Technical Paper

NavRadar – BRPC SDK

NRP Target Tracking class description

Document Version:

D [Release], Date: 15-Aug-2016

Abstract:

This document describes how to use NRP Target Tracking Class to interface software with Navico Radar MARPA functionality.

Document:	Total:
4	4

Revision History:

A	21-Sept-2009	Document created	Graeme Bell
B	11-Dec-2009	Updated for Library 1.0 Release	Graeme Bell
C	13-July-2016	Update for Library 3.0 Release adding Halo radar	Kevin Peng
D	15-Aug-2016	Update for Library 3.0 Release adding pulse radar	Kevin Soole

Glossary:

NRP	NavRadar Protocol. Protocol used in NAVICO BROADBAND, HALO and 12U/6X (PULSE) radar
ARPA	*Automatic radar plotting aid* – this is used to acquire and track targets and is primarily intended for collision avoidance. A less capable variant of this is Mini-ARPA or MARPA. Note: care is required in the correct usage of these terms as standards of performance and features may apply depending on market type.
CPA	*Closest Point of Approach* – It is the point where target and boat could collide if target or boat speed or course doesn't change.
MARPA	*Mini automatic radar plotting aid* – see ARPA.
Plot	Single echo target or union of different echo targets that merged together produced an object could be tracked in the region of interest. In some document this is called UAT
TCPA	*Time to Closest Point of Approach* – It is the time necessary to reach the collision condition if target or boat speed or course doesn't change.

Table of Contents

Table of Contents ii
1 Overview 1
2 NRP Target Tracking protocol 2
2.1 Target Identification 2
2.2 Target Information 2
2.2.1 Target Tracking States 3
2.3 State Information 5
2.3.1 Alarm Setup 5
2.3.2 Boat Properties 5
3 Class `tTargetTrackingClient`: definition and usage 6
3.1 Commands to the radar 6
3.1.1 bool Acquire(uint32_t id, uint32_t range_meters, uint16_t bearing_degrees, bool trueNorth) 7
3.1.2 bool Cancel(uint32_t serverID) 7
3.1.3 bool CancelAll() 7
3.1.4 bool SetBoatSpeed(eSpeedType speedType, uint32_t speed_metersPerSecondX10, eBearingType bearingType, uint32_t bearing_degrees) 7
3.1.5 bool QueryAll() 8
3.1.6 bool QueryAllTargets() 8
3.1.7 bool QueryAlarmSetup() 8
3.1.8 bool QueryProperties() 8
3.1.9 bool SetDangerDistance(uint32_t range_meters) 8
3.1.10 bool SetDangerTime(uint32_t time_seconds) 8
3.2 State Messages from the radar - iTargetTrackingClientStateObserver 8
3.2.1 void UpdateAlarmSetup(const t9174TargetTrackingAlarmSetup *pAlarmSetup) 8
3.2.2 void UpdateProperties(const t9174TargetTrackingAProperties *pProperties) 9
3.3 Tracked Targets from the radar - iTargetTrackingClientObserver 9
3.3.1 void UpdateTarget(const t9174TargetTrackingTarget *pTarget) 9
3.4 Class initialisation and configuration 9
3.4.1 int Connect(const char * pSerialNumber) 9
3.4.2 bool Disconnect() 9
3.4.3 bool AddStateObserver(iTargetTrackingClientStateObserver *pObserver) 9
3.4.4 bool RemoveStateObserver(iTargetTrackingClientStateObserver *pObserver) 10
3.4.5 bool AddTargetTrackingObserver(iTargetTrackingClientObserver *pObserver) 10
3.4.6 bool RemoveTargetTrackingObserver(iTargetTrackingClientObserver *pObserver) 10

1 Overview

This document describes how to use the NRP Target Tracking class to connect to a radar, and perform simple operations like acquiring a target and processing updates on their position. All data structures defined and described in this document are byte aligned and in little endian format.

