Issue an instruction, track its progress and retrieve responses

Introduction

Issuing an instruction is the creation of a new instruction by Consumer API, as well as its workflow and response tables by the Coordinator . Issuing an instruction is done by sending an appropriate JSON to the API endpoint responsible for creating new instructions.

For this section you should:

- Know what instruction to issue - read about searching instructions in find the right instruction definition
- Have an issued instruction that does not require authentication or authorization and follows the default workflow.

The C# examples assume you're using Tachyon Consumer SDK and you already have a correctly instantiated instance of Tachyon connector class in an object called connector.

All SDK methods return the same class called ApiCallResponse. Inside the object of ApiCallResponse you'll find a property called ReceivedObject. That object is the actual data received from the API. In the following examples this detail is left out, stating that the returned object contains the data. For example, when we say that XYZ object contains certain data, this means the ReceivedObject contains that data, since that's always true.

In most circumstances, the lifecycle of an Instruction is:

1. A user issuing the instruction, through Explorer or the Consumer API directly.
2. The Consumer API accepting the instruction, after which it is sent to Agents.
3. The Agents which execute the instruction, sending back any data it has produced or indicate a failure.

This data is made available to consumers (like Tachyon Explorer). For more information read about the Tachyon Instruction lifecycle.

Issuing an instruction

Pick an Instruction Definition to base your new Instruction on. To issue that instruction, use a POST to the Instructions controller - sending a JSON object describing what the instruction should be.

The endpoint you should send the data to is https://my.tachyon.server/Consumer/Instructions, where my.tachyon.server is replaced with the address of your Tachyon instance.

On this page:

Simple instruction

This example covers the "How are network adapters configured?" instruction definition. It does not require any parameters and it will be sent to all connected agents.

In this example, the instruction definition has the Id 1120, but we could also use its name, which is "1E-Explorer-TachyonCore-NetworkAdapterConfigurationDetails". You cannot use both Id and Name because it may introduce ambiguity.

Direct Consumer API call	C# code using Consumer SDK library
If you wish to use the Id you can make a POST request with following payload:	

Payload sent to https://my.tachyon.server/Consumer/Instructions

```
{
    "DefinitionId": 1120,
    "InstructionTtlMinutes": 60,
    "ResponseTtlMinutes": 60,
    "KeepRaw": 1
}
```

If you wish to use the Name instead, you can make a POST request with this payload:

Payload sent to https://my.tachyon.server/Consumer/Instructions

```
{
    "DefinitionName": "1E-Explorer-
TachyonCore-
NetworkAdapterConfigurationDetails",
    "InstructionTtlMinutes": 60,
    "ResponseTtlMinutes": 60,
    "KeepRaw": 1
}
```

In both cases the request returns this response:

Return payload

```
{
    "Id": 2004,
    "Sequence": 0,
    "Name": "1E-Explorer-TachyonCore-
NetworkAdapterConfigurationDetails",
    "Description": "Get the
configuration of the network
adapters. Windows Only.",
    "InstructionType": 0,
    "ReadablePayload": "How are
network adapters configured?",
    "Cmd": "SendAll",
    "Schema": [
        {
            "Name": "Description",
            "Type": "string",
            "Length": 128,
            "RenderAs": null
        },
        {
            "Name": "DHCPEnabled",
            "Type": "bool",
            "Length": 0,
            "RenderAs": null
        },
        {
            "Name":
"DHCPLeaseExpires",
            "Type": "datetime",
            "Length": 0,
            "RenderAs": null
        },
        {
            "Name":
"DHCPLeaseObtained",
```

Use the Instructions object inside the Tachyon connector instance.

If you wish to use the instruction definition's Id:

Issuing a new Instruction using instructions definition's Id

```
var newInstruction = connector.Instructions.SendInstruction
(1120, null, 60, 120);
if (newInstruction.Success)
{
   var idToTrack = newInstruction.ReceivedObject.Id;
   // track instruction and retrieve responses
}
else
{
   foreach (var error in newInstruction.Errors)
   {
      // display error
   }
}
```

If you wish to use the instruction definition's name:

Issuing a new Instruction using instructions definition's name

```
var newInstruction = connector.Instructions.SendInstruction("1E-
Explorer-TachyonCore-NetworkAdapterConfigurationDetails", null,
60, 120);
if (newInstruction.Success)
{
   var idToTrack = newInstruction.ReceivedObject.Id;
   // track instruction and retrieve responses
}
else
{
   foreach (var error in newInstruction.Errors)
   {
      // display error
   }
}
```

In either case, the newInstruction object will contain the same data you can see in the Json response on the left.

```
            "Type": "datetime",
            "Length": 0,
            "RenderAs": null
        },
        {
            "Name": "DHCPServer",
            "Type": "string",
            "Length": 32,
            "RenderAs": null
        },
        {
            "Name": "DNSDomain",
            "Type": "string",
            "Length": 32,
            "RenderAs": null
        },
        {
            "Name": "DNSHostName",
            "Type": "string",
            "Length": 32,
            "RenderAs": null
        },
        {
            "Name":
"DNSServerSearchOrder",
            "Type": "string",
            "Length": 65535,
            "RenderAs": null
        },
        {
            "Name": "IPAddress",
            "Type": "string",
            "Length": 65535,
            "RenderAs": null
        },
        {
            "Name": "IPSubnet",
            "Type": "string",
            "Length": 65535,
            "RenderAs": null
        },
        {
            "Name": "MACAddress",
            "Type": "string",
            "Length": 64,
            "RenderAs": null
        }
    ],
    "Aggregation": null,
    "KeepRaw": true,
    "Scope": null,
    "InstructionTtlMinutes": 60,
    "ResponseTtlMinutes": 60,
    "CreatedTimestampUtc": "2018-04-
18T13:29:25.09Z",
    "SentTimestampUtc": null,
    "Status": 0,
    "WorkflowState": 0,
    "StatusTimestampUtc": null,
    "Export": false,
    "ExportLocation": null,
    "ParentInstructionId": null,
    "InstructionDefinitionId": 1120,
    "CreatedBy":
"SomeDomain\\SomeUser",
    "ResultsFilter": null,
    "PreviousResultsFilter": null,
    "ConsumerId": 1,
    "ConsumerName": "Explorer",
    "ConsumerCustomData": null,
    "ParameterJson": null,
```

```
    "OffloadResponses": false,
    "RequestedFor": null,
    "ResponseTemplateId": null,
    "ResponseTemplateConfiguration":
null,
    "Workflow": "{\"StateMachine\":\"
State\"}",
    "Comments": null,
    "ScheduledInstructionId": null,
    "ApprovalOffloaded": false,
    "ActionedBy": null,
    "ActionReason": null
}
```

The data sent to the API includes:

Term	Definition
DefinitionId or **DefinitionName**	You can use either DefinitionId or DefinitionName, but not both as that can introduce ambiguity. DefinitionId should be the Id of the instruction definition your instruction should be based upon. DefinitionName should be the name of the instruction definition your instruction should be based upon.
InstructionTtlMinutes	Amount of time responses are gathered for. This TTL starts when the instruction goes live (is sent to agents), not when it's received by the API. This is because there might be a authentication and/or approval workflow to execute between the API receiving the instruction and it going live. Also, the instruction might never go live if it's rejected or stuck indefinitely in approval.
ResponseTtlMinutes	Amount of time the responses are available inside Tachyon after they've been gathered. When InstructionTtlMinutes elapses, ResponseTtlMinues starts.
KeepRaw	Is only used by aggregated instructions and defines if raw results should be kept next to aggregated ones. If you choose not to keep raw results, only aggregated ones will be available, so you'll not be able to drill down into the details of each aggregate. Though not strictly necessary, this setting was included to make sure raw results are kept, as if they're not sent, this setting will default to "false" and raw results will not be kept.

DefinitionId (or DefinitionName), InstructionTtlMinutes and ResponseTtlMinutes form the minimum information to provide to the API when issuing the instruction.

The recieved response contains information about the instruction consisting of the data you've sent, like the parameters, scope, TTLs as well as information inherited from the Instruction Definition it was based on. The most important piece of information is the Id of the instruction (in this case its 2004), which will be used to track the instruction's progress and retrieve responses.

Adding parameters

Some instructions require data for the instruction to work. This data is supplied in the form of a collection of parameters, each one a key - value pair. The key is the parameter name and can be obtained from the Instruction Definition.

The Instruction Definition describes the parameters and the rules each value must comply to. These parameters are added into the appropriate locations in the instruction server side when it's issued.

This example uses the Echo instruction definition ("1E-Explorer-TachyonAgent-Echo"), which in my Tachyon installation has an id of 60, and the following definition:

Direct Consumer API call	C# code using Consumer SDK library

You can retrieve the instruction definition using its Id by making a GET request to **https://my.tachyon.server/Consumer/InstructionDefinitions/id/60**, or using its name by making a request to **https://my.tachyon.server/Consumer/InstructionDefinitions/Name/1E-Explorer-TachyonAgent-Echo**.

Whichever method you choose the return payload will look something like this:

Response received from https://my.tachyon.server/Consumer/InstructionDefinitions/id/60


```
{
    "Id": 60,
    "Name": "1E-Explorer-TachyonAgent-Echo",
    "Description": "Send an echo request to devices",
    "InstructionSetId": 2,
    "InstructionSetName": "Everything",
    "InstructionType": 0,
    "ReadablePayload": "Echo message %msg%.",
    "Parameters": [
        {
            "Name": "msg",
            "Pattern": "%msg%",
            "DataType": "string",
            "ControlType": "freeText",
            "ControlMetadata": null,
            "Placeholder": "message to echo",
            "DefaultValue": null,
            "Validation": {
                "Regex": null,
                "MaxLength": "200",
                "AllowedValues": null,
                "NumValueRestrictions": null
            },
            "Value": null,
            "HintText": null,
            "Source": null
        }
    ],
    "Schema": [
        {
            "Name": "Message",
            "Type": "string",
            "Length": 256,
            "RenderAs": null
        }
    ],
    "Aggregation": null,
    "InstructionTtlMinutes": 60,
    "ResponseTtlMinutes": 120,
    "MinimumInstructionTtlMinutes": 10,
    "MaximumInstructionTtlMinutes": 1440,
    "MinimumResponseTtlMinutes": 10,
    "MaximumResponseTtlMinutes": 10080,
    "Workflow": null,
    "ResponseTemplateId": null,
    "OriginalFileName": null,
    "ResponseTemplateConfiguration": null,
    "Version": "7",
    "Author": "1E",
    "IsLicensed": true,
    "UploadedTimestampUtc": "2018-03-08T15:22:11.2Z",
    "NumberOfTimesExecuted": 6
}
```


Use the InstructionDefinitions object inside the Tachyon connector instance.

You can retrieve the definition by its Id:

Retrieving instruction definition details by Id

```
var definition = connector.
InstructionDefinitions.
GetInstructionDefinition(60);
```

or by its name:

Retrieving instruction definition details by Name

```
var definition = connector.
InstructionDefinitions.
GetInstructionDefinition("1E-
Explorer-TachyonAgent-Echo");
```

In eithercase, definition object will contain the same data you can see in the JSON response on the left

As we can see the Parameters array has a single parameter in this instruction definition. This object defines a parameter that has to be provided when this instruction is issued.

Direct Consumer API call

You can issue an instruction using the instruction definition's Id by making a POST request with following payload:

Payload sent to https://my.tachyon.server/Consumer/Instructions

```
{
    "DefinitionId": 60,
    "Parameters": [{
        "Name": "msg",
        "Value": "hello!"
    }],
    "InstructionTtlMinutes": 60,
    "ResponseTtlMinutes": 120,
    "KeepRaw": 1
}
```

or using the instruction definition's name by making a POST request with following payload:

Payload sent to https://my.tachyon.server/Consumer/Instructions

```
{
    "DefinitionName": "1E-Explorer-TachyonAgent-Echo",
    "Parameters": [{
        "Name": "msg",
        "Value": "hello!"
    }],
    "InstructionTtlMinutes": 60,
    "ResponseTtlMinutes": 120,
    "KeepRaw": 1
}
```

Both calls will return the same response:

C# code using Consumer SDK library

Use Instructions object inside the Tachyon connector instance.

