tr>
<tr>
<td>+VMCMonitor(in id : Guid, in name : string, in type : string, in condition : string, in value : string, in unit : string)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VMCMonitor</th>
<th>VMCVehicle</th>
</tr>
</thead>
<tbody>
<tr>
<td>_id : Guid</td>
<td>_vehicleID : Guid</td>
</tr>
<tr>
<td>_date : DateTime</td>
<td>_vmcID : Guid</td>
</tr>
<tr>
<td>_event : string</td>
<td>_host : string</td>
</tr>
<tr>
<td>_value : object</td>
<td>_port : int</td>
</tr>
<tr>
<td>+Id() : Guid</td>
<td>_pollInterval : double</td>
</tr>
<tr>
<td>+MessageDate() : DateTime</td>
<td>_enabled : bool</td>
</tr>
<tr>
<td>+EventName() : string</td>
<td>_vin : string</td>
</tr>
<tr>
<td>+Value() : string</td>
<td>_make : string</td>
</tr>
<tr>
<td>+VMCMonitor()</td>
<td>_model : string</td>
</tr>
<tr>
<td>+VMCMonitor(in id : Guid, in name : string, in type : string, in condition : string, in value : string, in unit : string)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VMCMMessage</th>
<th>VMCVehicle</th>
</tr>
</thead>
<tbody>
<tr>
<td>_id : Guid</td>
<td>_vehicleID : Guid</td>
</tr>
<tr>
<td>_date : DateTime</td>
<td>_vmcID : Guid</td>
</tr>
<tr>
<td>_event : string</td>
<td>_host : string</td>
</tr>
<tr>
<td>_value : object</td>
<td>_port : int</td>
</tr>
<tr>
<td>+Id() : Guid</td>
<td>_pollInterval : double</td>
</tr>
<tr>
<td>+MessageDate() : DateTime</td>
<td>_enabled : bool</td>
</tr>
<tr>
<td>+EventName() : string</td>
<td>_vin : string</td>
</tr>
<tr>
<td>+Value() : string</td>
<td>_make : string</td>
</tr>
<tr>
<td>+VMCMMessage()</td>
<td>_model : string</td>
</tr>
<tr>
<td>+VMCMMessage(in id : Guid, in name : string, in type : string, in condition : string, in value : string, in unit : string)</td>
<td></td>
</tr>
</tbody>
</table>

### VMTS.Common

<table>
<thead>
<tr>
<th>VMCUpdater</th>
<th>VMCVehicle</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>_vehicleID : Guid</td>
</tr>
<tr>
<td></td>
<td>_vmcID : Guid</td>
</tr>
<tr>
<td></td>
<td>_host : string</td>
</tr>
<tr>
<td></td>
<td>_port : int</td>
</tr>
<tr>
<td></td>
<td>_pollInterval : double</td>
</tr>
<tr>
<td></td>
<td>_enabled : bool</td>
</tr>
<tr>
<td></td>
<td>_vin : string</td>
</tr>
<tr>
<td></td>
<td>_make : string</td>
</tr>
<tr>
<td></td>
<td>_model : string</td>
</tr>
<tr>
<td></td>
<td>_year : string</td>
</tr>
<tr>
<td></td>
<td>_fuelCapacity : double</td>
</tr>
<tr>
<td></td>
<td>_redline : short</td>
</tr>
<tr>
<td></td>
<td>_monitors : List&lt;VMTS.VMC.Common.VMCMonitor&gt;</td>
</tr>
<tr>
<td></td>
<td>+VMCMonitor()</td>
</tr>
</tbody>
</table>
### VMTS Server

<table>
<thead>
<tr>
<th>Server: VMTSServer</th>
</tr>
</thead>
<tbody>
<tr>
<td>__vmcUpdater : IVMCUpdater</td>
</tr>
<tr>
<td>+Start(in properties : Hashtable)</td>
</tr>
<tr>
<td>+Stop()</td>
</tr>
<tr>
<td>+UpdateVehicle(in vehicle : VMCVehicle)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GUI: frmVMTSServer</th>
</tr>
</thead>
<tbody>
<tr>
<td>__vmtServer : VMTSServer</td>
</tr>
<tr>
<td>+frmVMTSServer()</td>
</tr>
<tr>
<td>+cmdTest_Click(in sender : object, in e : EventArgs)</td>
</tr>
<tr>
<td>-frmVMTSServer_Load(in sender : object, in e : EventArgs)</td>
</tr>
<tr>
<td>-cmdStart_Click(in sender : object, in e : EventArgs)</td>
</tr>
<tr>
<td>-cmdUpdate_Click(in sender : object, in e : EventArgs)</td>
</tr>
</tbody>
</table>