Target Tracking tracks objects based on Radar images, so any change that affects the Radar image can also affect the tracking algorithm (eg. if the gain changes the signal can became too weak or to strong to be correctly tracked). Good results are obtained having all gains in Auto mode.

Another important factor, for obtaining correct tracking from a mobile platform, is to connect a good heading sensor to the radar. This is useful for having a stabilised radar image and for determining the absolute position of different tracked objects. For improving tracking quality you should supply the speed and direction of the platform/vessel the radar is mounted on.

When tracking correctly the radar will be capable of supplying the following parameters on a target:

- Position expressed as range and bearing (both in true/absolute and in relative mode);
- Speed expressed as speed of the object and course (both in true/absolute and in relative mode);
- Estimated CPA (closest point of approach) and TCPA (time for closest point of approach)

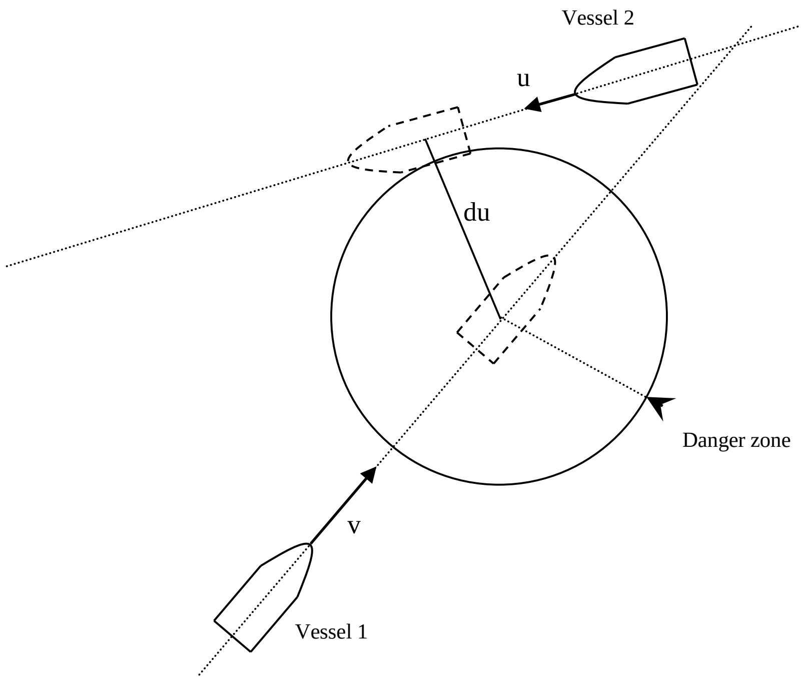

Figure 1: Closest Point of Approach

In the previous figure, we have 2 vessels (1 and 2) that are moving respectively with speed and direction u and v. The CPA is the minimum distance two vessels can reach if conditions don’t change. The TCPA is the time to reach that situation.

The tracked target can be *safe* or *dangerous*. A target can be considered *dangerous* if the CPA lies within the radius of the danger-zone ring and happens in a time that is less than the TCPA. A negative TCPA means that the CPA has already been left behind.

2 NRP Target Tracking protocol

Currently the communications protocol is based on multicast UDP/IP, with each radar on the network using a separate set of multicast addresses for receiving commands and the transmission of state plus target information.

2.1 Target Identification

When the radar receives a valid Acquire command it will allocate it a target tracking identifier (ServerTargetID) and associated resources. This ServerTargetID will be used in all subsequent communications to identify that target, and once allocated cannot be reallocated for other new targets until the radar has been told to release it via a `Cancel` command.

A valid ServerID will always lie in the range 1 to the maximum number of targets able to be tracked, in this case 10. If the Acquire command is invalid, or no further resources are available a negative ServerTargetID is be used to indicate there is a problem.