Parameters are passed as a dictionary where key is the name of the parameter and value is the value you wish the parameter to have.

You can use instruction definition's Id:

Issuing a new Instruction with a parameter

```
var parameters = new
Dictionary<string,string>{
{"msg", "hello!"} };
var newInstruction = connector.
Instructions.SendInstruction
(60, parameters, 60, 120);
if (newInstruction.Success)
{
   var idToTrack =
newInstruction.ReceivedObject.
Id;
   // track instruction and
retrieve responses
}
else
{
   foreach (var error in
newInstruction.Errors)
   {
      // display error
   }
}
```

or its name:

Issuing a new Instruction with a parameter

```
var parameters = new
Dictionary<string,string>{
{"msg", "hello!"} };
var newInstruction = connector.
Instructions.SendInstruction
("1E-Explorer-TachyonAgent-
Echo", parameters, 60, 120);
if (newInstruction.Success)
{
   var idToTrack =
newInstruction.ReceivedObject.
Id;
   // track instruction and
retrieve responses
}
else
{
   foreach (var error in
newInstruction.Errors)
   {
      // display error
   }
}
```

Return payload


```
{
    "Id": 2006,
    "Sequence": 0,
    "Name": "1E-Explorer-TachyonAgent-Echo",
    "Description": "Send an echo request to devices",
    "InstructionType": 0,
    "ReadablePayload": "Echo message hello!.",
    "Cmd": "SendAll",
    "Schema": [
        {
            "Name": "Message",
            "Type": "string",
            "Length": 256,
            "RenderAs": null
        }
    ],
    "Aggregation": null,
    "KeepRaw": true,
    "Scope": null,
    "InstructionTtlMinutes": 60,
    "ResponseTtlMinutes": 120,
    "CreatedTimestampUtc": "2018-04-19T07:10:01.567Z",
    "SentTimestampUtc": null,
    "Status": 0,
    "WorkflowState": 0,
    "StatusTimestampUtc": null,
    "Export": false,
    "ExportLocation": null,
    "ParentInstructionId": null,
    "InstructionDefinitionId": 60,
    "CreatedBy": "SomeDomain\\SomeUser",
    "ResultsFilter": null,
    "PreviousResultsFilter": null,
    "ConsumerId": 1,
    "ConsumerName": "Explorer",
    "ConsumerCustomData": null,
    "ParameterJson": "[{\"Name\":\"msg\",\"Pattern\":\"%msg%\",\"
DataType\":null,\"ControlType\":null,\"ControlMetadata\":null,\"
Placeholder\":null,\"DefaultValue\":null,\"Validation\":null,\"Value\":
\"hello!\",\"HintText\":null,\"Source\":null}]",
    "OffloadResponses": false,
    "RequestedFor": null,
    "ResponseTemplateId": null,
    "ResponseTemplateConfiguration": null,
    "Workflow": "{\"StateMachine\":\"State\"}",
    "Comments": null,
    "ScheduledInstructionId": null,
    "ApprovalOffloaded": false,
    "ActionedBy": null,
    "ActionReason": null
}
```


In either case, newInstruction object will contain the same data you can see in the Json response on the left.

In the response payload above, the parameters are now completed.

The Readable payload of the Instruction Definition changed from "Echo message %msg%." to "Echo message hello!." because the value of the "msg" parameter was "hello!"

If an instruction has more parameters there would be other entries in the array of parameters.

Narrowing the scope

If you do not want the instruction sent to all agents but to a subset, you can use the scope and describe the properties of the computers you want to target. To use the scope fill in the Scope field in the payload sent to the API.

You should read Defining the scope for detailed information about scope and expressions, including the fields you can use in the scope.

This example defines a scope sending the instruction to Windows 7 computers :

Direct Consumer API call

Making a POST request with following payload:

Payload sent to https://my.tachyon.server/Consumer/Instructions

```
{
    "DefinitionId": 1120,
    "InstructionTtlMinutes": 60,
    "ResponseTtlMinutes": 60,
    "Scope": {
        "Operator": "AND",
        "Operands": [{
            "Attribute": "OsType",
            "Operator": "==",
            "Value": "Windows"
        },
        {
            "Attribute": "OsVerTxt",
            "Operator": "Like",
            "Value": "%7%"
        }]
    }, "KeepRaw": 1
}
```

Returns the following response:

Return payload

```
{
    "Id": 2008,
    "Sequence": 0,
    "Name": "1E-Explorer-TachyonCore-
NetworkAdapterConfigurationDetails",
    "Description": "Get the configuration of the
network adapters. Windows Only.",
    "InstructionType": 0,
    "ReadablePayload": "How are network adapters
configured?",
    "Cmd": "SendAll",
    "Schema": [
        {
            "Name": "Description",
            "Type": "string",
            "Length": 128,
            "RenderAs": null
        },
        {
            "Name": "DHCPEnabled",
            "Type": "bool",
            "Length": 0,
            "RenderAs": null
        },
        {
            "Name": "DHCPLeaseExpires",
            "Type": "datetime",
            "Length": 0,
            "RenderAs": null
        },
        {
            "Name": "DHCPLeaseObtained",
            "Type": "datetime",
            "Length": 0,
            "RenderAs": null
        },
        {
```

C# code using Consumer SDK library

Use the Instructions object inside the Tachyon connector instance.

Here you'll have to create a send model representing an instruction.

Issuing a new Instruction with a scope

```
var instruction = new Tachyon.SDK.Consumer.
Models.Send.Instruction
{
   DefinitionId = 1120,
   InstructionTtlMinutes = 60,
   ResponseTtlMinutes = 120,
   KeepRaw = true,
   Scope = new ExpressionObject
   {
      Operator = "AND",
      Operands = new List<ExpressionObject>
      {
         new ExpressionObject
         {
            Attribute = "OsType",
            Operator = "==",
            Value = "Windows"
         },
         new ExpressionObject
         {
            Attribute = "OsVerTxt",
            Operator = "Like",
            Value = "%7%"
         }
      }
   }
};

var newInstruction = connector.Instructions.
SendInstruction(instruction);
if (newInstruction.Success)
{
   var idToTrack = newInstruction.
ReceivedObject.Id;
   // track instruction and retrieve responses
}
else
{
   foreach (var error in newInstruction.Errors)
   {
      // display error
   }
}
```

newInstruction object will contain the same data you can see in the JSON response on the left.

```
            "Name": "DHCPServer",
            "Type": "string",
            "Length": 32,
            "RenderAs": null
        },
        {
            "Name": "DNSDomain",
            "Type": "string",
            "Length": 32,
            "RenderAs": null
        },
        {
            "Name": "DNSHostName",
            "Type": "string",
            "Length": 32,
            "RenderAs": null
        },
        {
            "Name": "DNSServerSearchOrder",
            "Type": "string",
            "Length": 65535,
            "RenderAs": null
        },
        {
            "Name": "IPAddress",
            "Type": "string",
            "Length": 65535,
            "RenderAs": null
        },
        {
            "Name": "IPSubnet",
            "Type": "string",
            "Length": 65535,
            "RenderAs": null
        },
        {
            "Name": "MACAddress",
            "Type": "string",
            "Length": 64,
            "RenderAs": null
        }
    ],
    "Aggregation": null,
    "KeepRaw": true,
    "Scope": {
        "Operator": "AND",
        "Operands": [
            {
                "Attribute": "OsType",
                "Operator": "==",
                "Value": "Windows",
                "DataType": "String"
            },
            {
                "Attribute": "OsVerTxt",
                "Operator": "Like",
                "Value": "%7%",
                "DataType": "String"
            }
        ]
    },
    "InstructionTtlMinutes": 60,
    "ResponseTtlMinutes": 60,
    "CreatedTimestampUtc": "2018-04-19T10:02:27.523Z",
    "SentTimestampUtc": null,
    "Status": 0,
    "WorkflowState": 0,
    "StatusTimestampUtc": null,
    "Export": false,
    "ExportLocation": null,
    "ParentInstructionId": null,
```

```
    "InstructionDefinitionId": 1120,
    "CreatedBy": "SomeDomain\\SomeUser",
    "ResultsFilter": null,
    "PreviousResultsFilter": null,
    "ConsumerId": 1,
    "ConsumerName": "Explorer",
    "ConsumerCustomData": null,
    "ParameterJson": null,
    "OffloadResponses": false,
    "RequestedFor": null,
    "ResponseTemplateId": null,
    "ResponseTemplateConfiguration": null,
    "Workflow": "{\"StateMachine\":\"State\"}",
    "Comments": null,
    "ScheduledInstructionId": null,
    "ApprovalOffloaded": false,
    "ActionedBy": null,
    "ActionReason": null
}
```

Targeting specific devices

You can also send instructions to a specific list of computers. To do this, you'll need to call a different API endpoint.

The https://my.tachyon.server/Consumer/Instructions/Targeted endpoint takes a payload almost identical to one used previously, when we were using https://my.tachyon.server/Consumer/Instructions, but contains an additional field called 'Devices'. The Devices should be a list of FQDNs of the computers to send the instruction to.

All supplied FQDNs have to be valid and known to Tachyon at the time the instruction's issued.

Depending on the permissions of the user making the API call, the list of FQDNs might be filtered to exclude devices that user doesn't have permissions to, based on Management groups associated with their permissions. If it turns out that the caller doesn't have permission to any of the devices whose FQDNs they provided (so when the list is filtered down to zero elements), an error will be retrurned.

Direct Consumer API call	C# code using Consumer SDK library
Like with a regular instruction, you can issue a targeted instruction using instruction definition's Id by making a POST request with following payload:	

Payload sent to https://my.tachyon.server/Consumer/Instructions/Targeted

```
{
    "DefinitionId": 1120,
    "InstructionTtlMinutes": 60,
    "ResponseTtlMinutes": 60,
    "Devices": ["Device.SomeDomain.
local"],
    "KeepRaw": 1
}
```

or by using instruction definition's name by making a POST request with following payload:

Payload sent to https://my.tachyon.server/Consumer/Instructions/Targeted

```
{
    "DefinitionName": "1E-Explorer-
TachyonCore-
NetworkAdapterConfigurationDetails",
    "InstructionTtlMinutes": 60,
    "ResponseTtlMinutes": 60,
    "Devices": ["Device.SomeDomain.
local"],
    "KeepRaw": 1
}
```

Returns this response:

Return payload

```
{
    "Id": 2010,
    "Sequence": 0,
    "Name": "1E-Explorer-
TachyonCore-
NetworkAdapterConfigurationDetails",
    "Description": "Get the
configuration of the network
adapters. Windows Only.",
    "InstructionType": 0,
    "ReadablePayload": "How are
network adapters configured?",
    "Cmd": "SendList",
    "Schema": [
        {
            "Name": "Description",
            "Type": "string",
            "Length": 128,
            "RenderAs": null
        },
        {
            "Name": "DHCPEnabled",
            "Type": "bool",
            "Length": 0,
            "RenderAs": null
        },
        {
            "Name":
"DHCPLeaseExpires",
            "Type": "datetime",
            "Length": 0,
            "RenderAs": null
        },
        {
            "Name":
"DHCPLeaseObtained",
            "Type": "datetime",
            "Length": 0,
            "RenderAs": null
        },
        {
            "Name": "DHCPServer",
            "Type": "string",
            "Length": 32,
            "RenderAs": null
        },
        {
            "Name": "DNSDomain",
            "Type": "string",
```

Use Instructions object inside the Tachyon connector instance.

