Integrating the Target Workflow System (TWS) with the Command and Control Personal Computer (C2PC) System: Proof of Concept

28 August 2003

Edward C. Parks, 21936

Sponsor:	PEO(W)	**Contract No.:**	DAAB07-98-C-C201
Dept. No.:	W407	**Project No.:**	V17A

MITRE

Washington C3 Center

McLean, Virginia

Integrating the Target Workflow System (TWS) with the Command and Control Personal Computer (C2PC) System: Proof of Concept

ABSTRACT:

Under the sponsorship of OUSD AT&L and various Service[1] program managers, Joint Time Sensitive Target (TST) Simulation Experiments (SIMEXs) over the last two years have provided insight into how various systems and applications contribute to the TST development process. A fundamental problem with the targeting process is the manual entry of targeting information as the data is passed between systems used to prosecute TSTs. In many cases, the target identification/numbering formats are inconsistent. Developing multiple independent targets with a variety of sensors can exceed Command and Control (C2) user's capabilities. The Targeting Workflow System (TWS) architecture was developed to provide a framework for various targeting systems to create, update, and share target data with one another without human data entry and with a common numbering scheme.

As part of the Family of Interoperable Operational Pictures (FIOP) Management Plan[2], the FY02 – FY03 priorities included the development of a tactical Defense Information Infrastructure Common Operating Environment (DII COE) workstation. The FIOP System Engineering Working Group (SEWG) identified one of its initial objectives of creating a tactical workstation software baseline from the COE Common Operational Picture (COP). However, the software was shown to be unworkable for use in a wide area network environment. An alternate approach was to make the Marines' C2 Personal Computer (C2PC) software a COE mission application, establish a Joint Configuration Control Board for the software, and assign the Marines as Executive Agent over the software baseline and assign DISA responsibility for distributing the software to the Joint community.[3]

The goal of this "Proof of Concept" is to integrate TWS functionality with C2PC to provide a more robust methodology for developing, monitoring, and executing TSTs under a single C2 application leveraging the messaging and map capabilities.

[1] ONR31, NAVAIR PMA281 and SPAWAR PMW157 sponsor technical development and integration in the Strike Cell Laboratory in MITRE McLean, VA. ESC and AFC2ISRC sponsor similar activities in the Software Interoperability Facility for Time Critical Targeting at Hanscom AFB.
[2] Family of Interoperable Operational Pictures (FIOP) Management Plan FY 2002, 17 December 2001.
[3] White Paper Describing FIOP Approach to Systems Engineering under Task 2, Jesse Pirocchi, 26 February 2003

I. Description of TWS

Multiple paradigms have been developed to depict Time Sensitive Targeting (TST). A targeteer collects one piece of data about a target from one system, a second piece from another one, etc. Then the targeteer must "fit together" all the various pieces into a comprehensive picture of a specific target. Only then can a decision be made to strike the target or not. The key steps of collecting, analyzing, and making recommendations on a target are required for the target development process. Without accurate and timely acquisition, organization, and usage of relevant target data, a target strike may come too late or at an incorrect location. This process (Figure 1) is exasperated by the fact that the various systems involved in developing a time sensitive target do not communicate in a common format. As human intervention is required to pass information from one system to the next, errors can degrade or impede the target development process.

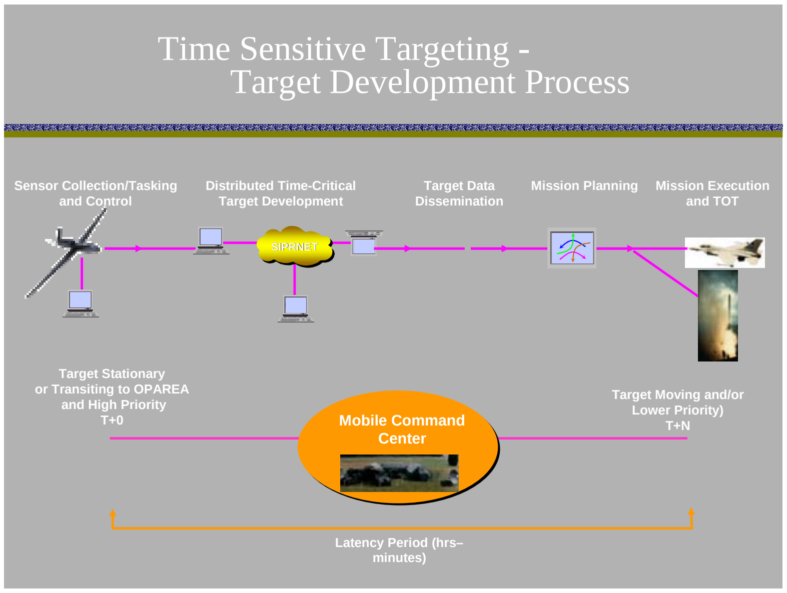

Time Sensitive Targeting -
Target Development Process
Sensor Collection/Tasking and Control
Distributed Time-Critical Target Development
Target Data Dissemination
Mission Planning
Mission Execution and TOT
SIPRNET
Target Stationary or Transiting to OPAREA and High Priority T+0
Mobile Command Center
Target Moving and/or Lower Priority) T+N
Latency Period (hrs–minutes)