2.2 Target Information

The following structure is used to report the current status of targets.

```
struct t9174TargetTrackingTarget
{
    uint8_t  targetValid;     // 0x00 invalid, 0x01 valid
    uint8_t  trueTarget;      // 0x00 simulated target, 0x01 true target
    uint32_t targetType;      // 0x00 vessel
    uint32_t targetID;        // client target id
    uint32_t serverTargetID;  // server target id
    uint32_t targetState;     // 0x00: acquiring;  0x01: safe target;  0x02: dangerous target
                              // 0x03: lost target; 0x04: acquire failure; 0x05: out of range
                              // 0x06: lost out of range;
                              // 0x10: failed acquire: max target exceeded;
                              // 0x11: failed acquire: invalid position
    uint32_t autoAcquired;    // 0x00: manually acquired; 0x01: automatically acquired

    uint32_t relativeRange_meters;           // relative distance expressed in meters
    uint32_t relativeBearing_degreesX10;     // relative bearing expressed in deci-degrees
    uint32_t relativeCourse_degreesX10;      // relative course expressed in deci-degrees
    uint32_t relativeSpeed_metersPerSecX10;  // relative speed in m/s multiplied by 10

    uint32_t absoluteRange_meters;           // absolute distance expressed in meters
    uint32_t absoluteBearing_degreesX10;     // absolute bearing expressed in deci-degrees
    uint32_t absoluteCourse_degreesX10;      // absolute course expressed in deci-degrees
    uint32_t absoluteSpeed_metersPerSecX10;  // absolute speed in m/s multiplied by 10

    uint8_t  absoluteValid;   // 0x00 absolute info invalid, 0x01 absolute info valid

    uint32_t CPA_meters;      // closest point of approach expressed in meters
    uint32_t TCPA_seconds;    // time to the closest point of approach expressed in seconds
    uint8_t  towardsCPA;      // 0x00 not towards CPA, 0x01 towards CPA
};
```

The fields are described in more detail in the following table.

Field	**Description**
`targetValid`	Following target information is valid; value is 0 if invalid or 1 if valid
`trueTarget`	Real or simulated target; 0 if simulated, 1 if real/true target
`targetType`	Type of target being tracked; 0 for vessel (only valid option)

`targetID`	User/client defined identifier for target
`serverTargetID`	Radar allocated target identifier; lies in the range 1..Max Trackable Targets for tracked targets, or may be less than zero if unallocated (e.g. invalid acquire)
`targetState`	Current state of the target. See Target Tracking States below.
`autoAcquired`	How target acquisition was initiated; 0x00 manually, 0x01 automatic
`relativeRange_meters`	Relative distance from radar to tracked target in metres
`relativeBearing_degreesX10`	Angle between this vessel and the one being tracked; in 10ths of a degree clockwise from the radars zero heading
`relativeCourse_degreesX10`	Angle of the targets direction relative to this vessel; in 10ths of a degree clockwise from the radars zero heading
`relativeSpeed_metersPerSecX10`	Relative speed of the target being tracked; in 10ths of a metre per second
`absoluteRange_meters`	Distance from radar to tracked target in metres (same as `relativeRange_meters`)
`absoluteBearing_degreesX10`	Angle to the target being tracked; in 10ths of a degree from north
`absoluteCourse_degreesX10`	Angle of the targets direction; in 10ths of a degree from north
`absoluteSpeed_metersPerSecX10`	Speed of the target being tracked; in 10ths of a metre per second
`absoluteValid`	Whether the absolute information being reported above is valid; 0 if invalid, 1 if valid. To be valid requires that the radar is attached to an appropriate sensor, or is being kept informed of the vessels heading and possibly its position via the `SetBoatSpeed` command
`CPA_meters`	Estimated distance from the target at the closest point of approach; in metres (see Overview)
`TCPA_seconds`	Amount of time before CPA will be reached; in seconds from now
`towardsCPA`	Direction of CPA; 0 moving away from CPA (already passed it), 1 moving towards CPA

Table 1: Target Information Structure

2.2.1 Target Tracking States

The follow diagram illustrates the states a target (ServerTargetID and resources) may possess, and the circumstances in which it will transition from one state to the next.