To issue a targeted instruction using instruction definition's Id:

Issuing a new targeted Instruction using instruction definition's Id

```
var newInstruction = connector.Instructions.
SendTargetedInstruction(1120, null, 60, 60, new List<string> {
"Device.SomeDomain.local" });
if (newInstruction.Success)
{
   var idToTrack = newInstruction.ReceivedObject.Id;
   // track instruction and retrieve responses
}
else
{
   foreach (var error in newInstruction.Errors)
   {
      // display error
   }
}
```

To issue it using instruction definition's name:

Issuing a new targeted Instruction using instruction definition's name

```
var newInstruction = connector.Instructions.
SendTargetedInstruction("1E-Explorer-TachyonCore-
NetworkAdapterConfigurationDetails", null, 60, 60, new
List<string> { "Device.SomeDomain.local" });
if (newInstruction.Success)
{
   var idToTrack = newInstruction.ReceivedObject.Id;
   // track instruction and retrieve responses
}
else
{
   foreach (var error in newInstruction.Errors)
   {
      // display error
   }
}
```

In either case, the newInstruction object will contain the same data you can see in the JSON response on the left.

```
            "Length": 32,
            "RenderAs": null
        },
        {
            "Name": "DNSHostName",
            "Type": "string",
            "Length": 32,
            "RenderAs": null
        },
        {
            "Name":
"DNSServerSearchOrder",
            "Type": "string",
            "Length": 65535,
            "RenderAs": null
        },
        {
            "Name": "IPAddress",
            "Type": "string",
            "Length": 65535,
            "RenderAs": null
        },
        {
            "Name": "IPSubnet",
            "Type": "string",
            "Length": 65535,
            "RenderAs": null
        },
        {
            "Name": "MACAddress",
            "Type": "string",
            "Length": 64,
            "RenderAs": null
        }
    ],
    "Aggregation": null,
    "KeepRaw": true,
    "Scope": null,
    "InstructionTtlMinutes": 60,
    "ResponseTtlMinutes": 60,
    "CreatedTimestampUtc": "2018-04-
19T11:07:19.4Z",
    "SentTimestampUtc": null,
    "Status": 0,
    "WorkflowState": 0,
    "StatusTimestampUtc": null,
    "Export": false,
    "ExportLocation": null,
    "ParentInstructionId": null,
    "InstructionDefinitionId": 1120,
    "CreatedBy":
"SomeDomain\\SomeUser",
    "ResultsFilter": null,
    "PreviousResultsFilter": null,
    "ConsumerId": 1,
    "ConsumerName": "Explorer",
    "ConsumerCustomData": null,
    "ParameterJson": null,
    "OffloadResponses": false,
    "RequestedFor": null,
    "ResponseTemplateId": null,

"ResponseTemplateConfiguration":
null,
    "Workflow": "{\"StateMachine\":
\"State\"}",
    "Comments": null,
    "ScheduledInstructionId": null,
    "ApprovalOffloaded": false,
    "ActionedBy": null,
    "ActionReason": null
```

```
}
```

The response does not contain the list of devices because it's kept in a form consumable by the Switch and not as FQDNs.

Converting this internal format back to FQDN list is expensive. If you want that list, you'll have to explicitly ask for it using a different endpoint: https://my.tachyon.server/Consumer/Instructions/{instructionId}/targetlist

Where instructionId is the Id of the instruction whose targets you want. The return payload will be a list of FQDNs.

Direct Consumer API call	C# code using Consumer SDK library
Making a GET request to https://my.tachyon.server/Consumer/Instructions /2010/targetlist returns this response: **Response received from https://my.tachyon.server/Consumer/Instructions/2010/targetlist** `[ "1EUKDEVWKS1204.1e.local" ]`	Use Instructions object inside the Tachyon connector instance. **Getting the list of target devices for given instruction** `var targets = connector.Instructions.GetTargetsByInstructionId(2010);`

Following up on an instruction

Follow up instructions are a way of chaining separate instructions so they're executing together and appear as if they're one instruction. Each instruction in the chain forms a part of the instruction by adding more code to execute.

To issue a follow up instruction you can use the APIs used in the examples and add "ParentInstructionId" fiield to the payload. The value of "ParentInstructionId" should be an Instruction Id.

Direct Consumer API call	C# code using Consumer SDK library
You can issue a follow up instruction using instruction definition's Id by making a POST request with following payload: **Payload sent to https://my.tachyon.server/Consumer/Instructions** `{ "DefinitionId": 1120, "InstructionTtlMinutes": 60, "ResponseTtlMinutes": 60, "KeepRaw": 1, "ParentInstructionId": 2012 }` You can also use the instruction definition's name by making a POST request with following payload	

Payload sent to https://my.tachyon.server/Consumer/Instructions

```
{
    "DefinitionName": "1E-Explorer-
TachyonCore-
NetworkAdapterConfigurationDetails",
    "InstructionTtlMinutes": 60,
    "ResponseTtlMinutes": 60,
    "KeepRaw": 1,
    "ParentInstructionId": 2012
}
```

Returns this response:

Return payload

```
{
    "Id": 2013,
    "Sequence": 0,
    "Name": "1E-Explorer-
TachyonCore-
NetworkAdapterConfigurationDetails",
    "Description": "Get the
configuration of the network
adapters. Windows Only.",
    "InstructionType": 0,
    "ReadablePayload": "How are
network adapters configured?",
    "Cmd": "SendList",
    "Schema": [
        {
            "Name": "Description",
            "Type": "string",
            "Length": 128,
            "RenderAs": null
        },
        {
            "Name": "DHCPEnabled",
            "Type": "bool",
            "Length": 0,
            "RenderAs": null
        },
        {
            "Name":
"DHCPLeaseExpires",
            "Type": "datetime",
            "Length": 0,
            "RenderAs": null
        },
        {
            "Name":
"DHCPLeaseObtained",
            "Type": "datetime",
            "Length": 0,
            "RenderAs": null
        },
        {
            "Name": "DHCPServer",
            "Type": "string",
            "Length": 32,
            "RenderAs": null
        },
        {
            "Name": "DNSDomain",
            "Type": "string",
            "Length": 32,
```

Use the Instructions object inside the Tachyon connector instance.

Here you'll have to create a send model representing an instruction. You can either use the DefinitionId or DefinitionName field in Tachyon.SDK.Consumer.Models.Send.Instruction, but you cannot use both.

Here's how you'd do it using instruction definition Id:

Issuing a follow-up instruction using instruction definition's Id

```
var instruction = new Tachyon.SDK.Consumer.Models.Send.Instruction
{
   DefinitionId = 1120,
   InstructionTtlMinutes = 60,
   ResponseTtlMinutes = 120,
   KeepRaw = true,
   ParentInstructionId = 2012
};
var newInstruction = connector.Instructions.
SendTargetedInstruction(instruction);
if (newInstruction.Success)
{
   var idToTrack = newInstruction.ReceivedObject.Id;
   // track instruction and retrieve responses
}
else
{
   foreach (var error in newInstruction.Errors)
   {
      // display error
   }
}
```

and here's how you'd do it using the instruction definition's name:

Issuing a follow-up instruction using instruction definition's name

```
var instruction = new Tachyon.SDK.Consumer.Models.Send.Instruction
{
   DefinitionName = "1E-Explorer-TachyonCore-
NetworkAdapterConfigurationDetails",
   InstructionTtlMinutes = 60,
   ResponseTtlMinutes = 120,
   KeepRaw = true,
   ParentInstructionId = 2012
};
var newInstruction = connector.Instructions.
SendTargetedInstruction(instruction);
if (newInstruction.Success)
{
   var idToTrack = newInstruction.ReceivedObject.Id;
   // track instruction and retrieve responses
}
else
{
   foreach (var error in newInstruction.Errors)
   {
      // display error
   }
}
```

In either case, the newInstruction object will contain the same data you can see in the JSON response on the left.

```
            "RenderAs": null
        },
        {
            "Name": "DNSHostName",
            "Type": "string",
            "Length": 32,
            "RenderAs": null
        },
        {
            "Name":
"DNSServerSearchOrder",
            "Type": "string",
            "Length": 65535,
            "RenderAs": null
        },
        {
            "Name": "IPAddress",
            "Type": "string",
            "Length": 65535,
            "RenderAs": null
        },
        {
            "Name": "IPSubnet",
            "Type": "string",
            "Length": 65535,
            "RenderAs": null
        },
        {
            "Name": "MACAddress",
            "Type": "string",
            "Length": 64,
            "RenderAs": null
        }
    ],
    "Aggregation": null,
    "KeepRaw": true,
    "Scope": null,
    "InstructionTtlMinutes": 60,
    "ResponseTtlMinutes": 60,
    "CreatedTimestampUtc": "2018-04-
19T14:16:40.127Z",
    "SentTimestampUtc": null,
    "Status": 0,
    "WorkflowState": 0,
    "StatusTimestampUtc": null,
    "Export": false,
    "ExportLocation": null,
    "ParentInstructionId": 2012,
    "InstructionDefinitionId": 1120,
    "CreatedBy":
"SomeDomain\\SomeUser",
    "ResultsFilter": null,
    "PreviousResultsFilter": null,
    "ConsumerId": 1,
    "ConsumerName": "Explorer",
    "ConsumerCustomData": null,
    "ParameterJson": null,
    "OffloadResponses": false,
    "RequestedFor": null,
    "ResponseTemplateId": null,

"ResponseTemplateConfiguration":
null,
    "Workflow": "{\"StateMachine\":
\"State\"}",
    "Comments": null,
    "ScheduledInstructionId": null,
    "ApprovalOffloaded": false,
    "ActionedBy": null,
    "ActionReason": null
}
```

Only questions can be followed up on. Actions can be issued as a follow-up but cannot be a parent instruction (you cannot follow-up on them). This is because they could modify the endpoint, and if an action was already executed, issuing a follow-up based on it would perform the action again before performing the follow-up instruction. This is because of instruction chaining.

Adding results filters

You can use results filters if you're interested in responses with a specific value.

For example, you might only be interested in the configuration of network adapters for computers belonging to a specific subnet. Because subnet is part of the Instruction's schema, you can add a results filter with the subnet to an Instruction. Only agents with responses matching the filter will send a reply.

If you're asking a follow up instruction, you can add a previous results filter, based on columns returned by the parent instruction. Read results filter for an instruction for more information.

This section focuses on using filters from a coding perspective.

Results filter and previous results filter are both expressions and should be placed in the "ResultsFilter" and 'PreviousResultsFilter" fields in the payload you send to the API when issuing an instruction. Read more in using scope and filter expressions.

In this example, the "1E-Explorer-TachyonCore-AllInstalledSoftware" instruction is issued as a follow up to the instruction issued in the previous example.

Direct Consumer API call	C# code using Consumer SDK library

Making a POST request with following payload:

Payload sent to https://my.tachyon.server/Consumer/Instructions

```
{
    "ParentInstructionId": 2013,
    "DefinitionId": 1051,
    "InstructionTtlMinutes": 10,
    "ResponseTtlMinutes": 30,
    "KeepRaw": 1,
    "ResultsFilter": {
        "Attribute": "Publisher",
        "Operator": "Like",
        "Value": "%Microsoft%"
    },
    "PreviousResultsFilter": {
        "Attribute": "DHCPServer",
        "Operator": "==",
        "Value": "192.168.2.5"
    }
}
```

Returns this response:

Return payload

```
{
    "Id": 2015,
    "Sequence": 0,
    "Name": "1E-Explorer-TachyonCore-
AllInstalledSoftware",
    "Description": "Returns all installed software.",
    "InstructionType": 0,
    "ReadablePayload": "What software is installed?",
    "Cmd": "SendList",
    "Schema": [{
        "Name": "Product",
        "Type": "string",
        "Length": 512,
```

```
    "RenderAs": null
},
{
    "Name": "Publisher",
    "Type": "string",
    "Length": 512,
    "RenderAs": null
},
{
    "Name": "Version",
    "Type": "string",
    "Length": 128,
    "RenderAs": null
},
{
    "Name": "InstallDate",
    "Type": "datetime",
    "Length": 0,
    "RenderAs": null
},
{
    "Name": "Architecture",
    "Type": "string",
    "Length": 20,
    "RenderAs": null
}],
"Aggregation": {
    "Schema": [{
        "Name": "Publisher",
        "Type": "string",
        "Length": 512,
        "RenderAs": null
    },
    {
        "Name": "Product",
        "Type": "string",
        "Length": 512,
        "RenderAs": null
    },
    {
        "Name": "Version",
        "Type": "string",
        "Length": 128,
        "RenderAs": null
    },
    {
        "Name": "Count",
        "Type": "int64",
        "Length": 0,
        "RenderAs": null
    }],
    "GroupBy": "Publisher,Product,Version",
    "Operations": [{
        "Name": "Count",
        "Type": "count"
    }]
},
"KeepRaw": true,
"Scope": null,
"InstructionTtlMinutes": 10,
"ResponseTtlMinutes": 30,
"CreatedTimestampUtc": "2018-04-20T08:03:43.893Z",
"SentTimestampUtc": null,
"Status": 0,
"WorkflowState": 0,
"StatusTimestampUtc": null,
"Export": false,
"ExportLocation": "",
"ParentInstructionId": 2013,
"InstructionDefinitionId": 1051,
"CreatedBy": "SomeDomain\\SomeUser",
```


Use the Instructions object inside the Tachyon connector instance.

