COMPACT TALK

Integration Description

Document version 1.2

WELAND
SOLUTIONS

Contents

1 Introduction

This document describes the third party interfaces used to integrate with Compact Talk. The document will provide a brief system overview and a description of the three ways available to integrate with Compact Talk . CTClient API(Application Programming Interface), XML- Interface and import and export from external sources.

Compact Talk can be communicated with using the following formats:

- Windows Communication Foundation (WCF)
  Compact Talk offers an API called CTClient that simplifies integration and will give full access to all functionality. CTClient is the recommended solution for systems that supports the .NET platform. Protocols that are supported are TCP/IP and IPC (Named Pipes).
- Web services
  To simplify integration from systems that doesn’t support the .NET platform an HTTP based web service is also integrated in Compact Talk. There is no full duplex communication available, so events have to be polled manually.
  The Web service supports SOAP 1.1 and follows WS-I BP 1.1. IIS does not have to be installed.
- XML Interface
  Handles basic command sent via XML-files.
- Import- and export interface
  - Handles import of orders and export of order data via flat files. The visual configuration interface allows for a lot of flexibility.

2 References

[1] Compact Talk V3 Configuration Manual
[2] API reference manual (CompactTalkAPI.chm)

3 System Overview

Compact Talk is divided into two major parts, Compact Talk service and CTClient API as shown in Figure 1.

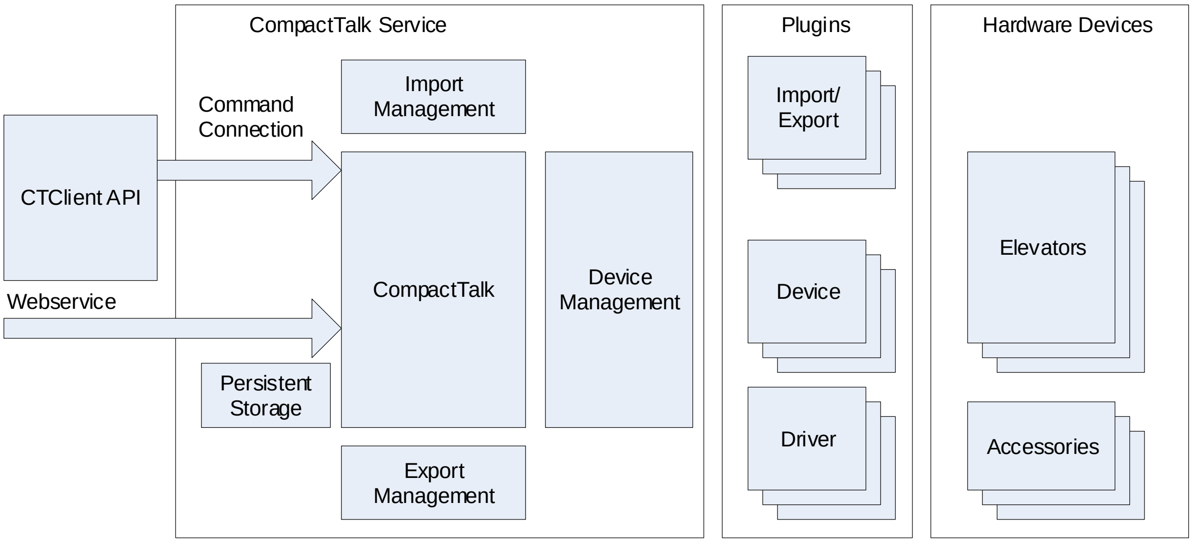


Figure 1, System Overview

3.1 Compact Talk Service

The service application handles the communication with third party implementations, order management, import and export management and persistent storage management. It is possible to access the Command connection directly using Windows Communication Foundation (WCF) but this document will not cover that method. It is also possible to access the CommandConnection via a HTTP as described in Section 9.

3.2 CTClient API

The client API covers connection management, error handling and synchronized event dispatching. This API is fully covered by this document. This API makes use of the Command connection.

3.3 Webservice

The webservice is a standard SOAP-based service accessible via HTTP.

3.4 Import and Export

These interfaces allow import of orders and export of selected order info triggered by status changes. A basic implementation of these interfaces that uses text files will be covered by this document.

4 Service Concept

Most active and replaceable components in Compact Talk are based on a class called BasicService or one of its sub classes. A service has a state that can be monitored and a few methods to control the state.
Services are handled by a service manager that stores all service references in a hierarchical structure. An example of a hierarchy that has two partitions, three elevators and one accessory per elevators is shown below.

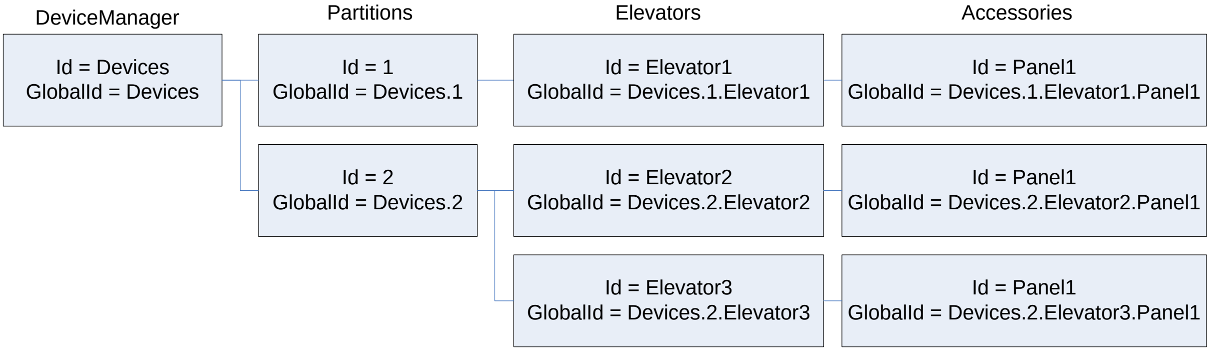

DeviceManager
Partitions
Elevators
Accessories
Id = Devices
GlobalId = Devices
Id = 1
GlobalId = Devices.1
Id = Elevator1
GlobalId = Devices.1.Elevator1
Id = Panel1
GlobalId = Devices.1.Elevator1.Panel1
Id = 2
GlobalId = Devices.2
Id = Elevator2
GlobalId = Devices.2.Elevator2
Id = Panel1
GlobalId = Devices.2.Elevator2.Panel1
Id = Elevator3
GlobalId = Devices.2.Elevator3
Id = Panel1
GlobalId = Devices.2.Elevator3.Panel1

Figure 2, Example of a service hierarchy

4.1 Service Types

There are five classes and an enumerated type involved in the service concept as shown in figure 3.

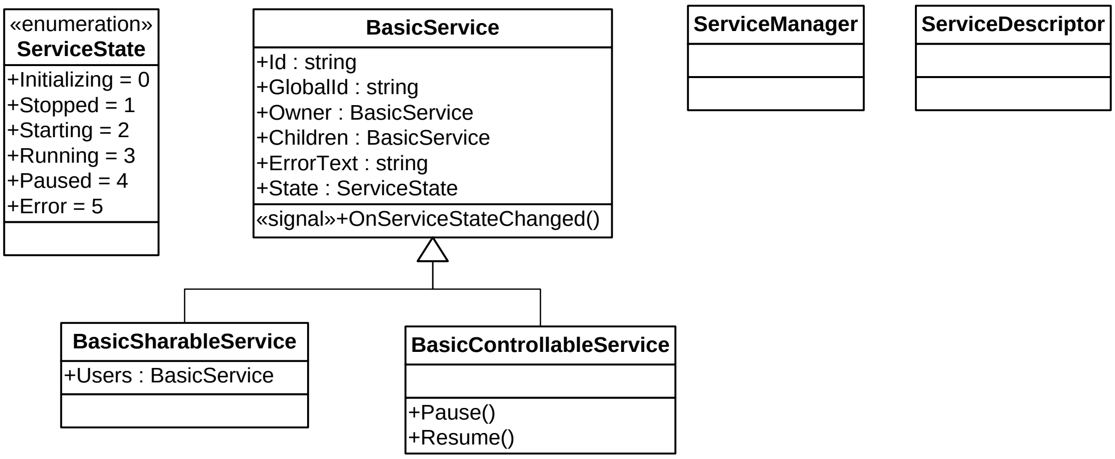


Figure 3, Service class hierarchy

4.1.1 BasicService

This type is the base class of the service types. It contains properties for identity, relations and state.
Communication drivers, for example, are based on this class. It also has an event handler that will fire an event each time the state is changed.

4.1.2 BasicControllableService

This class is a sub class of *BasicService* and it adds two methods to control the state of a service. Elevator devices, for example, are based on this class.

4.1.3 BasicSharableService

This type is also a sub class of *BasicService*. What's special with an instance of this class is that it can be referenced by multiple sources that share a single resource. The bus driver in the picture below is an example of a component that shares one resource. In this case it shares a multi drop modem to one or more elevator devices.

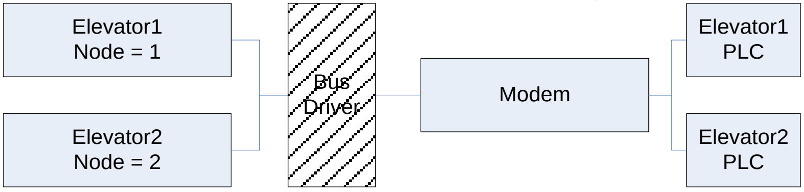


Figure 4, An example wich uses a BasicSharableService

4.1.4 ServiceManager

The service manager is a container for services and has methods for browsing the service hierarchy, retrieving current state of a service. It also has an event handler that will fire each time the state of a service is changed.

4.1.5 ServiceDescriptor

This class describes a loaded service in the server. For example elevator devices, accessories and ERP-plugins.

5 Basic Concepts

CT is using pick orders to order the elevators to retrieve trays to the operators. A pick order contains information about which elevator, tray and which service opening to send it to and information that can be displayed on accessories.