Figure 1. Target Development Timeline for a TST

The TWS is both a server and clients designed to support workflow management, single target referencing ID, and automated data collection. The server will manage and distribute track/object/target data, as well as ingest Moving Target Indicator (MTI) data via a socket interface to Joint Service Work Station (JSWS) Sybase databases at each site. Using Structured Query Language (SQL) and messaging via TCP/IP sockets, TWS clients allow users to create new targets or objects of interest, modify targets, designate completion of various tasks for particular targets, and identify location of imagery or other relevant information. Clients provide updated information to the server, which then distributes the updates via sockets to all connected clients. TWS clients have complete visibility and access to all targets created with TWS, though ownership of targets can be transferred at will. TWS is designed to provide at glance information on all targets. Targets may be visually filtered in a variety of ways. The TWS client is a Java applet accessible via a browser with the appropriate Java plug-in. Upon executing the applet, users logon and all data managed by the TWS server is broadcast to clients as it becomes available. The TWS architecture is depicted in Figure 2.

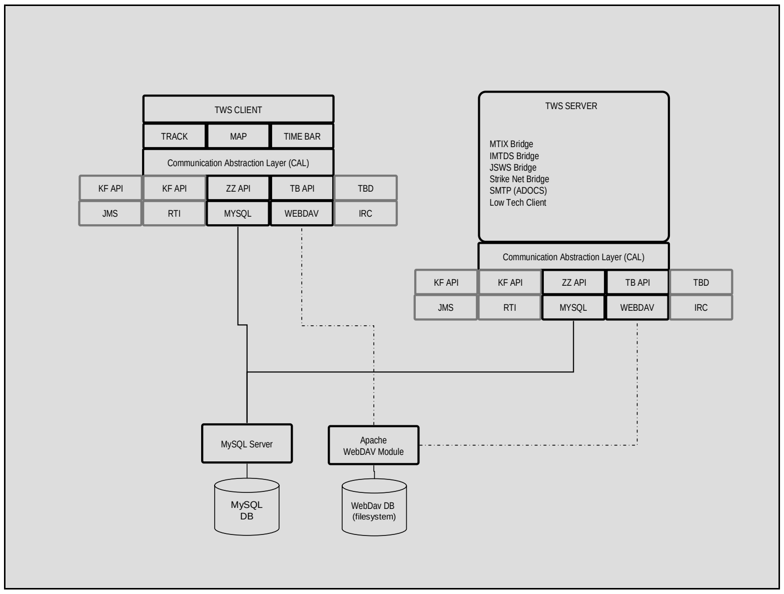


Figure 2: Target Workflow System Architecture

The TWS Server communicates with TWS clients and system interfaces via database SQL, WebDAV[4] or HTTP get/post methods. User event messages and mover/target update information messages will be shared software applications connected to this network. Current port for the database is 3306 and 80 for HTTP. Typical interfaces are depicted in Figure 3 and the message format is listed in Table 1.

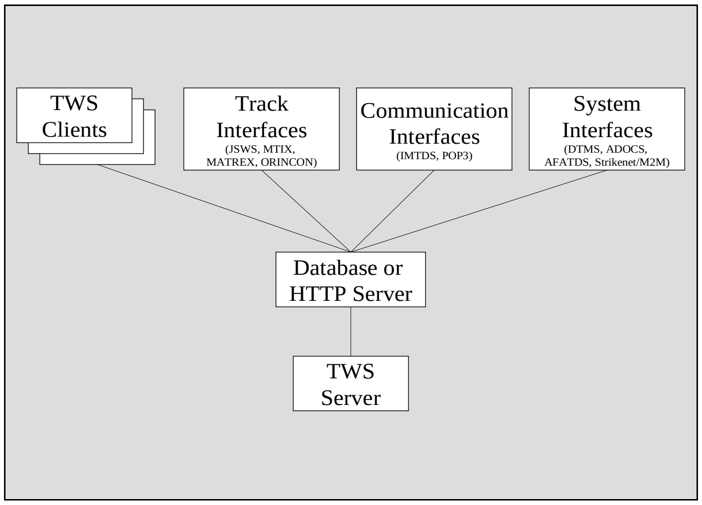

TWS Clients
Track Interfaces (JSWS, MTIX, MATREX, ORINCON)
Communication Interfaces (IMTDS, POP3)
System Interfaces (DTMS, ADOCS, AFATDS, Strikenet/M2M)
Database or HTTP Server
TWS Server

Figure 3. TWS Data Flow

[4] WebDAV is a standard module in Apache 2.x software. Strike Cell implementation for file locking and concurrency access is to append a short character string to each message to indicate message/file termination so that clients or the server will not try to process incomplete messages/files that are still being written by Apache.