Issuing a follow-up instruction with result filter and previous results filter

```
var instruction = new Tachyon.SDK.Consumer.
Models.Send.Instruction
{
   DefinitionId = 1051,
   InstructionTtlMinutes = 10,
   ResponseTtlMinutes = 30,
   KeepRaw = true,
   ParentInstructionId = 2013,
   ResultsFilter = new ExpressionObject
   {
      Operator = "Like",
      Attribute = "Publisher",
      Value = "Microsoft"
   },
   PreviousResultsFilter = new
ExpressionObject
   {
      Operator = "==",
      Attribute = "DHCPServer",
      Value = "192.168.2.5"
   }
};
var newInstruction = connector.Instructions.
SendTargetedInstruction(instruction);
if (newInstruction.Success)
{
   var idToTrack = newInstruction.
ReceivedObject.Id;
   // track instruction and retrieve
responses
}
else
{
   foreach (var error in newInstruction.
Errors)
   {
      // display error
   }
}
```

The newInstruction object will contain the same data you can see in the JSON response on the left.

```
    "ResultsFilter": {
        "Attribute": "Publisher",
        "Operator": "Like",
        "Value": "%Microsoft%",
        "DataType": "string"
    },
    "PreviousResultsFilter": {
        "Attribute": "DHCPServer",
        "Operator": "==",
        "Value": "192.168.2.5",
        "DataType": "string"
    },
    "ConsumerId": 1,
    "ConsumerName": "Explorer",
    "ConsumerCustomData": null,
    "ParameterJson": null,
    "OffloadResponses": false,
    "RequestedFor": null,
    "ResponseTemplateId": 1,
    "ResponseTemplateConfiguration": {
        "Name": "default",
        "TemplateConfigurations": [{
            "Id": "mainchart",
            "Title": "Installed software - 5 most common
application from 5 most common publishers",
            "Type": "Bar",
            "X": "Product",
            "Y": "Count",
            "Z": "Publisher",
            "PostProcessor": "processingFunction",
            "Size": 1,
            "Row": 1
        }],
        "PostProcessors": [{
            "Name": "processingFunction",
            "Function": "ProcessMultiSeries('Product',
'Count', 'Publisher', '5', '5', 'false')"
        }]
    },
    "Workflow": "{\"StateMachine\":\"State\"}",
    "Comments": null,
    "ScheduledInstructionId": null,
    "ApprovalOffloaded": false,
    "ActionedBy": null,
    "ActionReason": null
}
```


Tracking progress

Once the instruction is issued it becomes a part of a workflow. This workflow can include authentication and authorization steps and includes gathering responses, keeping them for a specific amount of time, optionally exporting them and performing a clean-up after the data gathered expires or is deemed no longer needed (for example, instruction was cancelled, see below).

Instructions can have one of the following statuses:

Status	Numerical representation	Description
Created	0	Instruction has just been created. This is a transitional state and the instruction will imminently move to 'InApproval', 'Authenticating' or 'Sent' state, depending on the workflow used by the instruction.
InApproval	1	Instruction is awaiting approval. It will remain in this state until it is approved, rejected or cancelled.
Rejected	2	Instruction has been rejected at approval stage.
Approved	3	Instruction has been approved. This is a transitional state and instruction will imminently move to 'Sent' state.
Sent	4	Instruction has been sent. This is a transitional state and instruction will imminently move to 'InProgress' state.

InProgress	5	Instruction is live and responses are being gathered. It will remain in this state until Instruction TTL runs out or instruction is cancelled.
Complete	6	Instruction has finished. Responses are no longer being gathered, but are available for viewing and instruction will not be sent to any more agents. Instruction will remain in this state until Responses TTL runs out.
Expired	7	Instruction has expired. Responses are no longer available.
Cancelling	8	Instruction has been cancelled but any results gathered were kept. Responses are no longer being gathered, are available for viewing and instruction will not be sent to any more agents. Instruction will remain in this state until Responses TTL runs out.
Cancelled	9	Instruction has been cancelled and any results gathered were deleted. Responses are no longer being gathered, are not available and instruction will not be sent to any more agents.
Failed	10	Instruction has failed.
Authenticating	12	Instruction is awaiting authentication using two-factor authentication mechanism. It remains in this state until it's successfully authenticated, user(s) fail authentication 3 times or it is cancelled.

When the Status of the instruction becomes "InProgress" the instruction is 'live' and is being sent to agents for processing. While the instruction remains in this state, responses are gathered and new agents coming on-line receive the instruction for processing, providing they meet scope or target list criteria, if those were defined for the instruction.

Once the instruction TTL elapses, the instruction is complete and changes its status to "Complete". At this moment responses TTL timer starts ticking and once that elapses the status will change to "Expired" and all responses gathered will be deleted from the system. If when issuing an instruction an export option was set to true, responses are exported automatically before they're deleted.

You can track the Instruction's progress using its Id and two separate APIs, Instructions and Instruction Statistics:

Instructions API - allows you to retrieve details about the instruction, including its Status, to find out where in its life cycle the instruction is.

Instruction Statistics API - allows you to track progress of responses coming in from the agents, best used while instruction is in "InProgress".

Checking Instruction status

Calling https://my.tachyon.server/Consumer/Instructions/{id} returns current information about given instruction, including its status.

Direct Consumer API call

Making a GET request to https://my.tachyon.server/Consumer/Instructions/2015 returns this response:

Return payload


```
{
    "Id": 2015,
    "Sequence": 24,
    "Name": "1E-Explorer-TachyonCore-AllInstalledSoftware",
    "Description": "Returns all installed software.",
    "InstructionType": 0,
    "ReadablePayload": "What software is installed?",
    "Cmd": "SendList",
    "Schema": [
        {
            "Name": "Product",
            "Type": "string",
            "Length": 512,
            "RenderAs": null
        },
        {
            "Name": "Publisher",
            "Type": "string",
            "Length": 512,
            "RenderAs": null
        },
        {
            "Name": "Version",
            "Type": "string",
            "Length": 128,
            "RenderAs": null
        },
        {
```


C# code using Consumer SDK library

Use Instructions object inside the Tachyon connector instance.

```
var instruction = connector.
Instructions.GetInstruction(2015);
```

object will contain the same data you can see in the JSON response on the left.

```
            "Name": "InstallDate",
            "Type": "datetime",
            "Length": 0,
            "RenderAs": null
        },
        {
            "Name": "Architecture",
            "Type": "string",
            "Length": 20,
            "RenderAs": null
        }
    ],
    "Aggregation": {
        "Schema": [
            {
                "Name": "Publisher",
                "Type": "string",
                "Length": 512,
                "RenderAs": null
            },
            {
                "Name": "Product",
                "Type": "string",
                "Length": 512,
                "RenderAs": null
            },
            {
                "Name": "Version",
                "Type": "string",
                "Length": 128,
                "RenderAs": null
            },
            {
                "Name": "Count",
                "Type": "int64",
                "Length": 0,
                "RenderAs": null
            }
        ],
        "GroupBy": "Publisher,Product,Version",
        "Operations": [
            {
                "Name": "Count",
                "Type": "count"
            }
        ]
    },
    "KeepRaw": true,
    "Scope": null,
    "InstructionTtlMinutes": 10,
    "ResponseTtlMinutes": 30,
    "CreatedTimestampUtc": "2018-04-20T08:03:43.893Z",
    "SentTimestampUtc": "2018-04-20T08:03:44.333Z",
    "Status": 5,
    "WorkflowState": 4,
    "StatusTimestampUtc": "2018-04-20T08:43:45.417Z",
    "Export": false,
    "ExportLocation": "",
    "ParentInstructionId": 2014,
    "InstructionDefinitionId": 1051,
    "CreatedBy": "SomeDomain\\SomeUser",
    "ResultsFilter": {
        "Attribute": "Publisher",
        "Operator": "Like",
        "Value": "%Microsoft%",
        "DataType": "string"
    },
    "PreviousResultsFilter": {
        "Attribute": "DHCPServer",
        "Operator": "==",
        "Value": "192.168.2.5",
```

```
        "DataType": "string"
    },
    "ConsumerId": 1,
    "ConsumerName": "Explorer",
    "ConsumerCustomData": null,
    "ParameterJson": null,
    "OffloadResponses": false,
    "RequestedFor": null,
    "ResponseTemplateId": 1,
    "ResponseTemplateConfiguration": {
        "Name": "default",
        "TemplateConfigurations": [
            {
                "Id": "mainchart",
                "Title": "Installed software - 5 most common
application from 5 most common publishers",
                "Type": "Bar",
                "X": "Product",
                "Y": "Count",
                "Z": "Publisher",
                "PostProcessor": "processingFunction",
                "Size": 1,
                "Row": 1
            }
        ],
        "PostProcessors": [
            {
                "Name": "processingFunction",
                "Function": "ProcessMultiSeries('Product',
'Count', 'Publisher', '5', '5', 'false')"
            }
        ]
    },
    "Workflow": "{\"StateMachine\":\"State\"}",
    "Comments": null,
    "ScheduledInstructionId": null,
    "ApprovalOffloaded": false,
    "ActionedBy": null,
    "ActionReason": null
}
```


The Status of the instruction is 5. Looking at the table above this means "InProgress". Because the instruction's in progress, you can looking at its statistics and retrieve responses.

Checking instruction statistics

You can get statistics from https://my.tachyon.server/Consumer/InstructionStatistics API. You can also provide an Id of the instruction you're interested in to get its statistics based.

The code below gets statistics for the issued instruction when using the results filters.

Direct Consumer API call	C# code using Consumer SDK library

Making a GET request to https://my.tachyon.server/Consumer/InstructionStatistics/2015 returns this response:

GET to https://my.tachyon.server/Consumer/InstructionStatistics/2015


```
{
    "Id": 2015,
    "EstimatedCount": 1,
    "ReceivedCount": 1,
    "SentCount": 1,
    "OutstandingResponsesCount": 0,
    "AverageExecTime": "00:00:03.3950000",
    "Runningtime": "00:10:00",
    "TotalBytesSent": 538,
    "TotalBytesReceived": 16942,
    "AverageBytesReceived": 16942,
    "TotalRowInserts": 1614,
    "TotalRowProcessed": 1662,
    "TotalSuccessRespondents": 1,
    "TotalSuccessNoDataRespondents": 0,
    "TotalErrorRespondents": 0,
    "TotalNotImplementedRespondents": 0,
    "TotalResponseTooLargeRespondents": 0,
    "TotalSubscribedEventsRespondents": 0,
    "InstructionId": 2015,
    "InstructionDefinitionId": 1051,
    "Sequence": 24
}
```


Use InstructionStatistics object inside the Tachyon connector instance.

Retrieving statistics for an Instruction

```
var stats = connector.InstructionStatistics.
Get(2015);
```

stats object will contain the same data you can see in the JSON response on the left.

Returned fields:

Property	Description
EstimatedCount	Estimates count of devices the instruction would be sent to.
ReceivedCount	Exact count of devices that responded.
SentCount	Exact count of devices the instruction was sent to.
OutstandingResponsesCount	Exact count of devices that are yet to respond.
AverageExecTime	Average time Agent took to process the instruction.
Runningtime	How long this instruction has been 'live' for.
TotalBytesSent	Total number of bytes sent.
TotalBytesReceived	Total number of bytes received.
AverageBytesReceived	Average number of bytes received from each device.
TotalRowInserts	Total number of rows inserted into the database.
TotalRowProcessed	Total number of rows processed.
TotalSuccessRespondents	Total number of devices that reported "success".
TotalSuccessNoDataRespondents	Total number of devices that reported they've successfully processed the instruction but have no data to send back.
TotalErrorRespondents	Total number of devices that reported error.
TotalNotImplementedRespondents	Total number of devices that reported they cannot execute the instruction because it contains functionality not implemented on their operating system.
TotalResponseTooLargeRespondents	Total number of devices that reported they couldn't send a response because it was too large

TotalSubscribedEventsRespondents	Tachyon versions to 4.1 inclusive. Total number of devices that reported they have subscribed to the event. Only valid for event type instructions.
InstructionId	Id of the Instruction these statistics belong to.
InstructionDefinitionId	Id of the Instruction Definition the Instruction to whom these statistics belong to was based on
Sequence	Sequence number of the Instruction. This may be different from the Id because instructions can be sent to agents in a different order to the one they were issued in. This depends on the need to authorize or authenticate instructions.