5.1 Order handling

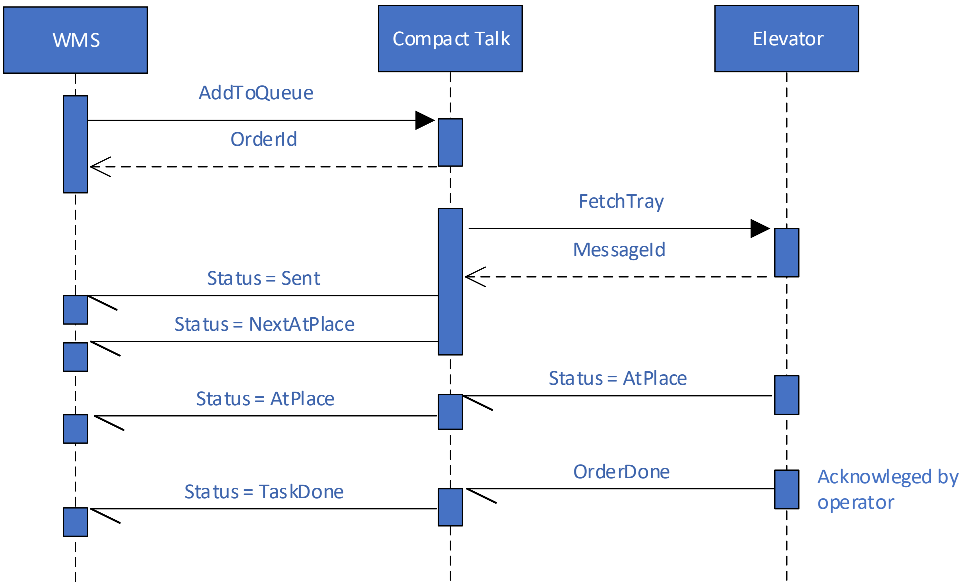
Pick orders are put on a queue before being dispatch the respective elevator, which means that a call to *AddToQueue* doesn't immediately put the order in the elevator, therefore it's important to keep track of the status on orders.
Orders are, after picking or placing is done, acknowledged either by the operator or externally by the client. The argument **noReturnOfTray** in the call *AddToQueue* controls whether it's allowed to acknowledge the order on the operator panel or not. If set to 1, *ExtAchOrder* needs to be called to return the tray.
The different statuses an order can have is listed below:

Status	**Description**
Posted	The order is passive and needs to be activated for it to be dispatched.
Selected	The order is active and are available for selection by the dispatcher.
Sent	The order has successfully been dispatched to the elevator device.
NextAtPlace	Indicates that the order is next in turn to enter an opening.
AtPlace	Indicates that the tray that was ordered is at picking position.
TaskDone	The order has been acknowledged by the operator or the client.
Statuses when external confirmation is used	
Accepted	The order has been confirmed by operator and client.
AcceptedStillAtPlace	The order has been confirmed by the operator and client but is waiting for external acknowledge from the client.
TaskDoneStillAtPlace	The order has been confirmed by the operator but is still not confirmed by the client.
Error status	
Refused	The order was rejected by the elevator device. The order is considered non active a can be deleted.

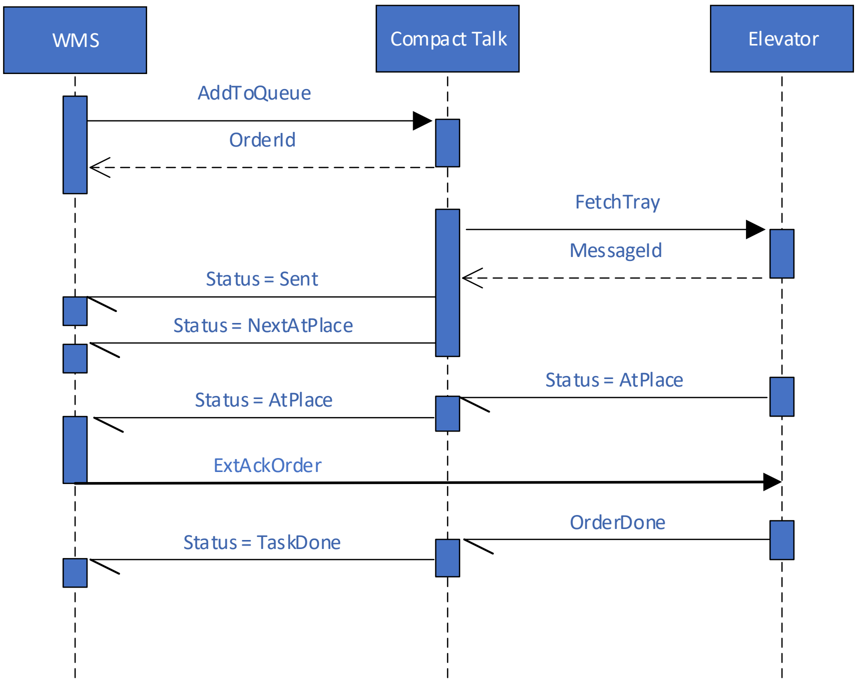
An example of the contents in the queue could look like below:

Order	Tray	Status
1	10	AtPlace
2	11	NextAtPlace
3	12	Sent
4	13	Selected

5.1.1 Status flow acknowledged by operator



5.1.2 Status flow acknowledged by client

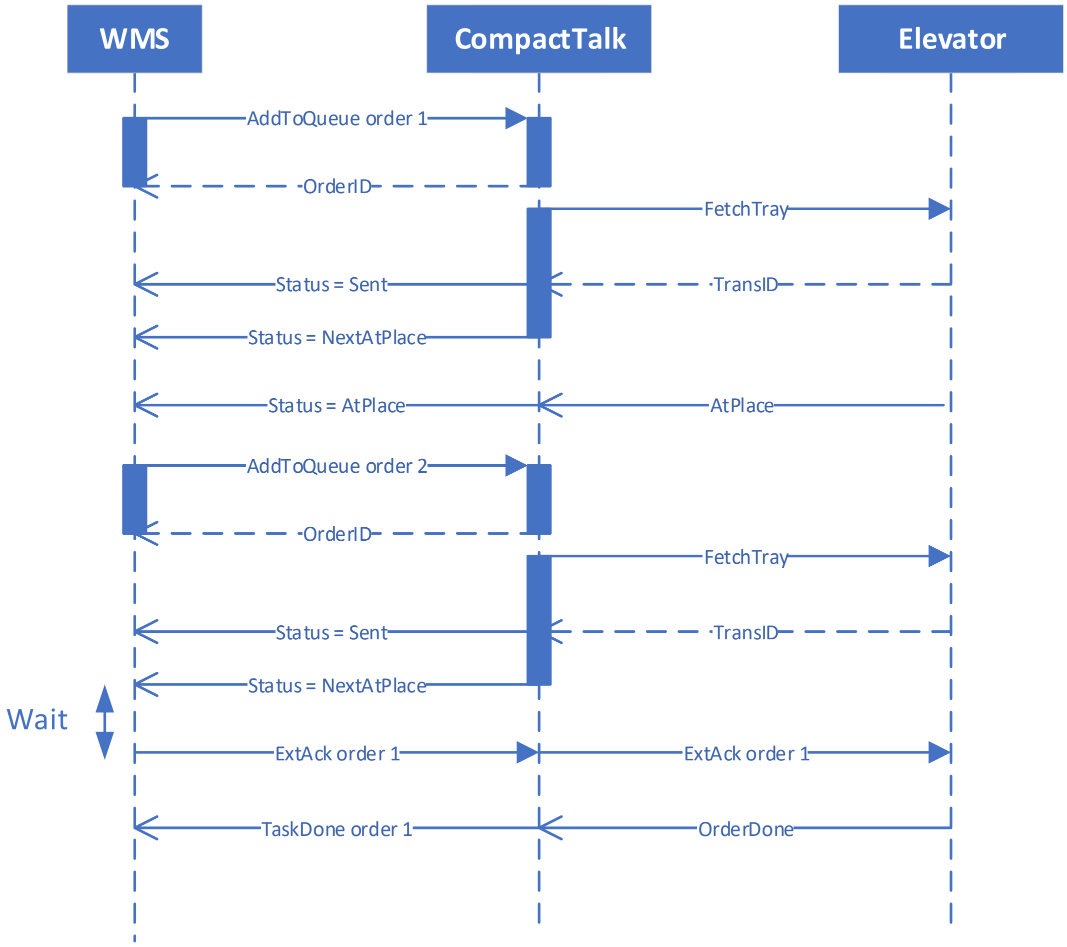


5.2 Event handling

The status changes of orders are dispatched as events to the client, the method differs depending on how you connect. CTClient API will do the work for you by polling the event queue and dispatch them as events to the application. When connecting to the WebService interface you have to explicitly poll for new events on the event queue.

Proper event handling is important because of timing issues caused by the fact that the order dispatcher doesn’t dispatch the order to the elevator immediately on the call to *AddToQueue.*

For example, let us say you only want one active order at a time but you don’t want the tray to be returned between orders if they target the same tray, then you have to make sure the second order have reached the elevator before acknowledging the first order, otherwise the elevator doesn’t know to let the tray stay at the opening. You do that by waiting for the status on the second order reaches **Sent** or higher. See diagram below:



5.3 Data Model

Description of data types and models

5.3.1 Tray

In an elevator there can be multiple trays. Each tray holds specific characteristic depending on the height of articles placed on the tray, and the total amount of weight it holds e.g.

Brick layout is not stored in Compact Talk and can only be gathered from WMS.

Field Name	**Type**	**Description**
Tray	object	Keeps information about a tray in the elevator
tray:Blocked	bool	If tray is blocked and cannot be used
tray:Height	int	Height of the tray in mm (occupied space in height).
tray:Level	int	Elevator **G1**: Represent Access Level Elevator **G2**: Represent position of tray from bottom limit position, to top height of current tray in mm.

tray:Weight	int	Weight of the tray in gram
tray:Id	int	Numerical id of tray
tray:Borrowed	bool	Indicate that a tray has been borrowed from the elevator and does not currently reside in the elevator.

6 Common programming cases

This section lists common integrations cases which covers the recommended minimum of functionality that should be implemented when integrating with Compact Talk. These cases are also covered by the sample application MiniWms.

6.1 External acknowledge

The external acknowledge method *ExtAckOrder* is useful when you want the WMS-system to be the one that have the last word before the tray is returned. To accomplish this you have to tell the elevator, when you add your order, that an external acknowledge is required before the tray is returned. You do that by setting the argument noReturnOfTray equal to 1 in the method *AddToQueue*. To be allowed to do the external acknowledge on the order it have to have a status which states that it’s on the table in the elevator.