Data Format/Structure:

Table 1. Target Update Message

Object	Attribute	DataType	Description
TWS Message	rtiid	long	Unique ID given by RTI
	twsid	String	Unique ID given by TWS, STN
	owner	Enumeration	TST Cell Site currently owning this target.
	force	Enumeration	Force Code of the target
	type	Enumeration	Unit type for the target
	latitude	double	current "perceived" location of the target
	longitude	double	current "perceived" location of the target
	altitude	double	current "perceived" location of the target
	heading	double	current "perceived" direction of the target
	speed	double	current "perceived" speed of travel of the target
	description	String	"Perceived" descriptive note on the target.
	status	Enumeration	current phase in the target development process
	sensor	Enumeration	current sensor system sending data on the target
	trackid	String	local system id for a target
	unitcount	Integer	"Perceived" number of vehicles assoc. with a target
	damage	Enumeration	"Perceived" operational status of the target
	alert	Enumeration	Alert level of a target
	istarget	Boolean	Denotes a target or non-target
	istst	Boolean	Denotes a TST target or non-TST target
	tststart	long	Time when a target became a TST
	tstend	long	"Window of Opportunity" on a TST end time.
	mensurationreport	String	9-line message results from mensuration
	viewable	Boolean	Denotes COP visibility of the target
	mysite	Enumeration	TST Cell Site currently working this target.
	mysystem	Enumeration	TST Cell System working this target
	timestamp	long	Time of a data update

System Translators:

The system translators accomplish 2 tasks. First, they format a system's current output into the data fields that can be translated in the Target Data Structure. Second, they convert the reformatted data and provide the send/receive functions for sharing with other systems over the communication layer.

Communication Layer:

The communication layer is the transport mechanism by which the various systems are tied together and share data with one another. The selection of the communication method will be largely based on a number of factors including: situation (LAN systems, WAN systems, wireless, other), available hardware, number of systems to integrate, estimated target set size (few dozen, thousands, etc…).

Data Storage System:

An online storage system is necessary to maintain near-real-time status on all targets being worked by the systems and service requests for information on those targets. Since each system only provides one part of the puzzle, the storage system, possessing the overall view for each target, can share the combined view of all the data to each client system. Additionally, as clients connect/disconnect for any reason, they are able to rejoin and not miss any data.

II. Description of C2PC[5]

The C2PC system is comprised of three components: a Global Command and Control System (GCCS) Unified Build (UB) host machine, the C2PC Client, and the C2PC Gateway. The UB host machine is the central source that feeds information to the C2PC Gateway and C2PC Client. C2PC requires a UB host in order to receive automatic updates to the information that it tracks. The UB host also provides the overlays and operator notes (OPNOTES) that may be imported into C2PC. It receives any quick reports, OPNOTES, or overlays sent out by C2PC, and it processes them appropriately. This represents the "Tactical Common Operational Picture".

The C2PC Gateway component processes track information received from and sent to the UB host. The Gateway must be up and running for the C2PC Client to receive track updates. The workings of the Gateway are transparent to the user. The C2PC Client component displays the map window and C2PC menu structure. Each C2PC workstation must contain a Client to run C2PC. The following figure shows the use of C2PC in the SIMEX environment that includes the implementation of a TWS client as a C2PC overlay. TWS would use Atlas for its Graphical User Interface (GUI) to display track icons and map data, the Tactical Management System (TMS) Application Programming Interface (API) to acquire track data, and continue to use the TWS Server to develop, maintain, and process TSTs.

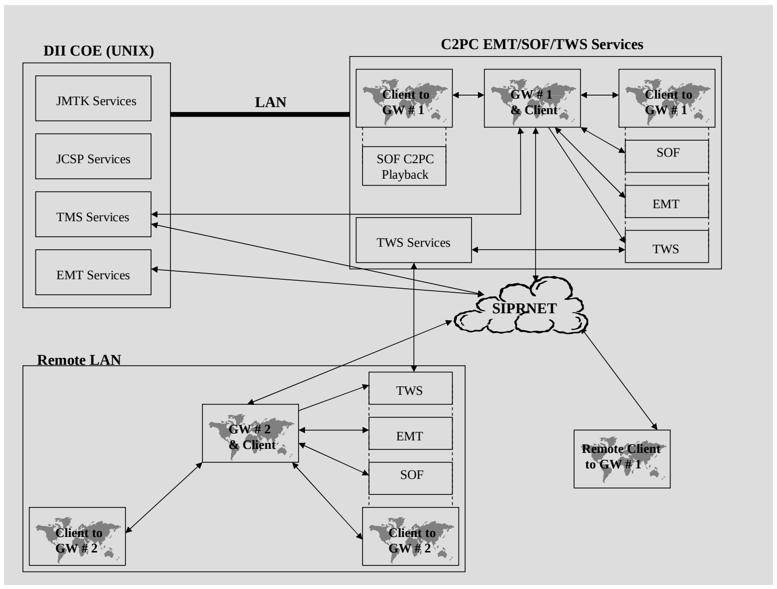


Figure 4. C2PC Data Flow

C2PC sets up a gateway connection for track and general military intelligence (TDMS and MIDB) for client workstations. Add-on applications can also provide more detailed MIDB data and query capabilities. The Intel Office client adds Land Track Query, Intel Query Tool, and Intel Filters to the TrackPlot pulldown. The Special Operations Forces (SOF) client adds SOF sensor reports through the Tools/Sensor pulldown menu.

[5] Command & Control PC (C2PC) User's Guide, Version 5.9.0.3, 13 December 2002

The C2PC window displays whenever you start the program. The C2PC window gives you access to the C2PC tools and features.