To track the progress of the instruction, look at the Total[...]Respondents properties and compare the values with previous statistics. If the numbers have increased, there are new responses available.

Retrieving responses

Responses become available once the instruction has gone "live", so when its Status has changed to "InProgress". When you ask for responses, you'll receive the ones already stored in the system, but until the instruction has finished or been cancelled, there might be more responses coming in. Because some instructions can take more than a few seconds to execute, depending on how complex they are and what they are doing, you might not see responses immediately, but after a short delay.

You should also take into account possible delays in instruction workflow, such as Tachon waiting for the issuing user to authenticate when issuing an instruction or waiting for instruction (usually action) to be approved. In general, if you're issuing an action, you should not expect it to immediatelly go 'live' (unless it uses a custom workflow which bypases authentication and/or approval) and instead you should periodicaly check if an instruction has gone 'live' i.e. its status has changed to 'InProgress'. It makes sense to start retireving responses only once that happened . An instruction that a user fails to authenticate or that is rejected during approval process will not go live and won't have any responses. If an instruction is forgotten, it can potentially be "stuck" in approval forever.

Once the instruction is live, you can start retrieving responses. If you wish to use polling mechanism to retrieve responses, there are two ways you can do this:

- Calling the Responses API to simply request responses, if there's nothing new, you'll receive an empty array of responses.
- Checking Instruction Statistics and examining the "TotalSuccessRespondents" property to see if it's increased, this would mean new agents have responded. In this case you should remember that even if this statistic doesn't change, there still might be responses you haven't fetched because they didn't fit on the last page you requested.

Using the checking Instruction Statistics option, gives you information about all types of responses, not just successful ones. If you poll for responses you will only receive successful responses as there's another API that needs to be called to retrieve other kinds of responses (see "Other responses" below). Checking the statistics helps you make a decision on which responses to retrieve and when to retrieve them.

You can aggregate instructions where by default only aggregated data is kept. To also have access to the raw data used to make the aggregates, you need to set the "KeepRaw" flag to true when issuing an instruction. If this flag is set to false, retrieving raw responses will not return an error. Instead, it will always return an empty array. Asking for aggregated responses for an instruction that's not aggregated results in an error.

Once the instruction ttl has elapsed, no new responses will be received from agents, but it might also happen that an instruction is cancelled before its ttl elapses. In this case, depending on whether the cancellation request specified that responses should be kept or not, responses might not be available after the cancellation takes effect.

When you are pooliing for responses you should be mindful of changes in the status of the instruction. If the instruction is allowed to run its course, the status will change from InProgress (5) to Complete (6). In this case, pooling for the duration od instruction ttl is fine. If the instruction is cancelled you should stop pooling when the status changes. If the status changed to Cancelling (8) then you can still retrieve responses. If the status changed to Cancelled (9) then you won't be able to retrieve responses because they have been removed when the instruction was cancelled.

You can get successful responses from:

- https://my.tachyon.server/Consumer/Responses/{instructionId}/ - for raw
- https://my.tachyon.server/Consumer/Responses/{instructionId}/Aggregate - for aggregated.

Other responses are available from:

- https://my.tachyon.server/Consumer/ResponseErrors/{instructionId} - for raw
- https://my.tachyon.server/Consumer/ResponseErrors/{instructionId}/Aggregate - for aggregated.

Those endpoints require similar payloads and return similar data. The main difference between aggregated and non aggregated is that aggregation does not support pagination.

Pagination

The above paragraph focused on pooling in the context of establishing if new responses are available, but that is just one aspect of the issue.

You should also be mindful of the number of responses you retrieve each time you ask for them. If your page size is small, you might still have responses to retrieve even though no new responses have been received.

Let's look at an example estate with 5 agents. An instruction is issued to all agents and each of them returns 20 rows. This means you have 100 responses. If you ask for 25 responses, you will receive 25 rows and even though no new responses are received from agents, you still have 75 responses to be retrieved from the API.

Agents can return many, even more than a hundred, rows in response to a single instruction, like an instruction that returns all installed software. TotalSuccessRespondents has the count of agents that have responded and not how many rows have been returned.

In an example at the end of this page you will see TotalSuccessRespondents used as page size, but that assumes that only 1 row is returned by the agent.

How can I tell if I'm "done" with responses?

This is a very common question and here we'll try to answer the it in its strictest meaning - "When can I be sure there will be no more responses to retrieve?". I many cases, this isn't quite what you might be after, as you can decide you don't need any more responses after a certain percentage of agents have sent a response, or a particular agent has replied. We will not delve into these cases, since they represent particular problems and it should be up you, the user, to decide when you think you are "done" with responses. Instead, we will focus on determining when we've retrieved all possible responses for an instruction.

In general, while the instruction is 'live', so its status is InProgress (5) and it is within its ttl, you should assume that more responses can appear at any time. Even if the statistics report that you've sent the instruction to 10 devices and 10 have replied, there might be 11th coming online just before ttl expires and it will process the instruction and send a response.

The only exception to this rule is if you're sending a targeted instruction (using FQDNs) and you can see that all targeted devices have responded. Then, because you specified exactly who should respond to the instruction and they all have responded, you can assume you're "done", even if instruction is still InProgress (in this instance it might be worth cancelling the instruction because you already have all the possible results).

If the instruction is no longer live then no new responses will be received because the instruction will not be sent to any new agents. By this stage aggregated responses would reach their final values. For raw responses, you should pick a page size that is appropriate for your needs and keep retrieving pages of responses up until the point where you receive fewer responses than the size of the page you defined. This would mean that because there weren't enough responses to fill the page, and no new responses will be coming in because the instruction is no longer live, you have retrieved all available responses. You can also retrieve everything in one go by specifying "0;0" start marker and maximum value allowed for an integer as page size but please remember that this might return a lot of data and pagination is usually preferred method of retrieving raw responses.

How can I tell if my instruction is aggregated or not?

Look at the "Aggregation" property of either Instruction or Instruction Definition. If it's null, the Instruction is not aggregated.

Raw responses

Raw responses are available for all instructions without aggregation. They may also be available for aggregated instructions if the 'KeepRaw' flag was set to true when they were issued. Because a single endpoint may return multiple response rows and there could be hundreds of thousands of connected endpoints, raw responses are designed to be retrieved using a paging mechanism. Because you don't know how may rows each endpoint has returned, you can just keep pooling Responses API.

When you ask for responses, you have to supply a start marker inside the "Start" property. If you'd like to start from the beginning (for example, the first response row) this marker should have the value of "0;0". Each time you receive responses, the object you receive will have, apart from responses themselves, a "Range" field.

The "Range" field is a marker denoting where the reading of responses finished. Next time you ask for responses, if you want to resume where you left off, supply the value you received as "Range" in "Start" property of the next call. When you get the next "page" of response you'll get another "Range" marker you can supply as "Start" in a subsequent call.

Direct Consumer API call	**C# code using Consumer SDK library**

Making a POST request with following payload:

Payload sent to https://my.tachyon.server/Consumer/Responses/2016

```
{
    "Filter":null,
    "Start": "0;0",
    "PageSize": 20
}
```

Returns the following response:

Return payload

```
{
    "Range": "MTsy",
    "Responses": [
        {
            "Id": 1,
            "ShardId": 1,
            "TachyonGuid": "4c4c4544-0051-5a10-
8058-b6c04f47354a",
            "Fqdn": "somemachine.somedomain",
            "Status": 0,
            "ExecTime": 113,
            "CreatedTimestampUtc": "2018-04-23T08:
42:06.24Z",
            "ResponseTimestampUtc": "2018-04-23T08:
42:05Z",
            "Values": {
                "Message": "hello! from
somemachine"
            },
            "Blob": null
        }
    ]
}
```

In the next POST call, use 'Range' from the response as value for "Start" inside the payload sent with the request:

2nd call with payload sent again to https://my.tachyon.server/Consumer/Responses/2016

```
{
    "Filter":null,
    "Start": "MTsy",
    "PageSize": 20
}
```

to receive the following response.

Return payload

```
{
    "Range": "MTsy",
    "Responses": []
}
```

In this case, there were no further responses.

Use Responses object inside the Tachyon connector instance.

Retrieving non aggregated responses in a loop

```
var pageSize = 10;
var startMarker = "0;0";
while (loopCondition)
{
   var responses = connector.Responses.GetResponses
(2016, startMarker, pageSize);
   if (responses.Success)
   {
      startMarker = responses.ReceivedObject.Range;
      if (responses.ReceivedObject.Responses.Any())
      {
         foreach (var response in responses.
ReceivedObject.Responses)
         {
            // display responses
         }
      }
   }
   else
   {
      foreach (var error in responses.Errors)
      {
         // display error
         // break the loop or wait to retry
      }
   }
   // wait a time interval
}
```

Aggregated responses

Aggregated responses should be far fewer than raw responses (unless the aggregation has been badly designed) and they do not support pagination.

Aggregated values will be re-calculated when another agent responds. This means that examining "TotalSuccessRespondents" will give you an indication when to get the aggregated values again. If the count has changed, it means another agent has responded and the values most likely changed, so you might want to retrieve them.

You can specify pagesize in the request, but this simply limits the number of responses retrieved. There's no start marker in the request or "Range" property in the response so you cannot request the next X responses. You always have to start from the beginning because when another agent responds, the aggregated values are re-calculated so the values you have already fetched would have changed.

Direct Consumer API call

Making a POST request with following payload:

Payload sent to https://my.tachyon.server/Consumer/Responses/2018/Aggregate

```
{
    "Filter":null,
    "PageSize": 20
}
```

will yield following response:

Return payload

```
{
    "Range": "0;0",
    "Responses": [{
        "Id": 1,
        "ShardId": 1,
        "TachyonGuid": "00000000-0000-0000-0000-
000000000000",
        "Fqdn": null,
        "Status": 0,
        "ExecTime": 0,
        "CreatedTimestampUtc": "0001-01-01T00:00:00Z",
        "ResponseTimestampUtc": "1970-01-01T00:00:
00Z",
        "Values": {
            "Publisher": "",
            "Product": "hMailServer 5.6.6-B2383",
            "Version": "",
            "Count": 1
        },
        "Blob": null
    },
    {
        "Id": 2,
        "ShardId": 1,
        "TachyonGuid": "00000000-0000-0000-0000-
000000000000",
        "Fqdn": null,
        "Status": 0,
        "ExecTime": 0,
        "CreatedTimestampUtc": "0001-01-01T00:00:00Z",
        "ResponseTimestampUtc": "1970-01-01T00:00:
00Z",
        "Values": {
            "Publisher": "",
            "Product": "IcoFX 2.12",
            "Version": "2.12",
            "Count": 1
        },
        "Blob": null
    },
```

C# code using Consumer SDK library

Use Responses object inside the Tachyon connector instance.