6.2 Stock management

A simple implementation of stock management is shown in the piece of code below. When the event *CTOrderStatusChangeEvent* is received you can switch on the Mode property of the order tied to the event. The mode of the order is set when you call the *AddToQueue* method.

```
void OnOrderStatusChangedEvent(CTOrderStatusChangeEvent evt)
{
          if (evt.Order.Status == OrderStatus.TaskDone)
          {
                //Do stock management based on the OrderMode
                    if (ctOrder.Mode == OrderMode.OUT)
                        //Subtract the evt.Order.AckQuantity from the quantity on the article record
                    else if (ctOrder.Mode == OrderMode.IN)
                        //Add the evt.Order.AckQuantity to the quantity on your article record
                    else if (ctOrder.Mode == OrderMode.INV)
                        //Replace the quantity on your article record with evt.Order.AckQuantity
          }
}
```

6.3 Order synchronization

Order synchronization is useful when your system has been down while Compact Talk still had active orders on its queue. What could have happened during this time is that the operator acknowledged an order so you missed the event when the order status was set to *TaskDone*. This leads to problems with stock management since you haven’t been given the acknowledged quantity from the operator. The code below shows code and description of how to handle this problem.

```
public class Class1()
{
        //Keep e list of orders sent to Compact Talk
        List<OrderRecord> m_OrderCache = new List<OrderRecord>();
        void SynchronizeOrders()
        {
            for (int i = 0; i < m_OrderCache.Count; i++)
            {
                //Retrieve the order from Compact Talk that has the same orderId as your cached order
                //The GetOrder method will first look for the order in Compact Talk’s order queue, if
                //it’s not found there it will look in the table of historical order data
                PickOrder ctOrder = m_CompactTalk.Command.GetOrder(m_OrderCache[i].CTOrderId);
                if (ctOrder.Status == OrderStatus.Historical)
                {
                    //A historical order means that it has gone trough a full status cycle and
                    //now only exists as an historical record in the database.
                    if (ctOrder.Mode == OrderMode.OUT)
                        //Subtract the evt.Order.AckQuantity from the quantity on the article record
                    else if (ctOrder.Mode == OrderMode.IN)
                        //Add the evt.Order.AckQuantity to the quantity on your article record
                    else if (ctOrder.Mode == OrderMode.INV)
                        //Replace the quantity on your article record with evt.Order.AckQuantity
                    //Remove the order from the cache and do implementation specific maintenance
                }
                else
                {
                    //Just uppdate the status on your cached order here
                    m_OrderCache[i].CTStatus = ctOrder.Status.ToString();
                }
            }
        }
}
```

The method SynchronizeOrders in the class above should be called every time you start your system or after a reconnect to Compact Talk after a dropped connection.

6.4 Maintenance

To be able to perform the maintenance methods listed below the elevator/elevators needs to be in a paused state. Paused state means that the dispatching of new orders to the elevator/elevators is put on hold. The method to be used is called *PauseService*.

The maintenance methods have an argument called servicePath which is used to select on what level in the device hierarchy that is targeted. The servicePath can target all elevators, a partition or a specific elevator.

To target all elevators use the servicePath “Devices”, for a partition use “Devices.<partitionid>” and for a specific elevator you can use either “Devices.<partitionid>.<elevatorid> or the servicePath alias which equals the elevator id. So for an elevator with id “E1” on partition “P1” the servicePath could look like “Devices.P1.E1” or just “E1”.

When maintenance is done don’t forget to call *ResumeService*.

6.4.1 ReturnTrays

This method is used to abort all active orders in the elevator and force the elevator to return active trays to their storage position. The status of the active orders will be rolled back to “Selected” which means they are ready to be selected when the dispatching is resumed.

Two operations exist for returning trays, and will trigger return tray/s and reset orders.

- ReturnTrays, support in G1, G2, Sim
- ReturnTraysByOpening, support in G2

Note: Service must be paused before sending ‘ReturnTray’ command.

6.4.2 ClearOrderQueue

This method is used when you want to delete all but active orders from the elevators covered by the given servicePath. To delete all orders covered by the servicePath you need to abort the active orders by calling the method *ReturnTrays* described above.

For you who are porting your integration from CT v1 it's important to know that this method doesn't do all the things the old one did. To do what the old one did you need to first call *PauseService* then *ReturnTrays* followed by *ClearOrderQueue*. The reason for the call to ReturnTrays is to rewind the status on active orders to "Selected" since *ClearOrderQueue* will otherwise leave them on the queue.

7 Accessories and their configuration

There is a number of accessories available for the elevators like lightbar, laserpointer and pickdisplay. To make them work properly necessary configuration has to be added to Compact Talk in one way or another. There are two types of configurations depending of which type of accessory is being used, a single target box and a full tray layout which is composed of a list of boxes. The box also has two custom properties that can be used by the accessory.

The table below shows which configuration is needed for which accessory.

Accessory	**Single box**	**Full layout**
Lightbar	X (Property that gives the depth has to be provided)	X (The depth property is calculated)
Laserpointer	X	X
Pickdisplay		X

There are two different ways to add the necessary configuration and they are listed in the table below.

Type	**Method call**	**Order import**	**Tray import**
Single box	AddToQueueWithSingleBoxCoords	X	
Full layout	AddTrayConfig		X

NOTE: Article pictures sent to the accessory Pickdisplay must follow the following prerequisites: 1920x1080px or maximum 5MB/picture.

7.1 Method call

Method calls are done via the external interface and the methods are described in the API reference documentation [2].

The method *AddToQueueWithSingleBoxCoords* will add the order and the rectangle representing the target box in one call.

AddTrayConfig will add the full layout of a tray. This method can be called just before a call to *AddToQueue* to add the configuration for the targeted tray or it can be called multiple times on startup to add all configuration at once.

To be able to recover the configuration when Compact Talk doing a restart with unfinished orders in its queue it will store the configuration in a cache file and reload it from there on startup. Because of the cache tray configuration only needs to be added to Compact Talk a second time if it has changed.

7.2 Order import

How to add single box configuration with the order import method is described in section 11 and in the Configuration Manual [1].

7.3 Tray import

Adding full tray configuration via tray import is described in section 11.3.

8 CTCLIENT API

The client API [2] covers connection management, error handling and synchronized event dispatching. It is implemented as a .NET assembly named Weland.CompactTalk.Client.dll.
The purpose of the client API is to simplify the integration work for a third party implementer. The main class of the API is called *CTConnection*.

8.1 CTConnection

Properties:

- **Command (CommandProxy)**
  A reference to the proxy representing a connection to the command interface of the service.
- **IsConnected (bool)**
  True if a connection has been established otherwise false.

Methods:

- **void Connect(string host, int commandPort, int eventPort)**
  Connects to the server, via TCP/IP, at the given host and port numbers.
- **void Connect(string host)**
  Connects to the server, via TCP/IP, at the given host using the default port numbers.
- **void Connect()**
  Connects locally to the service on a named pipe.
- **void Disconnect()**
  Disconnects from the service.

Events:

- **event ClientQueueChanged OnQueueChanged**
  Is fired when an order is added or removed from the queue.
- **event ClientOrderStatusChanged OnOrderStatusChanged**
  Is fired when the status is changed on an order.
- **event ClientServiceStateChanged OnServiceStateChanged**
  Is fired when the state of a service has been changed.
- **event ClientAckRequest OnClientAckRequest**
  Is fired when external confirmation is activated (configurable) and an order

reached the status TaskDone. Client must respond with a call to the method ConfirmAckRequest, or the order will remain on the queue.

- **event ClientOrderPriorityChanged OnOrderPriorityChanged**
  Is fired when the priority of an order is changed.
- **event ClientOpeningModeChanged OnOpeningModeChanged**
  Is fired when the mode is changed on a specific opening.
- **event ClientOpeningUserChanged OnOpeningUserChanged**
  Is fired when the logged in user is changed on a specific opening.
- **event ClientTrayWeightChanged OnTrayWeightChanged**
  Is fired when the weight of a specific tray is changed.
- **event ClientTrayHeightChanged OnTrayHeightChanged**
  Is fired when the height of a specific tray is changed.
- **event ClientOrderReturned OnOrderReturned**
  Is fired when an order reached TaskDone and the tray is returned to its position in the elevator. If multiple orders are picked from the same tray, an event for each order will be sent. The event contains orderId.
- **event ClientStopCodesChanged OnStopCodesChanged**
  Is fired when the list of stop codes is changed. More information in section 8.2.

8.2 Stop Codes

8.2.1 General

Stop codes in Compact Talk is given as a history list with up to ten latest stop codes. The latest stop code is always stored in the first position in the list and the oldest in last position. The OnStopCodesChanged event is fired when the stop code list is changed. The event contains the new list with stop code information, List<StopCode>.

StopCode class:

```
public class StopCode
{
    //See stop code definition list for valid codes
    public int Code { get; set; }

    //Describes the cause of the stop
    public string Cause { get; set; }
}
```

8.2.2 Stop Code Definitions

Stop cause	**Stop code**
State change	1
Substate change	2
Machine restarted	3

Sequences reset	4
E-Stop	1000
Lightcurtain	1001
Pit sensor blocked	1010
Pit sensor blocked, fetch from opening	1011
Pit sensor blocked, fetch from storage	1012
Brake contactor 1 error	1020
Brake contactor 2 error	1021
Brake relay 1 error	1022
Brake relay 2 error	1023
Lost encoder module, upper level	1030
Lost encoder module, lower level	1031
Inverter position changed on reboot	1032
Inverter has lost reference position	1033
Inverter error code present	1034
Reference search failed, elevator level	1040
Invalid position, upper level	1041
Invalid position, lower level	1042
Elevator on lower limit switch	1050
Elevator on upper limit switch	1051
Position error elevator, not at tray storage position	1060
Position error elevator, not at opening	1061
Position error elevator, not in unhook interval	1062
Position error elevator, not at unhook position	1063
Position error elevator, not at temporary storage	1064
Position error elevator, not in unhook interval of temporary storage	1065
Position error elevator, not at unhook of temporary storage	1066
Position error upper level	1070
Position error lower level	1071
Weight measurement error	1080
Tray returned to opening, too heavy	1081
Heightmeter error	1090
Tray too high for limited height opening	1091
Tray returned to opening, too high	1092

Tray fetch error opening, ghost	1100
Tray fetch error at storage, ghost	1101
Already a tray in opening, can't move tray to opening	1110
Already a tray in opening, twin shift, can't move tray to opening	1111
Level position blocks elevator movement, not free	1120
Lower level not on sensors, can't fetch from opening twin w/o extractor	1121
Upper level not on sensors, can't fetch from opening twin w/o extractor	1122
No level at pick position, can't fetch from opening twin w/o extractor	1123
Both levels not on sensors, can't fetch from storage	1124
Level not on sensors, can't fetch from storage	1125
Extractor or door blocks elevator movement	1126
Levels not on sensors, can't load tray	1127
Levels not on sensors, can't unload tray	1128
Tray in pick position does not match upper level tray or step	1130
Tray in pick position does not match lower level tray or step	1131
Tray on lower level does not match NextTray	1132
Tray on upper level does not match NextTray	1133
Tray cannot be reached by either level	1140
Tray cannot be reached	1141
No storage position set for requested tray	1142
Invalid tray position	1150
Invalid temporary tray position	1151
Invalid tray position, fetch next sequence reset	1152
Invalid tray unhook position	1153
Invalid tray unhook position, fetch next sequence reset	1154
Invalid tray position, can't twin shift at storage	1155
Invalid tray unhook position, can't twin shift at storage	1156
Invalid tray unhook position, temporary storage	1157
Invalid tray unhook position, fetch sequence reset	1158
Invalid tray position, fetch sequence reset	1159
Requested tray is blocked	1160
User has no access right to requested tray	1161
User has no access right to requested NextTray	1162
No temporary storage found	1170
Storage blocked by temporary tray	1171
No storage to return temporary tray, optimisation	1172