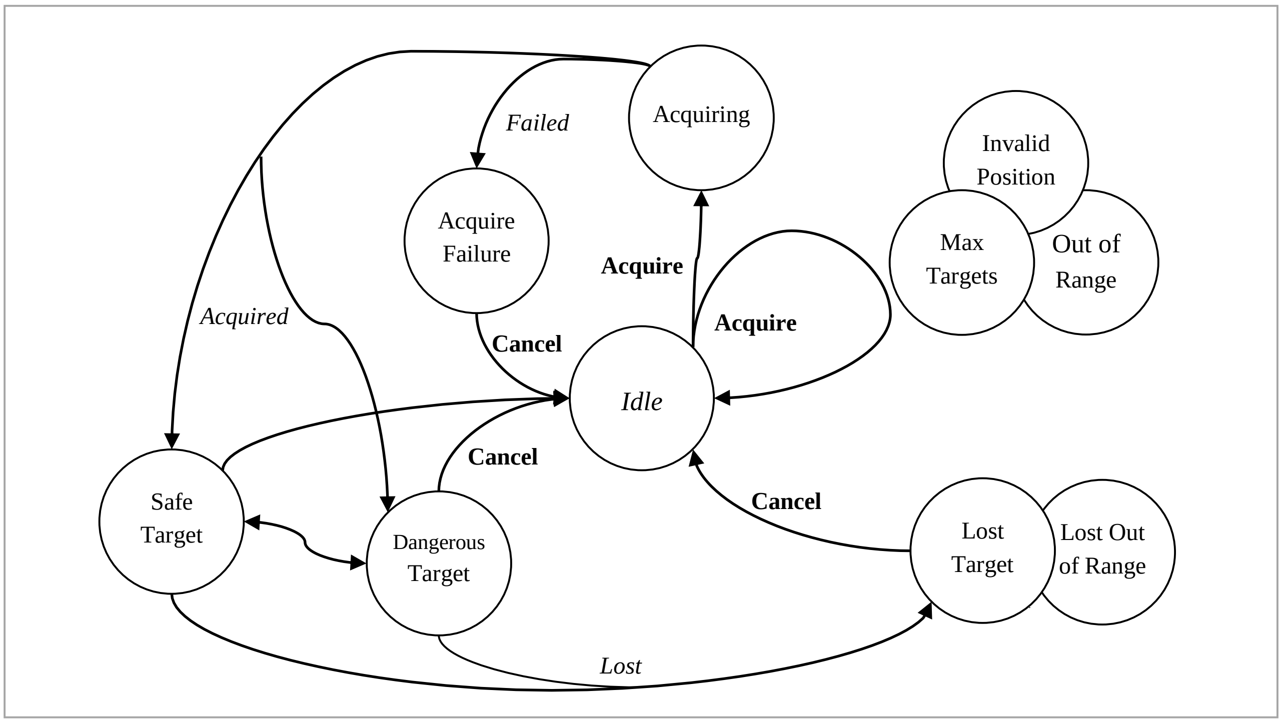

Figure 2: Target Tracking State Transitions

When an acquire command is not rejected the radar allocates a ServerTargetID and transitions its state to “Acquiring”.

State	**Description**
Idle	Initial target state. No ServerTargetID or associated resources have been allocated
Max Targets	Fake state indicating that an `Acquire` command has been rejected as the maximum number of targets (ServerTargetIDs) is already in use. This will continue to occur until at least one target is cancelled.
Invalid Position	Fake state indicating that an `Acquire` was rejected because an invalid position (bearing and range) was specified.
Out of Range	Fake state indicating that an `Acquire` was rejected because the specified position is out of the radars range.
Acquiring	The radar is attempting to acquire a target at the specified position, which may take several radar scans.
Acquire Failure	The radar failed to find a stable target. You need to `Cancel` this target to release it for a subsequent `Acquire` attempt.
Safe Target	Target has been successfully acquired and will pass a safe distance from this vessel (ie. the estimated CPA & TCPA are outside the dangerous-zone alarm settings.
Dangerous Target	Target has been successfully acquired but may pass dangerously close to this vessel (ie. the estimated CPA & TCPA are inside the dangerous-zone alarm settings).
Lost Target	An acquired target has been lost. You need to `Cancel` this target to release it for a subsequent `Acquire` attempt.