Retrieving Aggregated responses in a loop

```
while (loopCondition)
{
   var responses = connector.Responses.
GetAllAggregatedResponses(2018);
   if (responses.Success)
   {
      if (responses.ReceivedObject.Responses.
Any())
      {
         foreach (var response in responses.
ReceivedObject.Responses)
         {
            // display responses
         }
      }
   }
   else
   {
      foreach (var error in responses.Errors)
      {
         // display error
         // break the loop or wait to retry
      }
   }
}
```

```
    {
        "Id": 3,
        "ShardId": 1,
        "TachyonGuid": "00000000-0000-0000-0000-
000000000000",
        "Fqdn": null,
        "Status": 0,
        "ExecTime": 0,
        "CreatedTimestampUtc": "0001-01-01T00:00:00Z",
        "ResponseTimestampUtc": "1970-01-01T00:00:
00Z",
        "Values": {
            "Publisher": "",
            "Product": "IIS Express Application
Compatibility Database for x64",
            "Version": "",
            "Count": 1
        },
        "Blob": null
    },
    {
        "Id": 4,
        "ShardId": 1,
        "TachyonGuid": "00000000-0000-0000-0000-
000000000000",
        "Fqdn": null,
        "Status": 0,
        "ExecTime": 0,
        "CreatedTimestampUtc": "0001-01-01T00:00:00Z",
        "ResponseTimestampUtc": "1970-01-01T00:00:
00Z",
        "Values": {
            "Publisher": "",
            "Product": "IIS Express Application
Compatibility Database for x86",
            "Version": "",
            "Count": 1
        },
        "Blob": null
    }]
}
```


In the previous example, you can see the Fqdn field is null. This is because you're looking at aggregated data, meaning it's not coming from a single agent. Read the following sections for more details about the return object.

Other responses

Any response that does not yield data counts as an "Other" response. These responses are held in a separate table and do not conform to either regular or aggregation schema.

When an agent sends a reply to an instruction, the reply can have one of 5 statuses:

Numerical value	Name	Description
0	Success	Instruction executed successfully and data was returned.
1	SuccessNoContent	Instruction executed successfully but there was no data to return.
2	Error	An error occurred while executing the instruction.
3	NotImplemented	Instruction required functionality that's not implemented on the platform the Agent is running on.
4	ResponseTooLarge	Response to the instruction was too large to be sent. This is based on the Switch's settings of how large the payload can be.

Every status other than Success (0) counts as "Other" response.

You can retrieve these responses by calling:

https://my.tachyon.server/Consumer/ResponseErrors/{instructionId}/

Or in aggregated form by calling:

https://my.tachyon.server/Consumer/ResponseErrors/{instructionId}/Aggregate

Where they'll be aggregated by the Error Data. This helps finding common issues as they'll likely have the same error data.

Other responses are paginated the same way regular responses are. You use the "Range" property returned by the call as the value "Start" property of the next call, exactly the same way as you would with regular responses. Aggregated other responses are not paginated, so this behavior mimics the regular responses.

Raw other responses can be retrieved like this:

Direct Consumer API call

Making a POST request with following payload:

Payload sent to https://my.tachyon.server/Consumer/ResponseErrors/2017

```
{
    "Start": "0;0",
    "PageSize": 20
}
```

Returns this response:

Return payload

```
{
    "Range": "MTs2",
    "Responses": [
        {
            "Id": 1,
            "ShardId": 1,
            "TachyonGuid": "a4f14ba5-97f5-
f64c-b3ff-85d2f5d59cee",
            "Fqdn": "somemachine1.
somedomain",
            "Status": 3,
            "ExecTime": 0,
            "ErrorData": "WMI not supported",
            "CreatedTimestampUtc": "2018-04-
23T13:10:45.9Z",
            "ResponseTimestampUtc": "2018-04-
23T13:10:45Z"
        },
        {
            "Id": 2,
            "ShardId": 1,
            "TachyonGuid": "abf0b756-1ace-
3e4a-a8f5-7b219974f77a",
            "Fqdn": "somemachine2.
somedomain",
            "Status": 3,
            "ExecTime": 0,
            "ErrorData": "WMI not supported",
            "CreatedTimestampUtc": "2018-04-
23T13:10:45.9Z",
            "ResponseTimestampUtc": "2018-04-
23T13:10:46Z"
        },
        {
            "Id": 3,
            "ShardId": 1,
```

C# code using Consumer SDK library

Use ResponseErrors object inside the Tachyon connector instance.

Retrieving other responses

```
var pageSize = 10;
var startMarker = "0;0";
while (loopCondition)
{
   var responses = connector.ResponseErrors.
GetResponseErrors(2017, "0;0", pageSize)
   if (responses.Success)
   {
      startMarker = responses.ReceivedObject.Range;
      if (responses.ReceivedObject.Responses.Any())
      {
         foreach (var response in responses.
ReceivedObject.Responses)
         {
            // display response errors
         }
      }
   }
   else
   {
      foreach (var error in responses.Errors)
      {
         // display error
         // break the loop or wait to retry
      }
   }
}
```

responses.ReceivedObject object will contain the same data you can see in the Json response on the left.

```
            "TachyonGuid": "e28ff7ee-5d15-
ec46-94f5-3a2e5a9dde77",
            "Fqdn": "somemachine3.
somedomain",
            "Status": 3,
            "ExecTime": 0,
            "ErrorData": "WMI not supported",
            "CreatedTimestampUtc": "2018-04-
23T13:10:45.9Z",
            "ResponseTimestampUtc": "2018-04-
23T13:10:45Z"
        },
        {
            "Id": 4,
            "ShardId": 1,
            "TachyonGuid": "ae843206-f183-
7242-a88c-b408354f2388",
            "Fqdn": "somemachine4.
somedomain",
            "Status": 3,
            "ExecTime": 0,
            "ErrorData": "WMI not supported",
            "CreatedTimestampUtc": "2018-04-
23T13:10:45.9Z",
            "ResponseTimestampUtc": "2018-04-
23T13:10:45Z"
        },
        {
            "Id": 5,
            "ShardId": 1,
            "TachyonGuid": "b1ff281e-d97d-
764b-ac42-98dee0619a93",
            "Fqdn": "somemachine5.
somedomain",
            "Status": 3,
            "ExecTime": 830,
            "ErrorData": "WMI not supported",
            "CreatedTimestampUtc": "2018-04-
23T13:11:02.123Z",
            "ResponseTimestampUtc": "2018-04-
23T13:10:53Z"
        }
    ]
}
```


And aggregates like this:

Direct Consumer API call	C# code using Consumer SDK library

Making a POST request with following payload:

Payload sent to https://my.tachyon.server/Consumer/ResponseErrors/2017/Aggregate

```
{
    "PageSize": 20
}
```

will yield following response:

Return payload

```
[
    {
        "ErrorData": "WMI not supported",
        "Count": 5
    }
]
```

Use ResponseErrors object inside the Tachyon connector instance.

Retrieving aggregated other responses

```
while (loopCondition)
{
   var responses = connector.ResponseErrors.
GetAllAggregatedResponseErrors(2017);
   if (responses.Success)
   {
      if (responses.ReceivedObject.Any())
      {
         foreach (var response in responses.
ReceivedObject)
         {
            // display response errors
         }
      }
   }
   else
   {
      foreach (var error in responses.Errors)
      {
         // display error
         // break the loop or wait to retry
      }
   }
}
```

responses.ReceivedObject object will contain the same data you can see in the JSON response on the left.

Examining a Response object

Raw and aggregated responses, as well as raw "other" responses, return objects similar to one another:

Property	Description
Id	Id of the response. Will be unique within the shard.
ShardId	Id of the shard this Id comes from.
TachyonGuid	Tachyon GUID of the Agent who sent the response. Not valid for aggregated responses.
Fqdn	Fqdn of the Agent who sent the response. Not valid for aggregated responses.
Status	Status (see table above).
ExecTime	Instruction's execution time.
CreatedTimestampUtc	Timestamp when this response was created on the Agent.
ResponseTimestampUtc	Timestamp when this response was received by the Core API.

In addition, raw and aggregated responses have two more properties:

Property	Description
Values	Collection of key value pairs, where keys are columns from the instruction's schema.
Blob	This field would hold the entire response received from Agent for instructions that do not have a schema.

raw other responses have one extra property:

Property	Description
ErrorData	Error text. This will usually be a description of the error but can be a generic text with an expanded and more readable version of the "Status" field

Reading responses

Previous paragraph shows the response object in its generic form, but the really interesting part of it is the "Values" property.

While “Values” will always be there what is inside it is dictated by the instruction’s schema.

The name of the schema field will be the name of the property in the response’s “Values” object, and the value of the field in the response’s Value will be the value provided by the agent.

Assuming we have an instruction with following schema:

Schema

```
[
    {
        "Name": "BIOSVersion",
        "Type": "string",
        "Length": 512
    },
    {
        "Name": "Caption",
        "Type": "string",
        "Length": 512
    },
    {
        "Name": "Manufacturer",
        "Type": "string",
        "Length": 512
    },
    {
        "Name": "PrimaryBIOS",
        "Type": "bool",
        "Length": 0
    },
    {
        "Name": "ReleaseDate",
        "Type": "datetime",
        "Length": 0
    },
    {
        "Name": "SerialNumber",
        "Type": "string",
        "Length": 512
    }
]
```

The response object will look like this:

Sample response


```
{
        "Id": 1,
        "ShardId": 1,
        "TachyonGuid": "63fc4911-f6ca-463c-9bf9-4497bf88be1f",
        "Fqdn": "CLIENT-WIN7X64.TachyonDomain.Lab",
        "Status": 0,
        "ExecTime": 15,
        "CreatedTimestampUtc": "2020-02-11T14:47:17.307Z",
        "ResponseTimestampUtc": "2020-02-11T14:47:17Z",
        "Values": {
                "BIOSVersion": "VRTUAL - 12001807, Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz, BIOS Date: 12
/07/18 15:46:29  Ver: 09.00.08, BIOS Date: 12/07/18 15:46:29  Ver: 09.00.08",
                "Caption": "Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz",
                "Manufacturer": "American Megatrends Inc.",
                "PrimaryBIOS": true,
                "ReleaseDate": "2018-12-07T00:00:00+00:00",
                "SerialNumber": "8546-4968-3506-2716-2087-9169-05"
        },
        "Blob": null
}
```


As you can see, there's a property inside the "Values" for each schema object and the name of that property is the same as the Name of the object in the schema.

This means that this entry in the Values object:


```
"SerialNumber": "8546-4968-3506-2716-2087-9169-05"
```


Represents this schema field:


```
{
    "Name": "SerialNumber",
    "Type": "string",
    "Length": 512
}
```


Switching from high level to detailed view of responses

Using aggregated and raw data in combination gives an opportunity to present the data as an overview with the ability to see details for the chosen aggregate if needed. This feature is commonly referred to as "drill down" in Explorer, where one can expand an aggregated row to see all the raw responses that make up the aggregated row.

Both sets of data are required for this to work so the instruction has to have "KeepRaw" set to true.

This example issues an instruction based on "1E-Explorer-TachyonCore-AllInstalledSoftware" instruction definition, which looks like:

Instruction definition 1E-Explorer-TachyonCore-AllInstalledSoftware


```
{
    "Id": 1051,
    "Name": "1E-Explorer-TachyonCore-AllInstalledSoftware",
    "Description": "Returns all installed software.",
    "InstructionSetId": 2,
    "InstructionSetName": "Everything",
    "InstructionType": 0,
    "ReadablePayload": "What software is installed?",
    "Parameters": null,
    "Schema": [
        {
            "Name": "Product",
            "Type": "string",
            "Length": 512,
            "RenderAs": null
        },
```

```
        {
            "Name": "Publisher",
            "Type": "string",
            "Length": 512,
            "RenderAs": null
        },
        {
            "Name": "Version",
            "Type": "string",
            "Length": 128,
            "RenderAs": null
        },
        {
            "Name": "InstallDate",
            "Type": "datetime",
            "Length": 0,
            "RenderAs": null
        },
        {
            "Name": "Architecture",
            "Type": "string",
            "Length": 20,
            "RenderAs": null
        }
    ],
    "Aggregation": {
        "Schema": [
            {
                "Name": "Publisher",
                "Type": "string",
                "Length": 512,
                "RenderAs": null
            },
            {
                "Name": "Product",
                "Type": "string",
                "Length": 512,
                "RenderAs": null
            },
            {
                "Name": "Version",
                "Type": "string",
                "Length": 128,
                "RenderAs": null
            },
            {
                "Name": "Count",
                "Type": "int64",
                "Length": 0,
                "RenderAs": null
            }
        ],
        "GroupBy": "Publisher,Product,Version",
        "Operations": [
            {
                "Name": "Count",
                "Type": "count"
            }
        ]
    },
    "InstructionTtlMinutes": 10,
    "ResponseTtlMinutes": 30,
    "MinimumInstructionTtlMinutes": 10,
    "MaximumInstructionTtlMinutes": 1440,
    "MinimumResponseTtlMinutes": 10,
    "MaximumResponseTtlMinutes": 10080,
    "Workflow": null,
    "ResponseTemplateId": 1,
    "OriginalFileName": null,
    "ResponseTemplateConfiguration": {
        "Name": "default",
```

```
        "TemplateConfigurations": [
            {
                "Id": "mainchart",
                "Title": "Installed software - 5 most common application from 5 most common publishers",
                "Type": "Bar",
                "X": "Product",
                "Y": "Count",
                "Z": "Publisher",
                "PostProcessor": "processingFunction",
                "Size": 1,
                "Row": 1
            }
        ],
        "PostProcessors": [
            {
                "Name": "processingFunction",
                "Function": "ProcessMultiSeries('Product', 'Count', 'Publisher', '5', '5', 'false')"
            }
        ]
    },
    "Version": "4",
    "Author": "1E",
    "IsLicensed": true,
    "UploadedTimestampUtc": "2018-04-09T13:33:30.807Z",
    "NumberOfTimesExecuted": 4
}
```

To get the overview, request aggregated responses, then pick a row and request the raw responses used to form that row. Do this by looking at the aggreated column values and using them as a filter for the raw data request.