Inverter movement physically blocked	1180
Air flow (ATEX)	2000
Gas warning (ATEX)	2001
Gas alarm (ATEX)	2002
Fire alarm	2010
Sprinkler zone	2011
Borrow tray trolley not removed	2020
Borrow tray trolley not in position	2021
Tray already on borrow tray trolley	2022
No tray on borrow tray trolley	2023
Requested tray is borrowed	2024
Requested tray is not borrowed	2025
Load beam not installed	2030
Tray not correctly placed on opening sensors	2040
Tray fetch error operation station	2050
Tray placement error operation station	2051
Position error elevator at operation station	2052
Tray already in operation station	2053
Level not on sensors, can't fetch from operation station	2054
No tray in operation station	2055
Tray fetch error transfer station	2060
Tray placement error transfer station	2061
Position error elevator at transfer station	2062
Tray already in transfer station	2063
Chain and telescope not in inner position	2064
Requested tray is not transferred	2065
No tray in transfer station	2066
Tray in transfer station not correctly placed	2067
No storage found for tray to be transferred in	2068
Level not on sensors, can't fetch from transfer station	2069
UPS, on battery	2080
Tray fixture not locked	2090
Side table order mismatch	2100
Side table chain move to elev disallowed	2101
Side table chain move to pick disallowed	2102

Side table screw move to elev disallowed	2103
Side table screw move to pick disallowed	2104
Side table screw move to service disallowed	2105
Machine zone not enabled	2120

8.2 Code Samples

All the code samples are available in the SDK release of Compact Talk.
To be able to build a client application these assemblies needs to be referenced:

- Weland.CompactTalk.Framework.dll
- Weland.CompactTalk.Client.dll
- System.ServiceModel

There is no error handling in these samples.

8.2.1 Sample 1

This is a simple implementation that connects and retrieves the version of the Compact Talk service.

```
CTConnection connection = new CTConnection();
connection.Connect();
Version version = connection.Command.Version;
connection.Disconnect();
```

8.2.2 Sample 2

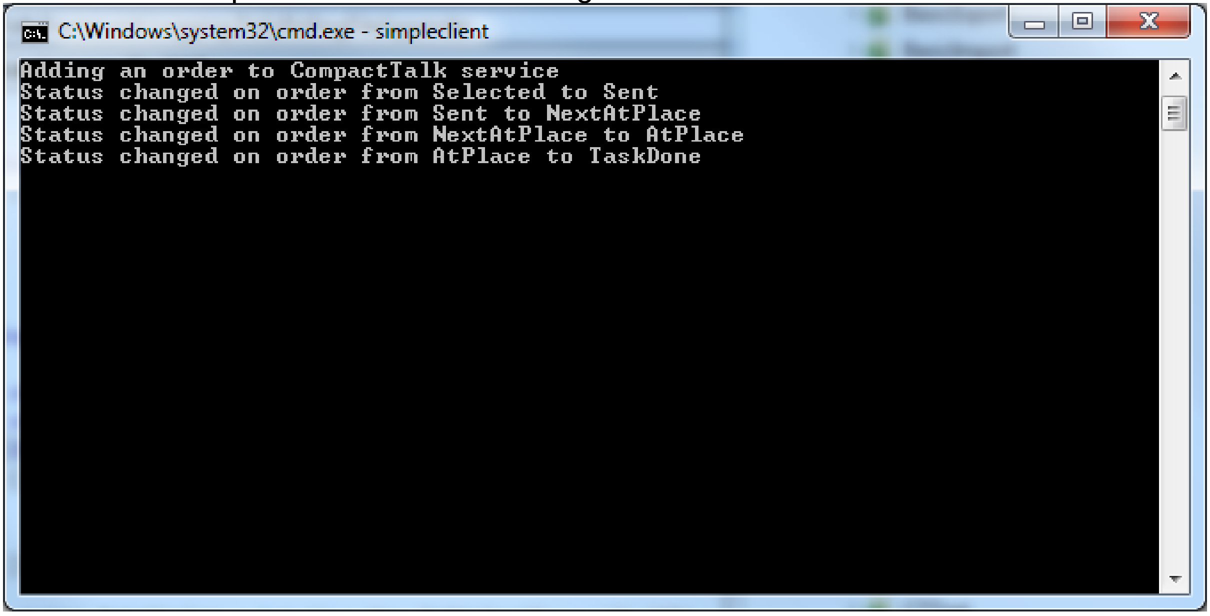
The second sample is a little more complete client that adds a pick order to the service and prints the status changes while it is executed.

```
using System.Text;
using Weland.CompactTalk.Client;
using Weland.CompactTalk.Framework.OrderManagement;
using Weland.CompactTalk;

namespace SimpleClient
{
    class Program
    {
        static void Main(string[] args)
        {
            //Create an instance of the CTConnect type.
            CTConnection con = new CTConnection(null);

            //Connect to service.
            con.Connect();
            //Hook up the order status event.
            con.OnOrderStatusChanged += new ClientOrderStatusChanged(OnClientOrderStatusChanged);
            Console.WriteLine("Adding an order to Compact Talk service");
            //Add an order to the service
            con.Command.AddToQueue("SimpleClient:1", "Sim_1", 1, "", 1, "", "",
                OrderMode.OUT, 0, 1, "", 100, "", "", "", "", "", "", true);
            //Wait for input to terminate the program.
            Console.ReadKey();
            //Disconnect from the service.
            con.Disconnect();
        }
        //This method is called every time the status is changed on the order.
        static void OnClientOrderStatusChanged(CTOrderStatusChangeEvent evt)
        {
            Console.WriteLine("Status changed on order from " + evt.OldStatus.ToString()
                + " to " + evt.Order.Status.ToString());
        }
    }
}
```

The console output looks like shown in figure 5.



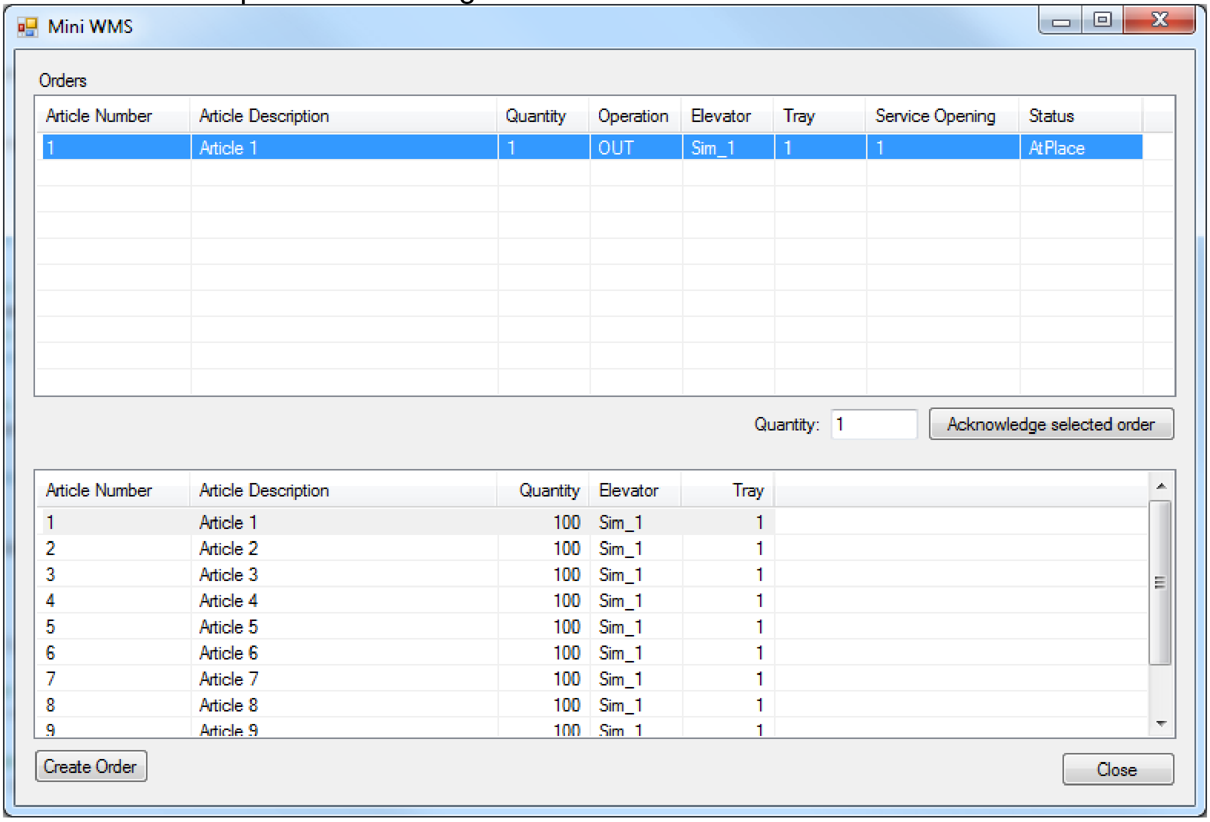
```
Adding an order to CompactTalk service
Status changed on order from Selected to Sent
Status changed on order from Sent to NextAtPlace
Status changed on order from NextAtPlace to AtPlace
Status changed on order from AtPlace to TaskDone
```

Figure 5, Console output from sample 1

8.2.3 A more complete sample, Mini WMS

This is a more complete sample application the covers most of the cases. It’s a simple WMS application that is capable of creating pick orders based on a list of available article records.

A file is used for persistent storage of the order list instead of a database.



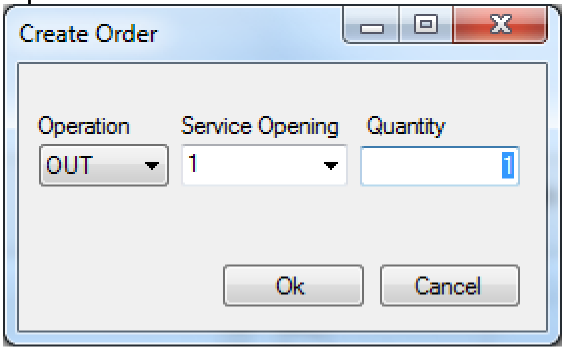
Article Number	Article Description	Quantity	Operation	Elevator	Tray	Service Opening	Status
1	Article 1	1	OUT	Sim_1	1	1	AtPlace

Quantity: 1

Article Number	Article Description	Quantity	Elevator	Tray
1	Article 1	100	Sim_1	1
2	Article 2	100	Sim_1	1
3	Article 3	100	Sim_1	1
4	Article 4	100	Sim_1	1
5	Article 5	100	Sim_1	1
6	Article 6	100	Sim_1	1
7	Article 7	100	Sim_1	1
8	Article 8	100	Sim_1	1
9	Article 9	100	Sim_1	1

Figure 6, MiniWMS main window

The main application window contains an order list, an article list, a button to create a pick order and a button to do an external acknowledge of an order.