Lost Out of Range	An acquired target has been lost because it passed out of range. You need to `Cancel` this target to release it for a subsequent `Acquire` attempt.

Table 2: Target Tracking States

2.3 State Information

With this service the radar reports its current settings namely; danger-zone alarm settings, and the current boat properties (speed, direction, etc) being used.

2.3.1 Alarm Setup

The following structure is used to report the current alarm setup.

```
struct t9174TargetTrackingAlarmSetup
{
    uint32_t  safeZoneDistance;
    uint32_t  safeZoneTime;
};
```

Field	**Description**
`safeZoneDistance`	Number of metres from the vessel a target must be estimated to pass within before an alarm will be raised (ie. Minimum allowed CPA).
`safeZoneTime`	Maximum number of seconds before the estimated CPA will be reached that will cause an alarm.

Table 3: t9174TargetTrackingAlarmSetup

2.3.2 Boat Properties

The following structure is used to report the current target tracking properties.

```
struct t9174TargetTrackingAProperties
{
    uint32_t  headingType;
};
```

Field	**Description**
`headingType`	The type of heading information currently being used for target-tracking. One of; • `eNoHeading`, unused • `eMagneticNorth`, relative to Magnetic North • `eTrueNorth`, relative to True North

Table 4: t9174TargetTrackingAProperties

3 Class `tTargetTrackingClient`: definition and usage

The `tTargetTrackingClient` class is able to interface a client with a radar without the client needing to worrying about the network protocol. This isolates the clients from future changes to the radar protocol.

The class works with the observer paradigm that means the object interested in receiving target tracking information needs to derive from the observer classes (`iTargetTrackingStateObservers` and `iTargetTrackingObservers`) and register itself with the `tTargetTrackingClient` class.

Example:

```
#include <NRPTargetTrackingClientObserver.h>
#include <TargetTrackingClient.h>

using Navico::Protocol;

class TargetTrackingClient
    : public NRP::iTargetTrackingClientObserver
    , public NRP::iTargetTrackingClientStateObserver
{
...

public:
    // iTargetTrackingClientObserver callbacks
    void UpdateTarget( const TargetTracking::t9174TargetTrackingTarget *target );

    // iTargetTrackingClientStateObserver callbacks
    void UpdateAlarmSetup( const TargetTracking::t9174TargetTrackingAlarmSetup *pAlarmSetup );
    void UpdateProperties( const TargetTracking::t9174TargetTrackingAProperties *pProperties );
...
};
```

Registration of observers with the `tTargetTrackingClient` class can be done using the two methods

```
bool AddTargetTrackingObserver(tNRPTargetTrackingClientObserver *pObserver )
bool AddStateObserver( tNRPTargetTrackingClientStateObserver *pObserver ).
```

After the registration you need to connect to the radar using the following method

```
bool tTargetTrackingClient::Connect( const char * pSerialNumber )
```

Providing the serial-number of the radar you wish to connect too. The serial-number of radars available on the network can be obtained from the `tMultiRadarClient` class.

Once you have successfully connected to the radar the `tTargetTrackingClient` will be able to transmit and receive commands to and from the radar. The reception of state information will start immediately so it is possible for an update call-back to be invoked even before the `Connect` method has returned.

3.1 Commands to the radar

Below are details of the methods provided by the `tTargetTrackingClient` for interacting with a radar.

Most commands return “true” if the operation was successfully initiated. This only means that the command was successfully queued for transmission over the Ethernet, not that the command has been successfully received and processed by the radar.

In most cases the radar will automatically send a status or state update message containing the changed setup or configuration data when it receives a command. For commands that don’t initiate an automatic response an appropriate Query command can be sent. This response can be used to confirm the operation has been completed successfully by the radar.