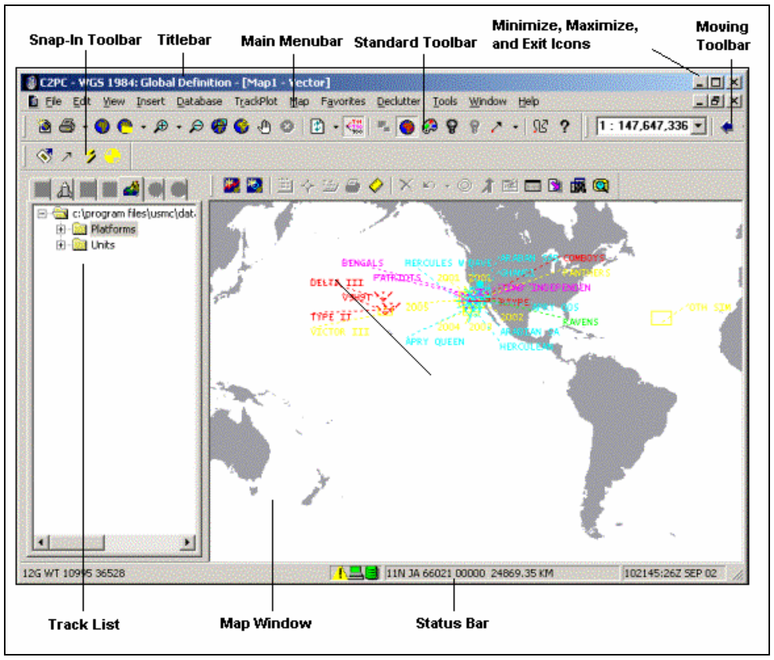


Figure 5. C2PC Window

The C2PC window consists of:

- Title bar
- Menu bar
- Toolbars
- Map window
- Tracks, Overlays, Routes, Units, or Formations List
- Status bar

C2PC has been designed to support application overlays. This allows developers to add functionality to C2PC by using the API Software Development Kits (SDKs) provided by the C2PC Program Office. Two of these SDKs are available to registered C2PC developers: C2PC TMS for database access; and C2PC Atlas for chart and mapping displays. TMS provides database distribution and management of tactical track data. Atlas services provide a standard interface through which applications can communicate with DII COE services for mapping, cartographic, geodetic, and imagery visualization.

III. Phase I Integration of TWS/C2PC – TMS

One of the primary goals of integrating TWS with C2PC is to minimize the amount of source code changes required to merge these two applications. Since TWS was written in Java, the Java Native Interface (JNI)[6] was used to access the TMS database and track data management features.[7] JNI allows Java code to operate with applications and libraries written in other languages, such as C, C++, and assembly.

[6] The Java Tutorial, Sun Microsystems, Inc., 2002.
[7] C2PC Tactical Management System (TMS) Programmer's Guide and Reference Document, Version 3.0, December 13, 2002 was used for all references to the TMS API.

The C2PC TMS database is made up of approximately 50 tables that support ten types of tracks[8] and their associated reports with 159 track and 74 report fields. The API provides interfaces that allow program access to the track data as described in Table 2.

Table 2. TMS Interfaces

Interface	**Method**	**Description**
IAddTrack		
	GetAttribute()	Retrieve field based on track type and enumerated track field name
	SetAttribute()	Set field based on track type and enumerated track field name
	GetReportAttribute()	Retrieve field based on track type and enumerated report field name
	SetReportAttribute()	Set field based on track type and enumerated report field name
	Clear()	Reset all the track attributes to default values
	Update()	Sends field update to the database
IEditTrack		
	GetAttribute()	Retrieve field based on track type and enumerated track field name
	SetAttribute()	Set field based on track type and enumerated track field name
	GetReportAttribute()	Retrieve field based on track type and enumerated report field name
	Update()	Sends field update to the database
	GetMultipleAttributes()	Retrieve fields based on track type and array of enumerated track field names
	GetMultipleReportAttributes()	Retrieve fields based on track type and array of enumerated report field names
IGetTrackInfo		
	GetAttribute()	Retrieve field based on track type and enumerated track field name
	GetReportAttribute()	Retrieve field based on track type and enumerated report field name
	GetMultipleAttributes()	Retrieve fields based on track type and array of enumerated track field names
	GetMultipleReportAttributes()	Retrieve fields based on track type and array of enumerated report field names
IAddReport		
	GetReportAttribute()	Retrieve field based on track type and enumerated report field name
	SetReportAttribute()	Set field based on track type and enumerated report field name
	Update()	Sends field update to the database
	Clear()	Reset all the report attributes to default values
	GetMultipleReportAttributes()	Retrieve fields based on track type and array of enumerated report field names
IEnumerateTracks		
	Reset()	Reset counter to beginning of track list

[8] Even though TMS supports ten types of tracks, only five of these can be updated by C2PC: (1) Unit; (2) Platform; (3) Acoustic; (4) Emitter; and (5) COMINT.