Operation: OUT
Service Opening: 1
Quantity: 1

Figure 7, Dialog to enter pick order information

The Create Order dialog is used to enter information needed to create the pick order. The complete code listing of the main form of the application is available below. There is a lot of code related to presentation of information in the listing, but the comments should help to isolate the functional bits.

```
public partial class MainForm : Form
    {
        //List of orders sent to Compact Talk
        List<OrderRecord> m_MyOrders = new List<OrderRecord>();
        //Index of orders sent to Compact Talk
        Dictionary<int, OrderRecord> m_MyOrdersByCTId = new Dictionary<int, OrderRecord>();
        //List of article records
        List<ArticleRecord> m_MyArticleRecords = new List<ArticleRecord>();
        //Index of articles
        Dictionary<int, ArticleRecord> m_MyArticleRecordsById = new Dictionary<int, ArticleRecord>();
```

```csharp
//Client api connection
CTConnection m_CompactTalk;
public MainForm()
{
    InitializeComponent();
    //Disable acknowledge button until we have a selected row in order view
    buttonExtAck.Enabled = false;
    //Generate 10 articles and add them to the list and index
    for (int i = 1; i <= 10; i++)
    {
        ArticleRecord ar = new ArticleRecord();
        ar.ArticleNo = i.ToString();
        ar.ArticleDesc = "Article " + i.ToString();
        ar.Elevator = "Sim_1";
        ar.Id = i;
        ar.Quantity = 100;
        ar.TrayNo = 1;

        m_MyArticleRecords.Add(ar);
        m_MyArticleRecordsById.Add(ar.Id, ar);
    }
    //Populate article view with the available article records
    for (int i = 0; i < m_MyArticleRecords.Count; i++)
    {
        ListViewItem lvItem = listViewArticles.Items.Add(m_MyArticleRecords[i].ArticleNo);
        lvItem.SubItems.Add(m_MyArticleRecords[i].ArticleDesc);
        lvItem.SubItems.Add(m_MyArticleRecords[i].Quantity.ToString());
        lvItem.SubItems.Add(m_MyArticleRecords[i].Elevator);
        lvItem.SubItems.Add(m_MyArticleRecords[i].TrayNo.ToString());
        lvItem.Tag = m_MyArticleRecords[i].Id;
    }
    //Load our orders from persistent storage
    LoadOrders();
    //Connect to Compact Talk
    try
    {
        m_CompactTalk = new CTConnection(this);
        m_CompactTalk.OnOrderStatusChanged += new
        ClientOrderStatusChanged(OnOrderStatusChangedEvent);
        m_CompactTalk.OnQueueChanged += new ClientQueueChanged(OnQueueChangedEvent);
        m_CompactTalk.Connect();
    }
    catch (Exception e)
    {
        MessageBox.Show("Failed to connect to Compact Talk");
        throw e;
    }
    //Do a recovery in case orders been acknowledge while this application was shut down
    SynchronizeOrders();
}
void LoadOrders()
{
    //Deserialize order list from file storage
    using (Stream file = File.Open("MyOrders.dat", FileMode.OpenOrCreate))
    {
        BinaryFormatter bformatter = new BinaryFormatter();
        if (file.Length > 0)
            m_MyOrders = (List<OrderRecord>)bformatter.Deserialize(file);
    }
    //Populate order view with available orders
    for (int i = 0; i < m_MyOrders.Count; i++)
    {
        ListViewItem lvItem = listViewOrders.Items.Add(m_MyOrders[i].ArticleNo);
        lvItem.SubItems.Add(m_MyOrders[i].ArticleDesc);
        lvItem.SubItems.Add(m_MyOrders[i].Quantity.ToString());
        lvItem.SubItems.Add(m_MyOrders[i].Operation);
        lvItem.SubItems.Add(m_MyOrders[i].Elevator);
        lvItem.SubItems.Add(m_MyOrders[i].TrayNo.ToString());
        lvItem.SubItems.Add(m_MyOrders[i].ServiceOpening.ToString());
        lvItem.SubItems.Add(m_MyOrders[i].CTStatus);
```

```
                lvItem.Tag = m_MyOrders[i].CTOrderId;
                //Add the order to the order index
                m_MyOrdersByCTId.Add(m_MyOrders[i].CTOrderId, m_MyOrders[i]);
            }
        }
        void SaveOrders()
        {
            //Serialize order list to file storage
            using (Stream orderFile = File.Open("MyOrders.dat", FileMode.OpenOrCreate))
            {
                BinaryFormatter bformatter = new BinaryFormatter();
                bformatter.Serialize(orderFile, m_MyOrders);
            }
        }
        void SynchronizeOrders()
        {
            //For each order in our order list, check to se if it matches Compact Talks order
            for (int i = 0; i < m_MyOrders.Count; i++)
            {
                PickOrder ctOrder = m_CompactTalk.Command.GetOrder(m_MyOrders[i].CTOrderId);
                if (ctOrder == null)
                {
                    //Compakt Talk has never seen an order with this id. This should never happen.
                    continue;
                }
                //The order has been acknowledged while we where shut down and now only exist in 
                historical storage
                if (ctOrder.Status == OrderStatus.Historical)
                {
                    //Do stock management based on the OrderMode
                    if (ctOrder.Mode == OrderMode.OUT)
                        m_MyArticleRecordsById[m_MyOrders[i].ArticleId].Quantity
                        -= ctOrder.AckQuantity;
                    else if (ctOrder.Mode == OrderMode.IN)
                        m_MyArticleRecordsById[m_MyOrders[i].ArticleId].Quantity += ctOrder.AckQuantit
y;
                    else if (ctOrder.Mode == OrderMode.INV)
                        m_MyArticleRecordsById[m_MyOrders[i].ArticleId].Quantity = ctOrder.AckQuantity
;
                    //Locate the article in the article view and update quantity so we doesn't present
 stale data
                    for (int j = 0; j < listViewArticles.Items.Count; j++)
                    {
                        if (m_MyOrders[i].ArticleId == (int)listViewArticles.Items[j].Tag)
                        {
                            listViewArticles.Items[j].SubItems[2].Text = m_MyArticleRecordsById[m_MyOr
ders[i].ArticleId].Quantity.ToString();
                            break;
                        }
                    }
                    //Locate the order in the order view and remove it
                    for (int j = 0; j < listViewOrders.Items.Count; j++)
                    {
                        if (m_MyOrders[i].CTOrderId == (int)listViewOrders.Items[j].Tag)
                        {
                            listViewOrders.Items.RemoveAt(j);
                            break;
                        }
                    }
                    //Remove the order from the order list and order index
                    m_MyOrdersByCTId.Remove(m_MyOrders[i].CTOrderId);
                    m_MyOrders.Remove(m_MyOrders[i]);

                    i -= 1;
                }
                else
                {
                    //Update the status of the order in the order view and in the order list
                    listViewOrders.Items[i].SubItems[7].Text = ctOrder.Status.ToString();
                    m_MyOrders[i].CTStatus = ctOrder.Status.ToString();
                }
            }
            //Save changes to the order list in the storage file
            SaveOrders();
        }
        void OnOrderStatusChangedEvent(CTOrderStatusChangeEvent evt)
        {
            //Check to see if it's one of our orders we received the event for, if not just ignore the
            event
            if (!m_MyOrdersByCTId.ContainsKey(evt.Order.Id))
                return;
            OrderRecord order = m_MyOrdersByCTId[evt.Order.Id];
            //Search for the order in the order view
            for (int i = 0; i < listViewOrders.Items.Count; i++)
            {
                if (order.CTOrderId == (int)listViewOrders.Items[i].Tag)
                {
                    //Update the order in the order view and in the order list with the new status
                    listViewOrders.Items[i].SubItems[7].Text = evt.Order.Status.ToString();
                    order.CTStatus = evt.Order.Status.ToString();

                    //If the new status is TaskDone
                    if (evt.Order.Status == OrderStatus.TaskDone)
                    {
                        //Do stock management based on the OrderMode
                        if (evt.Order.Mode == OrderMode.OUT)
                            m_MyArticleRecordsById[order.ArticleId].Quantity -= evt.Order.AckQuantity;
                        else if (evt.Order.Mode == OrderMode.IN)
                            m_MyArticleRecordsById[order.ArticleId].Quantity += evt.Order.AckQuantity;
                        else if (evt.Order.Mode == OrderMode.INV)
                            m_MyArticleRecordsById[order.ArticleId].Quantity = evt.Order.AckQuantity;
                        //Locate the article in the article view and update quantity so we doesn't
                        present stale data
                        for (int j = 0; j < listViewArticles.Items.Count; j++)
                        {
                            if (order.ArticleId == (int)listViewArticles.Items[j].Tag)
                            {
                                listViewArticles.Items[j].SubItems[2].Text = m_MyArticleRecordsById[
                                order.ArticleId].Quantity.ToString();
                            }
                        }
                    }
                    //Save changes to the order list in the storage file
                    SaveOrders();
                    break;
                }
            }
        }
        void OnQueueChangedEvent(CTQueueChangeEvent evt)
        {
            //Check to see if it's one of our orders we received the event for, if not just ignore the
            event
            if (!m_MyOrdersByCTId.ContainsKey(evt.Order.Id))
                return;
            OrderRecord order = m_MyOrdersByCTId[evt.Order.Id];
            //If the queue chane type is OrderDeleted
            if (evt.ChangeType == OrderQueueChangeType.OrderDeleted)
            {
                //Search the order view for the order
                for (int i = 0; i < listViewOrders.Items.Count; i++)
                {
                    if (order.CTOrderId == (int)listViewOrders.Items[i].Tag)
                    {
                        //Remove the order from the order view, the order list and the order index
                        listViewOrders.Items.RemoveAt(i);
                        m_MyOrdersByCTId.Remove(order.CTOrderId);
                        for (int j = 0; j < m_MyOrders.Count; j++)
                        {
                            if (m_MyOrders[j].CTOrderId == order.Id)
                            {
                                m_MyOrders.RemoveAt(j);
                                break;
                            }
                        }
                        //Save changes to the order list in the storage file
                        SaveOrders();
                        break;
                    }
                }
            }
        }
        private void buttonCreateOrder_Click(object sender, EventArgs e)
        {
            CreateOrderDlg dlg = new CreateOrderDlg();
            //Open the CreateOrderDlg to enter the pick order information
            if (dlg.ShowDialog() == System.Windows.Forms.DialogResult.OK)
            {
                ArticleRecord ar = m_MyArticleRecordsById[(int)listViewArticles.SelectedItems[0].Tag];
                //Create an order record
                OrderRecord order = new OrderRecord();
                order.ArticleId = ar.Id;
                order.ArticleNo = ar.ArticleNo;
                order.ArticleDesc = ar.ArticleDesc;
                order.Elevator = ar.Elevator;
                order.TrayNo = ar.TrayNo;
                order.Quantity = dlg.Quantity;
                order.ServiceOpening = dlg.ServiceOpening;
                order.Operation = dlg.Operation;
                try
                {
                    //Call Compact Talk to add the order to the queue
                    int orderId = m_CompactTalk.Command.AddToQueue(
                        "MiniWMS:" + order.Id,
                        order.Elevator,
                        order.TrayNo,
                        "",
                        order.ServiceOpening,
                        order.ArticleNo,
                        order.ArticleDesc,
                        (OrderMode)Enum.Parse(typeof(OrderMode), order.Operation),
                        0,
                        1,
                        "",
                        order.Quantity,
                        "",
                        "",
                        "",
                        "",
                        "",
                        "",
                        true);
                    order.CTOrderId = orderId;
                    order.Id = orderId;
                    order.CTStatus = "Selected";
                }
                catch (Exception ex)
                {
                    MessageBox.Show("Failed to add order to Compact Talk\n\n" + ex.Message);
                    return;
                }
                //Add the order to the order list and index
                m_MyOrders.Add(order);
                m_MyOrdersByCTId.Add(order.Id, order);
                //Save changes to the order list in the storage file
                SaveOrders();
                //Add a new ro to the order view
                ListViewItem lvItem = listViewOrders.Items.Add(order.ArticleNo);
                lvItem.SubItems.Add(order.ArticleDesc);
                lvItem.SubItems.Add(order.Quantity.ToString());
                lvItem.SubItems.Add(order.Operation);
```

```
                lvItem.SubItems.Add(order.Elevator);
                lvItem.SubItems.Add(order.TrayNo.ToString());
                lvItem.SubItems.Add(order.ServiceOpening.ToString());
                lvItem.SubItems.Add(order.CTStatus);
                lvItem.Tag = order.CTOrderId;
            }
        }
        private void buttonClose_Click(object sender, EventArgs e)
        {
            //Just close the window so the application can exit
            Close();
        }
        private void MainForm_FormClosing(object sender, FormClosingEventArgs e)
        {
            //Disconnect from Compact Talk
            m_CompactTalk.Disconnect();
        }
        private void listViewOrders_SelectedIndexChanged(object sender, EventArgs e)
        {
            //If no order selected in the order view, disable the ack button and empty the quantity
            field
            if (listViewOrders.SelectedIndices.Count < 1)
            {
                buttonExtAck.Enabled = false;
                textBoxQuantity.Text = "";
                return;
            }

            //If an order is selected in the order view, enable the ack button and
            //fill the quantity field with the quantity of the order
            buttonExtAck.Enabled = true;
            int orderId = (int)listViewOrders.SelectedItems[0].Tag;
            textBoxQuantity.Text = m_MyOrdersByCTId[orderId].Quantity.ToString();
        }
        private void buttonExtAck_Click(object sender, EventArgs e)
        {
            int orderId = (int)listViewOrders.SelectedItems[0].Tag;
            OrderRecord selectedOrder = m_MyOrdersByCTId[orderId];
            try
            {
                //Call Compact Talks ExtAckOrder method
                m_CompactTalk.Command.ExtAckOrder(
                    selectedOrder.Elevator,
                    selectedOrder.ServiceOpening,
                    float.Parse(textBoxQuantity.Text),
                    false
                    );
            }
            catch (Exception ex)
            {
                MessageBox.Show("Failed to acknowledge order. Error: " + ex.Message);
            }
        }
    }
```