Direct Consumer API call	C# code using Consumer SDK library

Making a POST request with following payload:
 	
Payload sent to https://my.tachyon.server/Consumer/Responses/2020/Aggregate
 	
`{ "PageSize":20 }`
 	
Returns this response containing aggregated responses:
 	
Return payload with aggregated responses
 	
`{ "Range": "0;0", "Responses": [{ "Id": 1, "ShardId": 1, "TachyonGuid": "00000000-0000-0000-0000-000000000000", "Fqdn": null, "Status": 0, "ExecTime": 0, "CreatedTimestampUtc": "0001-01-01T00:00:00Z", "ResponseTimestampUtc": "1970-01-01T00:00:00Z", "Values": { "Publisher": "1E", "Product": "1E Agent", "Version": "7.2.0", "Count": 1 }, "Blob": null`
	You can create a filter explicitly like:

```
    },
    {
        "Id": 2,
        "ShardId": 1,
        "TachyonGuid": "00000000-
0000-0000-0000-000000000000",
        "Fqdn": null,
        "Status": 0,
        "ExecTime": 0,
        "CreatedTimestampUtc":
"0001-01-01T00:00:00Z",
        "ResponseTimestampUtc":
"1970-01-01T00:00:00Z",
        "Values": {
            "Publisher": "1E",
            "Product": "1E
NomadBranch x64",
            "Version": "6.3.100",
            "Count": 1
        },
        "Blob": null
    },
    {
        "Id": 3,
        "ShardId": 1,
        "TachyonGuid": "00000000-
0000-0000-0000-000000000000",
        "Fqdn": null,
        "Status": 0,
        "ExecTime": 0,
        "CreatedTimestampUtc":
"0001-01-01T00:00:00Z",
        "ResponseTimestampUtc":
"1970-01-01T00:00:00Z",
        "Values": {
            "Publisher": "1E",
            "Product": "1E PXE Lite
Local",
            "Version": "3.1.200",
            "Count": 1
        },
        "Blob": null
    },
    {
        "Id": 4,
        "ShardId": 1,
        "TachyonGuid": "00000000-
0000-0000-0000-000000000000",
        "Fqdn": null,
        "Status": 0,
        "ExecTime": 0,
        "CreatedTimestampUtc":
"0001-01-01T00:00:00Z",
        "ResponseTimestampUtc":
"1970-01-01T00:00:00Z",
        "Values": {
            "Publisher": "1E",
            "Product": "1E Shopping
Agent",
            "Version": "1.0.200",
            "Count": 1
        },
        "Blob": null
    },
    {
        "Id": 5,
        "ShardId": 1,
        "TachyonGuid": "00000000-
0000-0000-0000-000000000000",
        "Fqdn": null,
        "Status": 0,
```

Drill down example with explicit details filter

```
var pageSize = 20;
var startMarker = "0;0";

var definition = connector.InstructionDefinitions.
GetInstructionDefinition(1051);
var instruction = connector.Instructions.SendInstruction
(definition.ReceivedObject.Id, null, 60, 120);
var aggregated = connector.Responses.GetAllAggregatedResponses
(instruction.ReceivedObject.Id);
var selectedRow = aggregated.ReceivedObject.Responses.
FirstOrDefault(p => p.Id == 1);
if (selectedRow != null)
{
    var responsesSearch = new Responses
    {
        PageSize = pageSize,
        Start = startMarker,
        Filter = new ExpressionObject
        {
            Operator = "AND",
            Operands = new List<ExpressionObject>
            {
                new ExpressionObject
                {
                    Attribute = "Publisher",
                    Operator = "==",
                    Value = "1E"
                },
                new ExpressionObject
                {
                    Attribute = "Product",
                    Operator = "==",
                    Value = "1E Agent"
                },
                new ExpressionObject
                {
                    Attribute = "Version",
                    Operator = "==",
                    Value = "7.2.0"
                }
            }
        }
    };

    var detailed = connector.Responses.GetResponses(instruction.
ReceivedObject.Id, responsesSearch);
}
```

or be more general by examining the group by columns from the original instruction definition schema:

```
        "ExecTime": 0,
        "CreatedTimestampUtc":
"0001-01-01T00:00:00Z",
        "ResponseTimestampUtc":
"1970-01-01T00:00:00Z",
        "Values": {
            "Publisher": "1E",
            "Product": "1E Shopping
Client Identity",
            "Version": "5.1.0",
            "Count": 1
        },
        "Blob": null
    },
    {
        "Id": 6,
        "ShardId": 1,
        "TachyonGuid": "00000000-
0000-0000-0000-000000000000",
        "Fqdn": null,
        "Status": 0,
        "ExecTime": 0,
        "CreatedTimestampUtc":
"0001-01-01T00:00:00Z",
        "ResponseTimestampUtc":
"1970-01-01T00:00:00Z",
        "Values": {
            "Publisher": "1E",
            "Product": "1E Tachyon
Agent x64",
            "Version": "3.1.0",
            "Count": 1
        },
        "Blob": null
    }]
}
```

From this list we will pick a row, for instance the 1E Agent (row Id 1).

From the definition we can see this instruction is aggregated on Publisher, Product and Version, so we can set up a filter for those columns and use it when requesting raw data using a POST request:

Drill down example with automatic aggregated column selection for the filter

```
var pageSize = 20;
var startMarker = "0;0";

var definition = connector.InstructionDefinitions.
GetInstructionDefinition(1051);
var instruction = connector.Instructions.SendInstruction
(definition.ReceivedObject.Id, null, 60, 120);
var aggregated = connector.Responses.GetAllAggregatedResponses
(instruction.ReceivedObject.Id);
var selectedRow = aggregated.ReceivedObject.Responses.
FirstOrDefault(p => p.Id == 1);
if (selectedRow != null)
{
    var filterElements = new List<ExpressionObject>();
    var groupedColumns = definition.ReceivedObject.Aggregation.
GroupBy.Split(',');
    foreach (var column in groupedColumns)
    {
        filterElements.Add(new ExpressionObject
        {
            Operator = "==",
            Attribute = column,
            Value = selectedRow.Values[column].ToString()
        });
    }

    var responsesSearch = new Responses
    {
        PageSize = pageSize,
        Start = startMarker,
        Filter = new ExpressionObject
        {
            Operator = "AND",
            Operands = filterElements
        }
    };

    var detailed = connector.Responses.GetResponses(instruction.
ReceivedObject.Id, responsesSearch);
}
```


The generic approach assumes the equality operator is what you wish to use with each column.

You could also examine the definition.ReceivedObject.Aggregation.Schema collection to get all the aggregated columns. Be aware that some of these columns might contain data computed by the aggregation operation itself, like count of elements. You might not want to filter on those because they do not exist in the original schema for raw data.

To mitigate you could correlate the aggregation schema:

- With the regular schema to make sure you're only searching for columns that do exist in the regular schema
- Columns with the aggregation operations list - each operation produces a column and it's these columns you do not want to filter on.

Payload sent to https://my.tachyon.server/Consumer/Responses/2020

```
{
    "Filter": {
        "Operator": "AND",
        "Operands": [{
            "Attribute":
"Publisher",
            "Operator": "==",
            "Value": "1E"
        },
        {
            "Attribute": "Product",
            "Operator": "==",
            "Value": "1E Agent"
        },
        {
            "Attribute": "Version",
            "Operator": "==",
            "Value": "7.2.0"
        }]
    },
    "Start": "0;0",
    "PageSize": 20
}
```

which returns raw responses matching the filter. These responses will be the rows used to form the aggregated row we picked.

Return payload with filtered raw responses

```
{
    "Range": "MTsyNQ==",
    "Responses": [{
        "Id": 24,
        "ShardId": 1,
        "TachyonGuid": "4c4c4544-
0051-5a10-8058-b6c04f47354a",
        "Fqdn":
"SomeDomain\\SomeMachine",
        "Status": 0,
        "ExecTime": 2991,
        "CreatedTimestampUtc":
"2018-04-24T07:53:49.81Z",
        "ResponseTimestampUtc":
"2018-04-24T07:53:48Z",
        "Values": {
            "Product": "1E Agent",
            "Publisher": "1E",
            "Version": "7.2.0",
            "InstallDate": "2017-09-
06T00:00:00+00:00",
            "Architecture": "x64"
        },
        "Blob": null
    }]
}
```

Processed responses

The last way of retrieving responses is using processed responses.

By making a GET request to https://my.tachyon.server/Consumer/Responses/Processed/{InstructionId} endpoint you can retrieve responses that have been processed according to the rules defined in Instruction definition's ResponseTemplateConfiguration settings.

Explorer uses this feature and aims at providing data in a readily consumable way by the visualization engine the Explorer uses. Read more about visualizations in custom response visualizations.

Cancelling and re-running instructions

Instructions can be cancelled if they have been issued but have not started (for example, because they have not been approved or authenticated) or when they are live (when they're gathering responses).

If the instruction is cancelled before it goes live, it will not be sent to agents and there will be no results to view. There'll still be an entry for this instruction so you'll be able to see what it was.

If the instruction has gone live you, you can stop the instruction instead of cancelling it and you'll have the option to either discard or keep the gathered results. If you choose to:

- Discard the results - the instruction goes to the "Cancelled" state, which is similar to "Expired" because responses are no longer available
- Keep the responses - the instruction goes to the "Cancelling" state, which is similar to "Complete" in a way that responses are still available for the duration of ResponseTtl of the instruction.

To cancel or stop an instruction, make a POST request to https://my.tachyon.server/Consumer/Instructions/{id}/cancel/{keepData}, where id is the Id of the instruction that's to be cancelled and keepData is a boolean flag indicating whether you want to keep or discard responses. The default is to "false", which means discard the responses.

Direct Consumer API call	C# code using Consumer SDK library

POST request to https://my.tachyon.server/Consumer/Instructions/2020 /cancel/true without any payload.
 	
There's no response payload and success HTTP status code will mean the instruction was cancelled.
	Use Instructions object inside the Tachyon connector instance.
	
	Cancelling an instruciton
	
	`var result = connector.Instructions.CancelInstruction(2020, true);`
	`if (result.Success)`
	`{`
	`    // instruction was cancelled`
	`}`

Both cancelling and stopping use the same endpoint. The difference is purely to the state the instruction is in when the cancellation occurs, Explorer will show this action as either Cancel or Stop, which is why we're using these terms in this document.

Re-running instructions

You can rerun instructions that are not live and not pending authentication or authorization. This causes the system to issue a new instruction that is a copy of the instruction you're re-running. This will include things like scope, parameters and filters.

> ⚠ Only questions can be rerun.

To rerun an instruction make a POST request to https://my.tachyon.server/Consumer/Instructions/{id}/rerun where id is the Id of the instruction you want to run again.