3.1.1 bool Acquire(uint32_t id, uint32_t range_meters, uint16_t bearing_degrees, bool trueNorth)

This command asks the radar to attempt to acquire a target at a certain distance and bearing relative to the boat. The radar will respond with the new state of the target at which point the `iTargetTrackingObserver::UpdateTarget` call-back will be invoked with the `targetID` field of the `t9174TargetTrackingTarget` structure set to the same as `id`.

Note that targets can not be acquired when the radar is running with its range set to less than 1/8th of a nautical-mile (~230 metres). See section 2 “NRP Target Tracking protocol” for more information about targets and the target acquisition process.

It will return “true” if the command was successful sent, “false” otherwise.

3.1.2 bool Cancel(uint32_t serverID)

This command cancels tracking of the `serverID` target. Where `serverID` is the number the server uses to identify the target. The radar will respond with the new state of the target at which point the `iTargetTrackingObserver::UpdateTarget` call-back will be invoked with the `serverTargetID` field of the `t9174TargetTrackingTarget` structure set to the same as ‘`serverID`’, but with the `targetValid` field false.

It will return “true” if the command was successful sent, “false” otherwise.

3.1.3 bool CancelAll()

This command cancels the tracking of all targets. The radar will respond with the new state of the targets at which point the `iTargetTrackingObserver::UpdateTarget` call-back will be invoked for each cancelled target, with the `serverTargetID` field of the `t9174TargetTrackingTarget` structure set to the targets ServerTargetID and the `targetValid` field false.

It will return “true” if the command was successful sent, “false” otherwise.

3.1.4 bool SetBoatSpeed(eSpeedType speedType, uint32_t speed_metersPerSecondX10, eBearingType bearingType, uint32_t bearing_degrees)

This command can be used, in the absence of heading sensors attached directly to the radar, to inform the radar about the vessels current speed and heading.

Note that unless the radar is fixed to a stationary platform (in which case this message is unnecessary) it is essential that it be provided with valid speed and heading information so it can stabilise the radar image for target tracking.

Field	**Description**
`speedType`	How the radar should interpret the following parameter. One of; • `eSOG`, as the speed-over-ground (ie. absolute speed) • `eWaterSpeed`, as the speed of the vessel relative to the water
`speed_metersPerSecondX10`	Speed of the vessel; in tenths of a metre per second (interpreted as an absolute value, or relative to the water depending on the previous parameter)
`bearingType`	How the radar should interpret the following parameter. Currently the only supported options are one of: • `eNone`, ignore it, • `eCOG`, degrees relative to True North

`bearing_degrees`	Heading of the vessel; as the number of degrees clockwise from the reference direction specified in the previous parameter

Table 5: SetBoatSpeed Parameters

It will return "true" if the command was successful sent, "false" otherwise.

3.1.5 bool QueryAll()

This command asks the radar to send all target tracking related information including: alarm setup; general properties; and tracked targets.

It will return "true" if the command was successful sent, "false" otherwise.

3.1.6 bool QueryAllTargets()

This command asks the radar to send the current tracking information on all targets.

It will return "true" if the command was successful sent, "false" otherwise.

3.1.7 bool QueryAlarmSetup()

This command asks the radar to send the target tracking alarm setup information.

It will return "true" if the command was successful sent, "false" otherwise.

3.1.8 bool QueryProperties()

This command asks the radar to send the current target tracking properties (e.g. boat speed and direction).

It will return "true" if the command was successful sent, "false" otherwise.

3.1.9 bool SetDangerDistance(uint32_t range_meters)

This command sets the danger zone distance. Used in conjunction with the danger zone time to establish when to raise an alarm based on a targets predicted closest point of approach (CPA), and time to CPA.

It will return "true" if the command was successful sent, "false" otherwise.

3.1.10 bool SetDangerTime(uint32_t time_seconds)

This command sets the danger zone time. Used in conjunction with the danger zone distance to establish when to raise an alarm based on a targets predicted closest point of approach (CPA), and time to CPA.