Mini WMS sample are using a file to store its order list instead of a database. It’s important that the orders are persisted to external storage so it’s possible to synchronize orders during recovery.

8.2.4 How to get Tray example

Here is an example how to get the Elevator information and also the trays found in that Elevator.

```
using System;
using Weland.CompactTalk;
using Weland.CompactTalk.Client;
using Weland.CompactTalk.Framework.Devices;
using Weland.CompactTalk.Framework.OrderManagement;
namespace SimpleClient
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            //Create an instance of the CTConnect type.
            CTConnection con = new CTConnection(null);
            //Connect to service a storageAddress (other then localhost)
            con.Connect(storageAddress);
            if (con.Connected)
            {
                //Hook up the order status event.
                con.OnOrderStatusChanged += new
                ClientOrderStatusChanged(OnClientOrderStatusChanged);
                Console.WriteLine("Adding an order to CompactTalk service");
                //Add an order to the service
                con.Command.AddToQueue("SimpleClient:1", "Sim_1", 1, "", 1, "", "",
                    OrderMode.OUT, 0, 1, "", 100, "", "", "", "", "", "", true);
                // Get Elevator Information position 0
                ElevatorInfo elevatorInfo = con.Command.GetElevatorInfo()[0];
                // Get all Trays from that Elevator Id
                Tray[] trays = con.Command.GetTrays(elevatorInfo.Id);
                //Wait for input to terminate the program.
                Console.ReadKey();
                //Disconnect from the service.
                con.Disconnect();
            }
        }
        //This method is called every time the status is changed on the order.
        private static void OnClientOrderStatusChanged(CTOrderStatusChangeEvent evt)
        {
            Console.WriteLine("Status changed on order from " +
            evt.OldStatus.ToString()
            + " to " + evt.Order.Status.ToString());
        }
    }
}
```

9 Web Service

Compact Talk has a built in web service which is compatible with most client platforms. Compact Talk is listening on the endpoint http://<hostname>:20012/CommandConnection.
WSDL information can be retreived on the url http://<hostname>:20012/CommandConnection?wsdl.
The web service supports SOAP 1.1 and follows WS-I BP 1.1. IIS does not have to be installed.
This interface requires manual polling for event by use of the methods ActivateEvents, ReActivateEvents, PollForEvent and PollForEvents.

9.1 Converting a CT 1.x client to 2.x

The major changes in the new interface are that transactions are now called orders which are represented by the type *PickOrder*.
Deviceid is now called compatibilityid. The id used in the new interface was called “name” in the old interface.
Event handling has also changed to a model where one must explicitly active an event queue for events to be produced.

9.1.1 New event handling

To get events to be created by Compact Talk you need to create an event queue by calling *ActivateEvents* or *ReActivateEvents* depending on the implementation of the client or situation.
ActivateEvents will create a queue and return unique id of that queue.
ActivateEvents should only be called once for each client session.
ReActivateEvents will either reactivate or create a queue with the given id.
The queue that is created will have a keep alive timer, currently set to 15 minutes, which is reset each time a call to one of the polling methods is made. When the timer elapses the queue will be destroyed to prevent memory leaks.
The polling of events is done using either *PollForEvent* or *PollForEvents* given the unique id of the queue.

9.1.2 Method translation table

The following list table changes in available methods and their usage.

Old method	New method	Description

`AddToQueue`	`[AddTrayConfig]`
	`AddToQueue`
		Tray configuration is no longer added with the AddToQueue method call.

`AddToQueue +`
`SetTransBoxCoordinates`
	`AddToQueueWithSingleBoxCoords`	This combination of methods is normally used when accessories like lightbar or laserpointer is used. The old method is error prone.

Init		Does not exist in the new interface

Start	`ResumeService("Devices")`	A better solution is to target individual elevators instead of all of them.
		Exampel:
		`ResumeService(“H1”)`
		`Will only resume the elevator with id “H1”.`

Stop (bool returnTrays)	PauseService("Devices") [ReturnTrays("Devices")]	Same advice as the method above.

ClearTransQueue	PauseService("Devices")
	ReturnTrays("Devices")
	ClearOrderQueue("Devices")
	ResumeService(“Devices”)
		Same advice as the method above.

DeleteTransaction	DeleteOrdersByCondition	This method is tricky to use because the format of the condition depends on the storage provider. Some properties have been renamed like "ELEVATORNAME" is now "ELEVATOR".
GetSize	GetOrderCount	
Sort		Is not supported by the new interface.
TransActivate	ActivateOrder	
GetAllTransactions	GetOrders	
GetTransactionAtOpening	GetOrderAtOpening	This method should not be used to monitor the order flow. Events that signal status changes on the order are the correct way.
ConfirmAckTransaction	ConfirmOrderAck	This method is only useful when confirmation of orders have been activated in the configuration to emulate the old behavior.
ExtAckTransaction	ExtAckOrder	
AreAllDevicesReadyForRunning		Not supported in the new interface.
SetLoggMask ReadLoggMask	SetLogThreshold	
GetDeviceVersion GetDeviceSignature	GetElevatorInfo GetSpecificElevatorInfo(string elevatorId)	In the new interface an object of the type ElevatorInfo is retrieved which contains all

		properties of the elevator.
GetMaxTrays	GetMaxTrayCount	
GetTrayStatus	GetTray	If GetTray returns an object of the type Tray if the tray exists otherwise null.
GetTrayBlockedStatus GetTrayLevel GetTrayWeight GetTrayHeight	GetTray	In the new interface an object of the type Tray is retrieved which contains all properties of the tray.
SetTraySize		Not supported in the new interface.
GetAllDevices	GetElevatorInfo	
GetDetailedDeviceInfo	GetSpecificElevatorInfo	ElevatorInfo contains less than the old DeviceInfoDetailed type did. See the documentation of ElevatorInfo in CompactTalkAPI.chm for more information.
GetField	GetOrder	Retrieve an object of the type PickOrder to get all the properties of the order.
WaitForEvent		Is not supported in the new interface.

Old event	New event	Description
CTAckEventArgs	CTAckRequestEvent	
CTDeviceEventArgs	CTServiceStateChangedEvent	
CTStatusEventArgs	CTOrderStatusChangeEvent	
CTElevatorStatusEventArgs	CTOpeningModeChangedEvent CTOpeningUserChangedEvent	

10 XML Interface

XML-interface allows method calls via XML files. When using this interface Compact Talk is forced to use a single order mode which means that it only accepts one order

at the time per elevator and opening. Compact Talk will auto acknowledge if there is an order with the status AtPlace in the queue before it adds a new order.
The documentation for this interface begins with an overview over the available methods and then continues with more details for the methods and some examples how they can be used.

10.1 Overview

The available Commands/Methods for the XML Interface are the following:

Command	**Description**
AddToQueue	AddToQueue is used to put orders in queue to Compact Talk.
ExtAckOrder	ExtAckOrder is used to externally acknowledge/confirm an order. Corresponding functionality can be done by using AddToQueue in combination with tray = 0
AddTrayConfig	Has to be used when an accessory is used that needs the full layout out of the tray. (Example of such accessory is the Picking Display)
ResetElevator	This method is used to abort all active orders in the elevator and force the elevator to return active trays. Corresponding functionality can also done by using AddToQueue in combination with tray=1000, and a parameter for the specific opening.