	GetCount()	Get count of tracks based on track type
	Refresh()	Build new track list
	Next()	Get next number of tracks on list
	Skip()	Skip number of tracks on list
	All()	Get all tracks – not limited to track type
General Methods		
	DeleteTrack()	Deletes track and associated reports from the database
	GetTrackCount()	Returns track count based on track type
	GetUID()	Returns Unique ID (UID) based on Command and Received Track Number (RTN)

TMS API Server

The TMS API SDK was installed on a Dell notebook running Windows 2000. Two applications were provided with the SDK to assist programming personnel with implementing API methods in their software. The supplied applications were developed in line with the coding examples listed in the Programming Guide. The test application "APIDriver[9]" was compiled and used as the model for Java development. The GUIs for both applications are displayed in Figure 6.

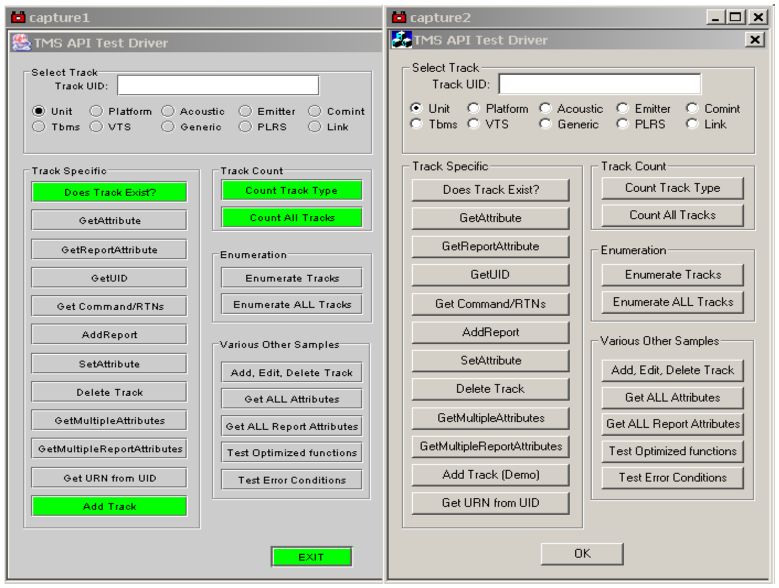


Figure 6. APIDriver Windows

The left window is the GUI from the Java application. The right window is the original C++ APIDriver display. A support program, R2J, which uses the C/C++ ".rc" file, was used to build the Java GUI[10]. The supporting "void" Java code was developed to process the user responses and call the

[9] Software provided with the TMS SDK was incomplete. Two source files were missing for the APIDriver application. The second application, TMSNotifierTestClient, had to be built with support from C2PC programming personnel. Discussions will be limited to APIDriver, since it is a more complete example for developing Java code using the TMS API SDK

[10] R2J was developed to use Java Version 1.2. The current version of Java (1.4.x) uses a slightly different "LookandFeel" manager. The original line "UIManager.setLookAndFeel("com.sun.javax.swing.plaf.windows.WindowsLookAndFeel")" was replaced with "UIManager.setLookAndFeel(UIManager.getCrossPlatformLookAndFeelClassName())".

appropriate JNI function. Only four functions were developed in the Java version of APIDriver: (1) Add Track; (2) Count Track Type; (3) Count All Tracks: and, (4) Does Track Exist.

Before any methods can be used, the TMS API Server object must be instantiated using CoCreateInstance. The following code example is from the Programming Guide that accomplishes this.

```
#import <TmsApiServer.tlb>

ITmsAPIEx* m_spModifier;  // pointer to the TMS API Server Object

// CoCreate TrackModification object
HRESULT hRes = CoCreateInstance(CLSID_TmsModifier, NULL, CLSCTX_LOCAL_SERVER,
IID_ITmsAPIEx, (void**)(&m_spModifier));
```

The TMS APIDriver "Add Track" creates a track based on the track type selected with the radio button in the upper left panel. A track maintains two "keys". The first key is the Unique ID (UID), which is assigned at the time of creation. This key can be changed by a GCCS Server, with an update back to the originator with the new UID. The second key is a combination of the Command Code (CMD) and Received Track Number (RTN) that are assigned at the time of creation. If the CMD and RTN are NOT updated at the time of creation, these two fields will remain with system generated default values. The CMD and RTN will remain with the track until it is deleted. An add track example follows[11]:

```
    m_spModifier = NULL;
    CoInitialize(NULL);
    jint result = 0;

HRESULT hRes = CoCreateInstance(CLSID_TmsModifier, NULL, CLSCTX_LOCAL_SERVER,
IID_ITmsAPIEx, (void**)(&m_spModifier));

 if (m_spModifier) {
   try {
     m_spModifier->AddTrack(TrackType, &pUnk);
```

After the track is created, fields can be set before the "Update[12]" method is invoked[13], as follows:

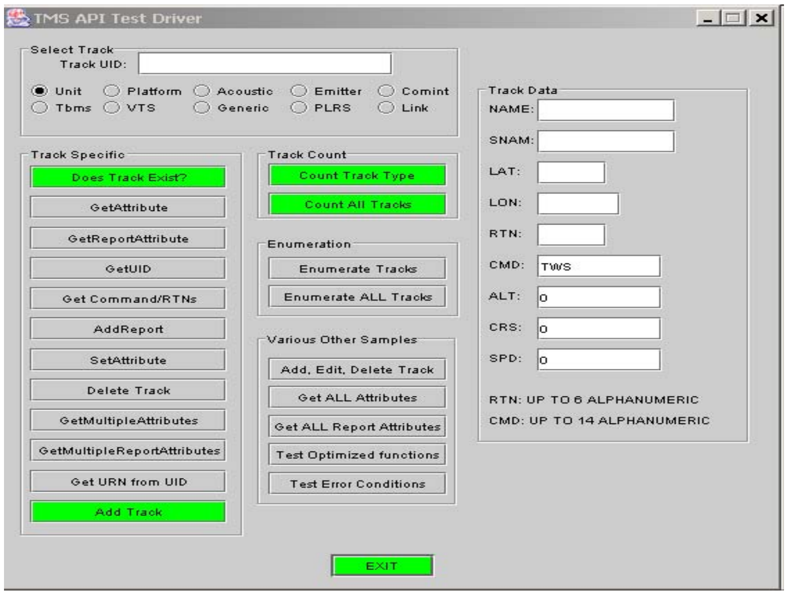
```
   if (pUnk) {
     pUnk->QueryInterface(IID_IAddTrack,(void**)&pAddTrack);
     if (pAddTrack) {
             _variant_t vt = track_LAT;
       // Try to set an attribute
       pAddTrack->SetReportAttribute(RPT_LATITUDE, vt);
             vt = track_LON;
       pAddTrack->SetReportAttribute(RPT_LONGITUDE, vt);
       if (SUCCEEDED(pAddTrack->Update())) {
       Sleep(2000); //wait for any threads to finish
       }
       else {
         AfxMessageBox("Error Adding Track");
```

[11] Complete code is not included in the examples. See attachments for completed Java and C++ programs.
[12] APIDriver used a display message after the "Update" method. Testing showed that a time delay was required if a display was not used. Further analysis is required on this problem.
[13] During testing of both the Java and C++ APIDriver, it was determined that the SetAttribute method for "TRK_NAME" and TRK_SHORTNAME" did not function. These two fields could only be modified after an add/update, then performing an "EditTrack".

```
        }
```

JNI Implementation

Using APIDriver's "Add, Edit, Delete Track" as a template for data access, a Java to C++ interface was designed using JNI. The basic Java GUI was modified to allow the user to add and query ("Does Track Exist?") tracks using selected TWS fields available in TMS, as depicted in the Figure 7.



NAME:	
SNAM:	
LAT:	
LON:	
RTN:	
CMD:	TWS
ALT:	0
CRS:	0
SPD:	0

RTN: UP TO 6 ALPHANUMERIC
CMD: UP TO 14 ALPHANUMERIC

Figure 7. Java Add/Query Window

TWS uses a TADIL-J message for communicating with the C2 environment[14]. This allows tracks to be processed for targeting by tactical air assets. As part of the SIMEXs, each node is allocated a set of TADIL message codes so that analysts will know where targets were generated. The TADIL target number is normally maintained in the GCCS "TRK_SHORTNAME" field and displayed as "Shortname". Since one of the two primary key sets for TMS are RTN/CMD, the Java/JNI/C++ applications use an RTN and CMD fields to create tracks. The "Shortname" field should be set to the same value as RTN.

After the user sets values for the entries in the right-most panel and selects a track type (via the radio buttons), and selects "Add Track", the following Java code is executed to pass data to the JNI class:

```
        Interface4.AddTrk(trackType1,
                track_NAME,
                track_SNAME,
                lat1,
                lon1,
                track_RTN,
                track_CMD,
                alt1,
                crs1,
                spd1);
```

The JNI code is:

[14] Strike Network (STRIKENET) Concept, Kevin M. Forbes, 1 March 2003

```
class Interface4 {
   public static native int AddTrk(
                        int trackType1,
                        String track_NAME,
                        String track_SNAME,
                        double lat1,
                        double lon1,
                        String track_RTN,
                        String track_CMD,
                        double alt1,
                        double crs1,
                        double spd1);
}
```

After compiling the Java application, an "Interface4.class file will be created. Header files are then created by executing "javah -jni main_java_program Interface4", which builds the following Interface4.h file:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Interface4 */

#ifndef _Included_Interface4
#define _Included_Interface4
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     Interface4
 * Method:    AddTrk
 * Signature: (ILjava/lang/String;Ljava/lang/String;DDLjava/lang/String;Ljava/lang/String;DDD)I
 */
JNIEXPORT jint JNICALL Java_Interface4_AddTrk
  (JNIEnv *, jclass, jint, jstring, jstring, jdouble, jdouble, jstring, jstring, jdouble, jdouble, jdouble);

#ifdef __cplusplus
}
#endif
#endif
```

The C++ application requires JNI level code to interface with the original Java application, which follows:

```
JNIEXPORT jint JNICALL Java_Interface4_AddTrk (
            JNIEnv *env,
            jclass cls,
            jint trackType,
            jstring track_NAME,
            jstring track_SNAME,
            jdouble track_LAT,
            jdouble track_LON,
            jstring track_RTN,
            jstring track_CMD,
            jdouble track_ALT,
            jdouble track_CRS,
            jdouble track_SPD)
```

At this time, the C/C++ application is ready to access/update the passed variables. The C/C++ application must be compiled with the following parameters[15] to create a ".dll" file.