Direct Consumer API call	C# code using Consumer SDK library

POST request to https://my.tachyon.server/Consumer/Instructions/2020/rerun without any payload to return this response:
 	
Payload sent to https://my.tachyon.server/Consumer/Instructions/2020 /rerun
 	
`{`
`  "Id": 2022,`
`  "Sequence": 0,`
`  "Name": "1E-Explorer-TachyonCore-AllInstalledSoftware",`
`  "Description": "Returns all installed software.",`
`  "InstructionType": 0,`	

```
"ReadablePayload": "What software is installed?",
"Cmd": "SendAll",
"Schema": [
    {
        "Name": "Product",
        "Type": "string",
        "Length": 512,
        "RenderAs": null
    },
    {
        "Name": "Publisher",
        "Type": "string",
        "Length": 512,
        "RenderAs": null
    },
    {
        "Name": "Version",
        "Type": "string",
        "Length": 128,
        "RenderAs": null
    },
    {
        "Name": "InstallDate",
        "Type": "datetime",
        "Length": 0,
        "RenderAs": null
    },
    {
        "Name": "Architecture",
        "Type": "string",
        "Length": 20,
        "RenderAs": null
    }
],
"Aggregation": {
    "Schema": [
        {
            "Name": "Publisher",
            "Type": "string",
            "Length": 512,
            "RenderAs": null
        },
        {
            "Name": "Product",
            "Type": "string",
            "Length": 512,
            "RenderAs": null
        },
        {
            "Name": "Version",
            "Type": "string",
            "Length": 128,
            "RenderAs": null
        },
        {
            "Name": "Count",
            "Type": "int64",
            "Length": 0,
            "RenderAs": null
        }
    ],
    "GroupBy": "Publisher,Product,Version",
    "Operations": [
        {
            "Name": "Count",
            "Type": "count"
        }
    ]
},
"KeepRaw": true,
"Scope": null,
```


Use Instructions object inside the Tachyon connector instance.

Re-running an instruction

```
var rerunInstruction = connector.
Instructions.RerunInstruction(2020);
if (rerunInstruction.Success)
{
    // rerunInstruction.ReceivedObject has
the newly issued instruction
}
else
{
    foreach (var error in rerunInstruction.
Errors)
    {
        // display error
        // break the loop or wait to retry
    }
}
```

rerunInstruction.ReceivedObject object will contain the same data you can see in the JSON response on the left.

```
    "InstructionTtlMinutes": 10,
    "ResponseTtlMinutes": 30,
    "CreatedTimestampUtc": "2018-04-25T13:22:49.983Z",
    "SentTimestampUtc": "2018-04-24T07:53:45.217Z",
    "Status": 0,
    "WorkflowState": 0,
    "StatusTimestampUtc": "2018-04-24T08:33:46.43Z",
    "Export": false,
    "ExportLocation": "",
    "ParentInstructionId": null,
    "InstructionDefinitionId": 1051,
    "CreatedBy": "1E\\Michael.Grzywnowicz",
    "ResultsFilter": null,
    "PreviousResultsFilter": null,
    "ConsumerId": 1,
    "ConsumerName": "Explorer",
    "ConsumerCustomData": null,
    "ParameterJson": null,
    "OffloadResponses": false,
    "RequestedFor": null,
    "ResponseTemplateId": 1,
    "ResponseTemplateConfiguration": {
        "Name": "default",
        "TemplateConfigurations": [
            {
                "Id": "mainchart",
                "Title": "Installed software - 5 most
common application from 5 most common publishers",
                "Type": "Bar",
                "X": "Product",
                "Y": "Count",
                "Z": "Publisher",
                "PostProcessor": "processingFunction",
                "Size": 1,
                "Row": 1
            }
        ],
        "PostProcessors": [
            {
                "Name": "processingFunction",
                "Function": "ProcessMultiSeries
('Product', 'Count', 'Publisher', '5', '5', 'false')"
            }
        ]
    },
    "Workflow": "{\"StateMachine\":\"State\"}",
    "Comments": null,
    "ScheduledInstructionId": null,
    "ApprovalOffloaded": false,
    "ActionedBy": null,
    "ActionReason": null
}
```


Return payload contains details of newly issued instruction.

Remember that the new instruction will be an exact copy of the instruction that was re-run. If you want to change anything, you'll have to issue a new instruction as seen above.

Pseudo-code examples

This section brings together the previous examples with pseudo-code examples of how to do certain things.

These examples describe a concept of how things can be done, and are not designed to be copy-pasted into working code. You should be able to copy and adjust them to the language you're using.

Example 1: Find a machine with specific hardware and perform an action on it.

Company XYZ, who makes network adapters, has announced that a driver for one of their cards called "GreatNIC" has an issue. You want to find all computers in your estate using that model of network adapter and update those computers by installing an updated driver.

You use the following instructions:

- The question which finds network adapters has the Id of 1 and returns a number of columns, among which are Manufacturer and Model
- The action which installs the driver has the Id of 2 and takes one parameter with the pattern of %DRIVERPATH%.

This example assumes that the instructions will run their course and will be authenticated and will not be cancelled or rejected in approval.

```
{
    var instruction = new instance of Instruction payload
    {
        Id = 1,
        InstructionTtlMinutes = 10,
        ResponseTtlMinutes = 60,
        ResultsFilter = {
                Operator = "AND",
                Operands = [{
                        Operator = "==",
                        Attribute = "Manufacturer",
                        Value = "XYZ"
                    },
                    {
                        Operator = "==",
                        Attribute = "Model",
                        Value = "GreatNIC"
                    }
                ]
            }
    }

    var sentInstruction = send the 'instruction' object to Tachyon's Instructions API via a POST and receive
response payload with information about issued instruction.
    sleep for 10 minutes
    // at this point, the instruction should have finished.
    var statistics = get instruction's statistics from Tachyon InstructionStatistics API via a GET and use
sentInstruction.Id as instruciton id.
    // we will assume that no successful response means no devices have the network card we're looking for.
    if (statistics.TotalSuccessRespondents == 0)
        return

    var responseSearch = new instance of object used to retrieve responses
    {
        Start = "0;0",
        PageSize = statistics.TotalSuccessRespondents //we'll assume this number is fairly low and use it to
get every result on one page. This assumes that each Client sends back only 1 row of data.
    }
    var responses = get responses from Tachyon's Responses API via a POST and use sentInstruction.Id as
instruction Id.
    // present a list of devices that will have their drivers updated to a user
    var followUpInstruction = new instance of Instruction payload
    {
        Id = 2,
        ParentInstructionId = sentInstruction.Id,
        InstructionTtlMinutes = 1440,
        ResponseTtlMinutes = 60 minutes,
        Parameters = [{
            Name = "driver",
            Pattern = "%DRIVERPATH%",
            Value = "\\someshare\somefolder\newdriver.exe"
        }]
    }
    var sendAction = send the 'followUpInstruction' object to Tachyon's Instructions API via a POST and
receive response payload with information about issued instruction.
    // because this is an action, it will most likely require authentication and approval, so we should wait
until those have been done and instruction is live.
        while instruction state is not 5
    {
        var followupAction = get the instruction with the id of sendAction.Id
                if followupAction.Status is 5 then break loop
                else sleep for 1 minute // this is just an arbitrary time interval, which should be adjusted
```

```
as needed.
    }
    while instruction is live //in other words, loop for the duration of InstructionTtlMinutes
    {
        var start = "0;0"
        var responseSearch = new instance of object used to retrieve responses
        {
            Start = start,
            PageSize = 25 //we'll use a hard-coded page of 25, but this can be adjusted if needed
        }
        var actionResponses = get responses from Tachyon's Responses API via a POST and use sentAction.Id as
instruction Id.
        // present the list of devices that had their drivers updated to the user
        start = actionResponses.Range
        sleep for 2 minutes // this is just an arbitrary time interval, which should be adjusted as needed.
        check loop condition by examining if time now is greater than instruction's send time plus
instruction ttl.
    }
}
```

Example 2: Retrieving aggregated responses

Assuming that we have issued an instruction with the Id of 12, here's how one could program retrieval of aggregated responses.

```
bool isInstructionLive = false
while isInstructionLive is false
{
    var instruction = get instruction with the Id of 12
    if instruction status is 5
    {
        isInstructionLive = true
    }
    else
    {
        sleep for 60 seconds // this is just an arbitrary time interval, which should be adjusted as needed.
    }
}

bool shouldContinue = true
var receivedCount = 0
while shouldContinue is true
{
    var instructionNow = get instruction with the Id of 12
    stats = get statistics for the instruction with the Id of 12
    if stats.TotalSuccessRespondents is greater then receivedCount
    {
        // Last count of received responses was lower then new count. Fetching aggregated responses.
        var responses = get aggregated responses for the instruction with the Id of 12
        receivedCount = stats.TotalSuccessRespondents
        // process responses
    }

    if instructionNow.Status is not 5 //if the instruction status has changed from 5 to something else
because the instruction ttl elapsed or the instruction has been cancelled.
    {
        var responses = get aggregated responses for the instruction with the Id of 12
        shouldContinue = false
        // process responses
        break while loop
    }

    if shouldContinue is true
        sleep for 60 seconds // this is just an arbitrary time interval, which should be adjusted as needed.
}
```

The basis for this algorithm is that when one is retrieving aggregated responses, the aggregated values can only change when a new device/client sends a response, because this causes the aggregated values to change.

We're not using pagination here because aggregated responses do not support pagination.

Example 3: Retrieving raw responses

Assuming that we have issued an instruction with the Id of 12, and it either has no aggregation of has aggregation but the KeepRaw flag was set to true when the instruction was issued, here's how one could program retrieval of raw responses.

```
bool isInstructionLive = false
while isInstructionLive is false
{
    var instruction = get instruction with the Id of 12
    if instruction status is 5
    {
        isInstructionLive = true
    }
    else
    {
        sleep for 60 seconds // this is just an arbitrary time interval, which should be adjusted as needed.
    }
}

bool shouldContinue = true
var marker = "0;0" // This is a marker telling the Consumer API to start from the beginning
while shouldContinue is true
{
    var responseSearch = new instance of object used to retrieve responses
    {
        Start = marker,
        PageSize = 25 // this is just an arbitrary value, which should be adjusted as needed
    }
    var responses = get raw responses from Tachyon's Responses API via a POST, send responseSearch object as
payload and use 12 as instruction Id.
    marker = response.Range
    // process responses

    var instructionNow = get instruction with the Id of 12
    if instructionNow.Status is not 5 //if the instruction status has changed from 5 to something else
because the instruction ttl elapsed or the instruction has been cancelled.
    {
        shouldContinue = false
        // instruction has finished
        break while loop
    }

    if shouldContinue is true
        sleep for 60 seconds // this is just an arbitrary time interval, which should be adjusted as needed.
}
```

The above example is a scaffoling for retrieving raw responses. There is a crucial point in this example - at the point with the comment saying "instruction has finished".

At this point you have gone through the response retrieval at least once and the instruction is no longer live. This means that there might be responses you still haven't retrieved.

You could just get all responses in one go by changing the pageSize in responseSearch to sufficiently large number (for instance in C# you could use int.MaxValue) and making one last call to retrieve any remaining responses. This, however, may return a large dataset so it is usually preferred to use pagination.

```
responseSearch = new instance of object used to retrieve responses
{
    Start = marker,
    PageSize = maximum value supported by integer
}
var responses = get raw responses from Tachyon's Responses API via a POST, send responseSearch object as
payload and use 12 as instruction Id.
// process responses
```

You also could keep the pagesize as-is and continue looping getting more and more responses up until the point where you receive fewer responses than the page can hold:

```
var fullPage = true
while fullPage is true
{
    var responses = get raw responses from Tachyon's Responses API via a POST, send responseSearch object as payload and use 12 as instruction Id.
    marker = response.Range
    if number of entries responses collection is lower than the page size requested // in this example hard-coded to 25
        fullPage = false
    // process responses

    //You could add a sleep here, but since there are no new responses coming in from agents it might be preferred to simply retrieve and process responsese a page at a time without delay.
}
```

in both cases, the code would be called where the "instruction has finished" comment is located.

Another option is to exit the code completely and loop through all responses again, startign from the beginning, or even retrieve them all in one go.

Which approach to choose depends on your particular use-case.