10.2 Commands

10.2.1 AddToQueue method

AddToQueue creates a new order in Compact Talk queue. All order received are executed in the sequence they are retrieved.
List of available parameters when using AddToQueue

Parameter	**Mandatory**	**Description**
TransId	Yes/No	If response messages is used this element is mandatory. It is used to tie the command and response together.

ElevatorId	Yes	Identity of the elevator that is the target for the order. Must be the id that is set in the device configuration.
Tray	Yes	The identity of the tray to be fetched. Must be an existing tray in the machine. Note! There is also two reserved tray numbers which can be used for special functionality If tray = 0 is used Compact Talk will handle it as an ExtAckOrder method. If tray = 1000 is used Compact Talk will handle it as a ResetElevator method.
Opening	Yes	Number of the service opening on the elevator. Can be 1,2 or 3 depending on how many openings the machine has. Note! There is also a special opening number “99” which can be used to perform ResetElevator functionality.
NoReturnOfTray	No	Instructs the elevator what to do with the tray when the operator acknowledge on the panel. If set to 0 = Tray return when order is confirmed at panel If set to 1 = Tray is not returned when order is confirmed at panel, waits for an external acknowledgement from the WMS. If the value is not specified the default value of 1 will be used
NextTray	No	Specifies the identity of the next tray (1-n). Has to be used if the Twin functionality is desired. Default value is set to = 0
ArtNo	No	Article number. Displayed on the panel and the picking display accessory to guide the operator.
ArtDescr	No	Article description. Displayed on the panel and the picking display accessory to guide the operator.
Quantity	No	The quantity for the order. Displayed on the panel and the picking display accessory to guide the operator.
Extra information for accessories		
Info1	No	Additional order information displayed on the picking display.

Info2	No	Additional order information displayed on the picking display.
Info3	No	Additional order information displayed on the picking display.
Info4	No	Additional order information displayed on the picking display.
Info5	No	Additional order information displayed on the panel and picking display.
CurrentBoxName	Yes if PickDisplay else No	Location on tray. Used by picking display.
Mode	Yes if PickDisplay else No	This value has no logical functionality in Compact Talk. However it is used to show the operator which operation should be done. Depending on the value there is a symbol shown for the operator on the Picking Display. Available modes IN = Put away, OUT = Pick, INV = Inventory.
TrayCoord	No	Information that can be presented on the panel.
Job	No	Name of the job the order is a part of. Displayed on picking display to guide the operator.
XPos	Yes if “Laser pointer/lightbar” should be used otherwise No.	X-position of the box If AddTrayConfig is used there’s no need for this.
YPos	Yes if Laserpointer otherwise No.	Y-position of the box If AddTrayConfig is used theres no need for this.
XSize	Yes if LightBar otherwise No.	The width of the box If AddTrayConfig is used there’s no need for this.
Para1	Yes if LightBar otherwise No.	If LightBar is used this value should be set to what should be presented in the Y-digit display If AddTrayConfig is used there’s no need for this.

AddToQueue Examples

This section illustrates some examples of how the XML interface method calls can be done.

Example 1 - Fetch single tray (no twin)

This shows a simple example how to fetch one tray in an elevator without any accessories

```
<AddToQueue>
  <ElevatorId>E1</ElevatorId>
  <Tray>1</Tray>
  <Opening>1</Opening>
</AddToQueue>
```

The above example adds an order, fetch tray number 1 to opening 1 for elevator E1.

When next order is sent the following happens.

1. If next order uses same tray number as already being present in the opening at the elevator. The tray is left on the opening. Although existing order is acknowledged.

2. If next order is to be on another tray number the order is acknowledged and the tray changed to the ordered number.

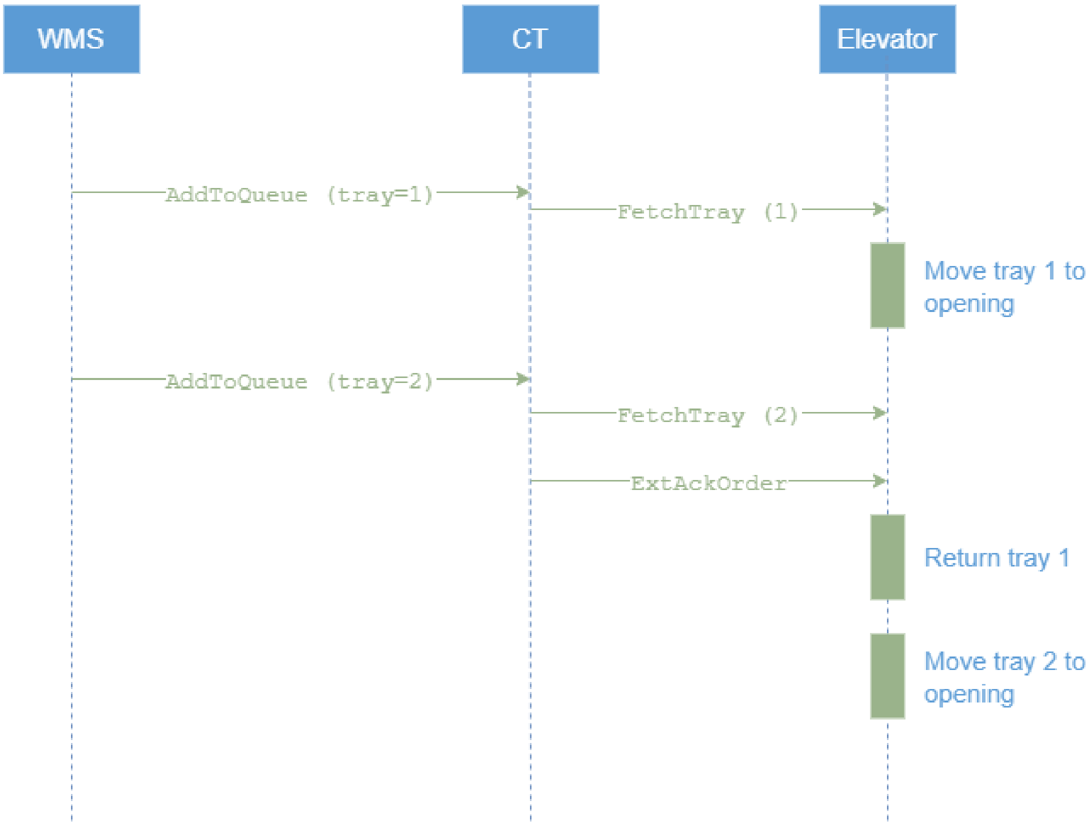


Figure 1. Sequence diagram for elevator with no Twin functionality

Example 2 – AddToQueue with Twin functionality

To be able to use the machines twin functionality, it’s required to send two tray numbers in the AddToQueue method. This is done by using <Tray> and <NextTray> at the same time.

```
<AddToQueue>
  <ElevatorId>E1</ElevatorId>
  <Tray>1</Tray>
  <Opening>1</Opening>
  <NextTray>2</NextTray>
</AddToQueue>
```

The above example shows how to add an order to fetch tray number 1 to opening 1 on elevator E1. It also tells the machine to fetch tray number 2 to be ready for twin mode. The machine then awaits order from WMS.

Scenario 1 – Normal Twin

If next order is the same tray number that was sent in the <NextTray> parameter previously then the machine will change to that tray (normal twin)

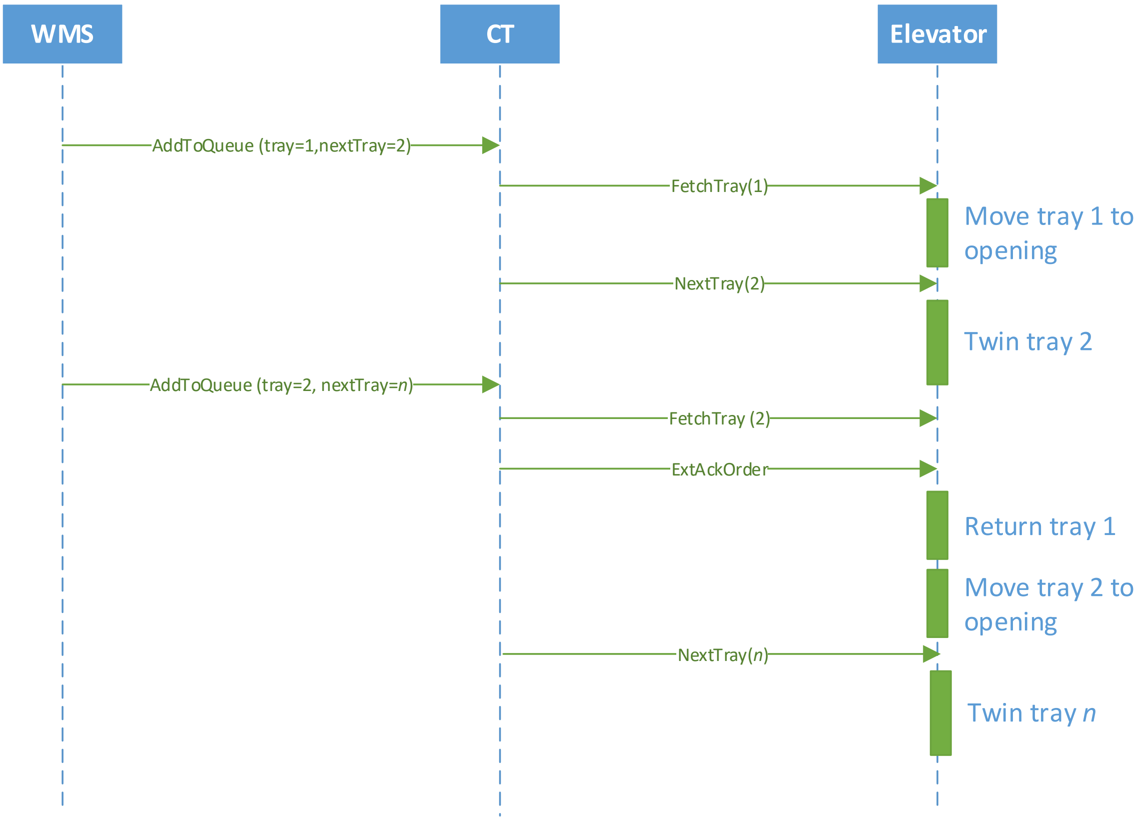


Figure 2. Sequence diagram for normal Twin operation

Scenario 2 – Expected Twin not desired

If instead the next order is another tray number than the previously sent <NextTray>. The machine will return the tray that’s waiting and fetch the new ordered tray number. The machine will then change to the new tray number. See sequence description.