It will return "true" if the command was successful sent, "false" otherwise.

3.2 State Messages from the radar - iTargetTrackingClientStateObserver

Here are listed the `Navico::Protocol::NRP::iTargetTrackingClientStateObserver` observer methods invoked when the radar sends various types of target-tracking state & settings information.

3.2.1 void UpdateAlarmSetup(const t9174TargetTrackingAlarmSetup *pAlarmSetup)

The `pAlarmSetup` parameter points to a structure that contains all the target-tracking alarm setup information sent by the radar. See chapter 2.3.1 "Alarm Setup" for a detailed description of the structure and all its fields.

This state information will be sent by the radar whenever it changes, or it is requested to via the `bool` QueryAll() or `bool` QueryAlarmSetup() commands.

3.2.2 void UpdateProperties(const t9174TargetTrackingAProperties *pProperties)

The `pProperties` parameter points to a structure that contains all the target-tracking setting information sent by the radar. See chapter 2.3.2 "Boat Properties" for a detailed description of the structure and all its fields.

This state information will be sent by the radar whenever it changes, or it is requested to via the `bool` QueryAll() or `bool` QueryProperties() commands.

3.3 Tracked Targets from the radar - iTargetTrackingClientObserver

Here are listed the `Navico::Protocol::NRP::iImageTargetTrackingClientObserver` observer methods invoked when the radar sends target update information.

3.3.1 void UpdateTarget(const t9174TargetTrackingTarget *pTarget)

The `pTarget` parameter points to a structure that contains all the information sent by the radar. See chapter 2.2 "Target Information" for a detailed description of the structure and all its fields.

3.4 Class initialisation and configuration

This describes other methods used in `tNRPTargetTrackingClient` class to initialise and configure it properly.

3.4.1 int Connect(const char * pSerialNumber)

This method connects a `tTargetTrackingClient` object to a specific radar, opening multicast sockets and starting its internal threads so that it can send and receive message to and from the radar.

The parameter `pSerialNumber` must point to the valid serial-number of a radar on the network. A list of radars attached to the network can be obtained from the `tMultiRadarClient` class.

It will return 0 if the operation was successful, or one of the `Protocol::eError` enum values otherwise.

3.4.2 bool Disconnect()

This method disconnects the `tTargetTrackingClient` object from the network, closing all open network sockets and shutting down its internal threads.

It will return "true" if the operation was successful, "false" otherwise.

3.4.3 bool AddStateObserver(iTargetTrackingClientStateObserver *pObserver)

This method adds an observer to all target-tracking "state" messages that can come from the radar. Multiple observers can be registered, and each of them will be informed on the reception of a state message using the appropriate call-back.

Note that update call-back methods are called from a separate thread so you will need to ensure thread safe access to any data maintained external to the `tTargetTrackingClient` object. Also certain methods, such as this one should not be invoked from the call-backs as they may result in a deadlock.

It will return "true" if the operation was successful, "false" otherwise.

3.4.4 bool RemoveStateObserver(iTargetTrackingClientStateObserver *pObserver)

This method removes a previously registered state observer. After an observer has been removed, it will not receive any further updates.

It will return "true" if the operation was successful, "false" otherwise.

3.4.5 bool AddTargetTrackingObserver(iTargetTrackingClientObserver *pObserver)

This method adds an observer to all target-tracking "target" messages that can come from the radar. Multiple observers can be registered, and each of them will be informed on the reception of a state message using the appropriate call-back.

Note that update call-back methods are called from a separate thread so you will need to ensure thread safe access to any data maintained external to the `tTargetTrackingClient` object. Also certain methods, such as this one should not be invoked from the call-backs as they may result in a deadlock.

It will return "true" if the operation was successful, "false" otherwise.

3.4.6 bool RemoveTargetTrackingObserver(iTargetTrackingClientObserver *pObserver)

This method removes a previously registered target observer. After an observer has been removed, it will not receive any further updates.

It will return "true" if the operation was successful, "false" otherwise.