```
cl -Ic:\j2sdk1.4.1_01\include -Ic:\j2sdk1.4.1_01\include\win32 -IC:\TWS_Dev\TmsSDK\Debug\bin -LD
ConnTMS.cpp -FeConnTMS.dll
```

One other step must be performed prior to execution of these applications. The original Java application must invoke a "load library" for the dll, as follows:

```
  static {
    System.loadLibrary("ConnTMS");
  }
```

Two interfaces were developed for "Does Track Exist?". The first interface provided single field retrieval capability and the second provided multi-field retrievals. The following Java code segment invokes "Interface3.TrackExists" to get individual fields returned from TMS and Interface5.TrackExists using an array of enumeration values for the "GetMuptipleReportAttributes()" method. The JNI uses another array for the returned data.

```
 void jButtonIDC_TRACK_EXISTS_actionPerformed(ActionEvent e) {
  //Need to get Track UID and Track Type to pass to interface
        track_UID = null;
        jTextFieldIDC_EDIT_UID.setText(track_UID);
        trackType1 = Integer.decode(trackType).intValue();
        track_CMD = jTextFieldIDC_EDIT_CMD.getText().toUpperCase();
        track_RTN = jTextFieldIDC_EDIT_RTN.getText().toUpperCase();
    trackHold=Interface3.TrackExists(trackType1,track_RTN,track_CMD,1);//UID
        jTextFieldIDC_EDIT_UID.setText(trackHold);
    trackHold=Interface3.TrackExists(trackType1,track_RTN,track_CMD,2);//SHORTNAME
        jTextFieldIDC_EDIT_SNAME.setText(trackHold);
    trackHold=Interface3.TrackExists(trackType1,track_RTN,track_CMD,3);//NAME
        jTextFieldIDC_EDIT_NAME.setText(trackHold);
    Interface5.TrackExists(trackType1,track_RTN,track_CMD,arrayType, arrayRet);
    trackHold = formatCoords(arrayRet[0],true,0);//LAT
        jTextFieldIDC_EDIT_LAT.setText(trackHold);
    trackHold = formatCoords(arrayRet[1],false,0);//LON
        jTextFieldIDC_EDIT_LON.setText(trackHold);
    alt1 = arrayRet[2];//ALT
        jTextFieldIDC_EDIT_ALT.setText(Double.toString(alt1));
    crs1=arrayRet[3];//CRS
        jTextFieldIDC_EDIT_CRS.setText(Double.toString(crs1));
    spd1=arrayRet[4];//SPD
        jTextFieldIDC_EDIT_SPD.setText(Double.toString(spd1));
 }
```

Interface 3 passes a single "fieldType" variable while Interface5 uses an array. Values are based on the eTrackDataFields enumeration table in C2PCTmsAPIServer.tlh, which was provided with the TMS SDK.

```
class Interface3 {
   native public static String TrackExists(
                 int trackType1,
                 String track_RTN,
```

[15] This example is set up for Java SDK Version 1.4.1 and Microsoft C++ Version 6.0

```
                        String track_CMD,
                        int fieldType);
}
```

```
class Interface5 {
     native public static void TrackExists(
                        int trackType1,
                        String track_RTN,
                        String track_CMD,
                        long [] arrayType,
                        double [] arrayRet);
}
```

The following code segment is the C++ method for Interface5. For JNI to access the C++ arrays, the “Release” statements at the end of the method must be invoked prior to the “return”.

```
JNIEXPORT void JNICALL Java_Interface5_TrackExists (
            JNIEnv *env,
            jclass cls,
            jint trackType,
            jstring track_RTN,
            jstring track_CMD,
            jlongArray arrayType,
            jdoubleArray jdaRet)
{
CString cs;
CString csMsg;

jfieldID fid;
jstring jstr;
const char *str;
jdouble jHold;

unsigned int num_arrays = env->GetArrayLength(arrayType);
jlong *element = env->GetLongArrayElements(arrayType,0);

unsigned int num_array1 = env->GetArrayLength(jdaRet);
jdouble *arrayRet = env->GetDoubleArrayElements(jdaRet,0);

 const char *csCommand = env->GetStringUTFChars(track_CMD, NULL);
  const char *csRTN = env->GetStringUTFChars(track_RTN, NULL);
  myEnumeration(trackType);
  m_spModifier = NULL;
  CoInitialize(NULL);
  IUnknown* pUnk = NULL;
  jint result = 0;
  jdouble sum;
  CString csCommandRtn;
  CString csUID;
  HRESULT hRes = CoCreateInstance(CLSID_TmsModifier, NULL, CLSCTX_LOCAL_SERVER,
IID_ITmsAPIEx, (void**)(&m_spModifier));
  try {
    if (m_spModifier) {
       csCommandRtn.Format("%s|%s",csCommand, csRTN);
      _bstr_t bstrCommandRtn(csCommandRtn);
      hRes = m_spModifier->GetTrackInfo(holdTrackType, bstrCommandRtn, &pUnk);
    if (SUCCEEDED(hRes)) {  // Track exists
      if (pUnk) {
        pUnk->QueryInterface(IID_IGetTrack,(void**)&pGetTrack);
        if (pGetTrack) {