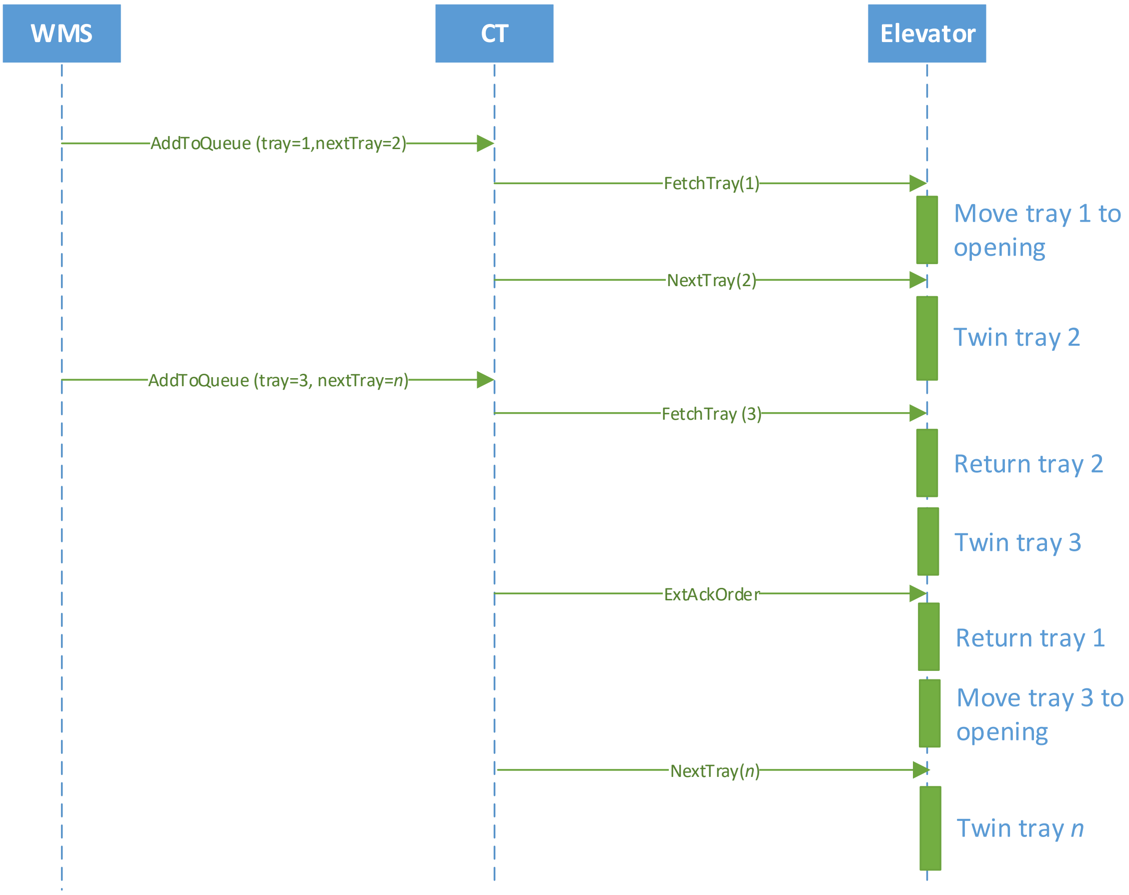


Figure 3. Sequence diagram for aborted Twin operation

10.2.2 ExtAckOrder

ExtAckOrder method is to be used to externally acknowledge/confirm an order. If property NoReturnOfTray is set equal to 1. An external acknowledge is required.

Parameter	Mandatory	Description
ElevatorId	Yes	Identity of the elevator that is the target for the order. Should be the id that is set in the device configuration.
Opening	Yes	Number of the service opening 1 – n
TransId	Yes/No	If response messages are used this element is mandatory. It is used to tie the command and response together.

Example 1

The following example shows an External Acknowledgement on Elevator E1, opening 1.

```
<ExtAckOrder>
  <ElevatorId>E1</ElevatorId>
  <Opening>1</Opening>
```

```
</ExtAckOrder>
```

Example 2

Same functionality can be achieved by using the AddToQueue command by setting tray to 0.

```
<AddToQueue>
  <ElevatorId>E1</ElevatorId>
  <Tray>0</Tray>
  <Opening>1</Opening>
</AddToQueue>
```

10.2.3 AddTrayConfig

To be able to use accessories such as PickDisplay the AddTrayConfig method needs to be used. This to provide the accessory information about the layout of the tray. Data sent to Compact Talk about the tray configuration is cached. Because of the cache the configuration of the layout of the tray only needs to be added to Compact Talk a second time if it has changed.

Parameter	Mandatory	Description
ElevatorId	Yes	Identity of the elevator that is the target.
Tray	Yes	The identity of the tray to be fetched. Must be an existing tray in the machine.
Boxes	Yes	A list of boxes that represents storage areas on the tray.
TransId	Yes/No	If response messages are used this element is mandatory. It is used to tie the command and response together.

Example

This example shows to use AddTrayConfig method for one tray (tray 1) in Elevator E1. Finally the example shows how to use the added tray layout in combination with AddToQueue.

```
<AddTrayConfig>
  <ElevatorId>E1</ElevatorId>
  <Tray>1</Tray>
  <Boxes>
    <Box>
      <Name>A-1</Name>
      <XPos>0</XPos>
      <YPos>0</YPos>
      <XSize>244</XSize>
      <YSize>164</YSize>
    </Box>
    <Box>
      <Name>A-2</Name>
      <XPos>0</XPos>
      <YPos>164</YPos>
      <XSize>244</XSize>
      <YSize>164</YSize>
    </Box>
  </Boxes>
</AddTrayConfig>
<AddToQueue>
  <ElevatorId>E1</ElevatorId>
  <Tray>1</Tray>
  <Opening>1</Opening>
  <NoReturnOfTray>0</NoReturnOfTray>
  <CurrentBoxName>A-1</CurrentBoxName>
</AddToQueue>
```

CurrentBoxName is in this example is set to “A-1” which is currently described in the AddTrayConfig.

10.2.4 ResetElevator

Maintenance function that recovers orders and elevator for the specified opening/openings

Parameter	Mandatory	Description
ElevatorId	Yes	Identity of the elevator that is the target.
Opening	No	Number of the service opening. Use opening 1 – n. 99 means entire elevator/ all openings on the elevator. If value is not set 99 will be used as standard.
TransId	Yes/No	If response messages are used this element is mandatory. It is used to tie the command and response together.

Example 1

The following example shows a ”reset of” opening 1 in elevator E1.

```
<ResetElevator>
  <ElevatorId>E1</ElevatorId>
  <Opening>1</Opening>
</ResetElevator>
```

Example 2

Same functionality can be achieved by using the AddToQueue command with tray number 1000.

```
<AddToQueue>
  <ElevatorId>E1</ElevatorId>
  <Tray>1000</Tray>
  <Opening>1</Opening>
</AddToQueue>
```

10.3 Response Messages

If response messages are needed it’s possible, in the configuration, to select which level of feedback that is wanted.

10.3.1 CommandResponse

Parameter	Description
Command	The name of the command
Result	If 0, the command failed
TransId	If response messages is used this element is mandatory. It is used to tie the command and response together.
ErrorMessage	Contains a description of the error that occurred. Only included when **Result** is 0..

Example 1
An example of a successful command.

```
<CompactTalkResponse xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Response xsi:type="CommandResponse">
    <TransId>1</TransId>
    <Command>AddToQueue</Command>
    <Result>916</Result>
  </Response>
</CompactTalkResponse>
```

Example 1
An example of an unsuccessful command.

```
<CompactTalkResponse xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Response xsi:type="CommandResponse">
    <TransId>3</TransId>
    <Command>AddToQueue</Command>
    <Result>0</Result>
    <ErrorMessage>[E=G2_1,T=333] ValidateOrderData: Tray number 333 does not exist
within elevator G2_1</ErrorMessage>
  </Response>
</CompactTalkResponse>
```

10.3.2 OrderStatusResponse

Parameter	Description
TransId	If response messages are used this element is mandatory. It is used to tie the command and response together.
Status	Either Sent, NextAtPlace or AtPlace depending on configuration.

```
<?xml version="1.0" encoding="utf-8"?>
<CompactTalkResponse xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Response xsi:type="OrderStatusResponse">
    <TransId>2</TransId>
    <Status>Sent</Status>
  </Response>
</CompactTalkResponse>
```

10.3.3 TaskDoneResponse

Parameter	Description
TransId	If response messages are used this element is mandatory. It is used to tie the command and response together.
Mode	Either IN, OUT or INV. Same value that was used in the AddToQueue command.
AckQuantity	The amount of material handled by the operator. Only has a value if **NoReturnOfTray** was set to 0 in the AddToQueue call.

```
<?xml version="1.0" encoding="utf-8"?>
<CompactTalkResponse xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Response xsi:type="TaskDoneResponse">
```

```
    <TransId>2</TransId>
    <Mode>OUT</Mode>
    <AckQuantity>0</AckQuantity>
  </Response>
</CompactTalkResponse>
```

11 Import and Export

The import and export functions of Compact Talk is a plugin based solution which in a basic installation contains plugins for basic flat file import and export, but others can be added by other parties.

11.1 Import

To import files to Compact Talk there are three things that needs to be done, adding an instance of an importer to the configuration, basic configuration and defining the format of the file content. This is done with the configuration tool. See the Configuration Manual [1] for more details and examples.

11.2 Export

To export files from Compact Talk there are three things that needs to be done, adding an instance of an exporter to the configuration, basic configuration and defining the format of the file content. This is done with the configuration tool. See the Configuration Manual [1] for more details and examples.

11.3 Tray configuration import

Tray configuration import is a specialized type of import that uses a flat file to handle tray layout configurations. To aid the operator Compact Talk uses this configuration to control accessory devices highlighting the boxes where an article is placed. There are methods available in the client interface that do the same thing, but there are cases where the import method is to be preferred.

11.3.1 Configuration

Use the configuration tool to select the file to be imported, see the Configuration Manual [1] for details.

11.3.2 Format

The tray configuration file needs to contain one box per row split up in 9 columns, where 2 are optional, separated with a pipe character ‘|’.
The nine columns are:

1. Elevator identity
2. Tray number
3. The name of the box. Used when adding an order to identify in which box the article is placed.
4. Position in X (mm)
5. Position in Y (mm)

6. Size in X (mm)
7. Size in Y (mm)
8. Optional numeric value. Implementation specific.
9. Optional text value. Implementation specific.

Example:
Elevator_1|1|A-1|101|1|100|200
This example shows a box for **Elevator_1** tray **1** with the name **A-1**, with coordinates **101, 1**, width **100** and height **200**.