        SAFEARRAY *psaRequest;
        SAFEARRAY *psaReturn;

        ////////////////////////////////////////
        // Create and Fill in Request SafeArray
        ////////////////////////////////////////
        SAFEARRAYBOUND rgsabound[1];
        rgsabound[0].lLbound = 0;
        rgsabound[0].cElements = num_arrays;

        psaRequest = SafeArrayCreate(VT_I4, 1, rgsabound);

                    long *pFields;
                    HRESULT hr = SafeArrayAccessData(psaRequest,(void HUGEP* FAR*)&pFields);
              for (int i = 0; i < num_arrays; i++) {
          pFields[i] = element[i];
              }

        SafeArrayUnaccessData(psaRequest);
        pGetTrack->GetMultipleReportAttributes(psaRequest, &psaReturn);

        VARIANT *pVariant;
              hr = SafeArrayAccessData(psaReturn,(void HUGEP* FAR*)&pVariant);

        _variant_t vt;
        for (i = 0; i < psaReturn->rgsabound[0].cElements; i++) {
          vt = pVariant[i];
          sum = vt;
          arrayRet[i] = sum;
              }

              SafeArrayUnaccessData(psaReturn);

        SafeArrayDestroy(psaRequest);
        SafeArrayDestroy(psaReturn);
        pGetTrack->Release();
        pGetTrack = NULL;
        }
      }
      pUnk->Release();
      pUnk = NULL;
    }
  }
}
catch ( _com_error &ce ) {
              CString msg = "GetMultipleReports Error: ";
              CString errmsg = static_cast<BSTR>( ce.Description() );
              if ( errmsg.IsEmpty() ) {
                          msg += ce.ErrorMessage();
              } else {
```

```
                                msg += errmsg;
                  }
    if (pGetTrack) {
      pGetTrack->Release();
      pGetTrack = NULL;
    }
    if (pUnk) {
      pUnk->Release();
      pUnk = NULL;
    }
    AfxMessageBox(msg, MB_SETFOREGROUND);
  }
env->ReleaseLongArrayElements(arrayType,element,0);
env->ReleaseDoubleArrayElements(jdaRet,arrayRet,0);
return;
}
```

c

Examples of the Java and C2PC track lists are shown in the following figure[16].

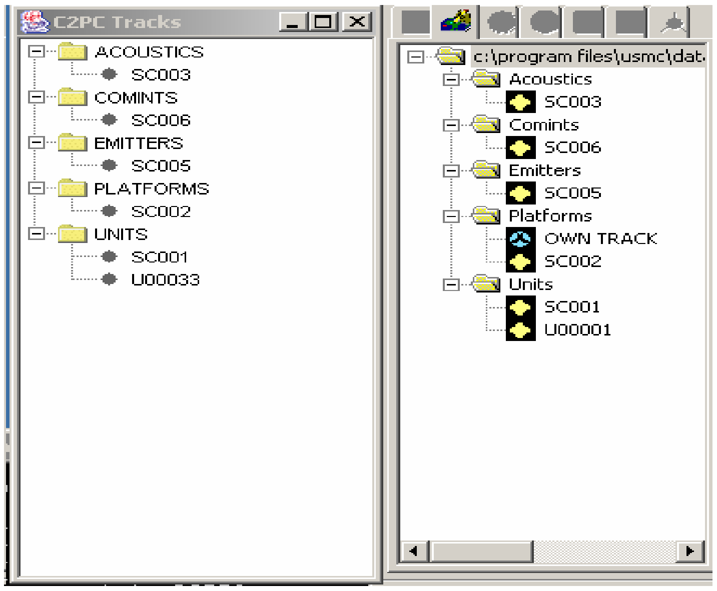


Figure 8. Track Lists

C2PC/JAVA APIDriver Execution

During the testing of the Java/JNI/C++ applications, several problems were noted:

- The same code is used to add tracks, where the "Track Type" parameter is used to designate the type of track. All five types that can be added by the TMS API were stable.
- With an active C2PC/GCCS interface, trying to add an Acoustic track failed. If C2PC was not connected to a GCCS Server, Acoustic tracks could be added. After connecting C2PC to a GCCS Server, the Acoustic tracks would remain.

[16] The Java application displays the RTN, while C2PC displays the track shortname, if available, or the Local Track Number (LTN).

- The Programmer’s Guide documentation for the API enumeration tables was not complete. Several tables were outdated, requiring review of the C++ support files included with the SDK.

IV. TWS / C2PC CONOPs

Initial review and testing of the C2PC SDK TMS API supported the use of JNI for Java application access to the C2PC database. Several discrepencies were noted with the SDK and reported to the C2PC Program Office. The implementation of a JNI API for each of the original C++ interfaces would allow Java and other programming languages to access the C2PC database, even across multiple platforms and operating systems.

Other C2PC SDKs are available, including the Communications, Atlas (C2PC GUI), and Alerts interfaces. These SDKs should also be evaluated to verify their interoperability with Java and other programming languages.