Getting Started With MTConnect Clients: A Tutorial

Armando Fox et al.
MTConnect Consortium
sdk@mtconnect.org

DRAFT ((Fri, 23 May 2008)) Version 1.0

Contents

1 Introduction

1.1 What Is MTConnect?

MTConnect[1] is a standard that codifies a simple *command set* and *data representation* for the easy exchange of manufacturing data. White papers and overview documents describing MTConnect conceptually can be found at `http://www.mtconnect.org`.

MTConnect is a lightweight standard because it is built on top of mature open standards that are functionality-rich, widely deployed, understood by many programmers, and supported by many development tools. Both MTConnectand all the standards on which it depends are unencumbered from an intellectual property standpoint: specifications can be freely downloaded by anyone (though this isn't necessary in order to successfully use MTConnect), and no royalties are required in order to implement the standards. See `http://www.mtconnect.org` for more detailed information on licensing.

1.2 Purpose and Scope of This Document

Since MTConnect is based entirely on 100% open standards, **no SDK** is necessary to create MTConnect applications. Nonetheless, AMT provides a set of Client Libraries, Utilities and Extensions (CLUE) to facilitate "getting you off the ground" with simple applications that wish to avoid dealing directly with HTTP and XML altogether. The current version of CLUE, described in this tutorial, works with the Microsoft .NET framework. .NET is *not* an open standard and currently runs only on Windows, but we chose to support it because of its wide popularity and customer demand. The CLUE libraries themselves are released under the same liberal open-source license as all other MTConnect software; only .NET itself is proprietary, not the .NET-compatible code provided in the form of the CLUE libraries.

1.2.1 Goals

- A "quick start", tutorial-style guide for engineers wishing to write applications that collect and analyze data from MTConnect-compliant equipment.
- An overview of both the MTConnect "native" API's (using XML and HTTP) and the MTConnect Client Utility Library for Microsoft .NET Framework.

1.2.2 Non-Goals

- **Not a specification:** The definitive MTConnect specification can always be found at `http://www.mtconnect.org`.

[1]MTConnect is a trademark of the Association for Manufacturing Technology. While this document and the software described in it are licensed royalty-free as described in section 1.3, use of the MTConnect trademark must be approved by The Association for Manufacturing Technology, as described on `http://www.mtconnect.org`.

- **Completeness:** While all the examples presented are correct and MTConnect-compliant at the time of writing, we do not use or describe every feature of MTConnect here, nor do we guarantee that these examples will remain correct as the specification evolves.
- The CLUE library is built on top of .NET's excellent support for HTTP, XML and XPath. This document does *not* describe how to write applications using those underlying libraries directly. You are encouraged to browse the heavily-commented (and open source) CLUE code in addition to the wealth of available literature on programming XML and XPath using .NET.
- This guide does *not* explain how to design MTConnect-compliant controllers or create adapters to make legacy controllers MTConnect-compliant. The separate MTConnect Implementor's Guide covers those topics.

1.2.3 Assumptions and Prerequisites

The intended audience of this document should:

- Have basic knowledge of using Web applications
- Have experience developing code in one or more modern programming languages or frameworks (C/C++, C#, Microsoft .Net, Java, etc.)
- Understand the basic concepts of Object-Oriented Programming (OOP) at the level of a language such as Java, C++ or C#.

1.2.4 Topics Covered

1. Section 2 describes the architectural concepts and components behind MTConnect and should be read by everyone who wants to use MTConnect in any way. It describes how to construct and transmit commands to an MTConnect Agent and how to interpret the responses to those commands encoded in XML (eXtensible Markup Language), all using an ordinary Web browser.
2. Section **??** describes how to write client applications for MTConnect using the AMT-supplied Client Library, Utilities and Extensions (CLUE) for Microsoft .NET. The CLUE libraries are not part of the MTConnect specification, but they streamline the task of writing simple MTConnect applications.

1.3 License, Copyright and Trademark Information

1.3.1 Software License

You are free to use the CLUE libraries and any AMT-provided example code in your own commercial or noncommercial products without paying any royalty or notifying AMT, as long as you acknowledge AMT's copyright on the underlying code, agree not to hold AMT responsible for any aspect of the code's behavior, and do not use the name(s) of AMT to endorse or promote your products without AMT's express permission.

(Keep in mind that there are many, many open source implementations of HTTP, XML and XPath libraries available for a variety of platforms; you can also use any of those in lieu of CLUE.)

The following are the terms of the license (also available at `http://www.mtconnect.org`), as of this printing (Fri, 23 May 2008) :

2 Architectural Review of MTConnect

In this section we review the key architectural concepts of MTConnect. It's strongly suggested that you follow the examples by using a Web browser to interact with the live Agent provided by the MTConnect Consortium for developer testing. In the next section we will see how to interact with the agent from a Client program rather than "manually" via a Web browser.

2.1 MTConnect Environment

The MTConnectspecification requires that all MTConnectcommunication take place over HTTP[2] running over TCP/IP. Therefore, Clients can be developed in any programming language that has support for networking, and you can even use a Web browser (which is basically a generic HTTP client) to interact in limited ways with MTConnect devices.

An MTConnect Agent is a software entity that interprets MTConnect commands and returns machine data on behalf of one or more machines that it "represents" in the MTConnect environment (hence the term Agent). The Agent software may be part of the controller's software stack, or it may be a separate piece of software running on a computer that communicates with a controller via a proprietary interface. The MTConnect Consortium operates a publicly-accessible simulator of an MTConnect-enabled vertical CNC mill[3]. If you have access to a Web browser, it's strongly recommended you use it to contact the Test Agent and follow the examples in this guide.

You mean Agents are publicly accessible?We've chosen to make the Test Agent publicly accessible with no passwords, firewalls or other security measures, for the convenience of MTConnect developers. In a production scenario, the Agent would likely be protected by one or more of these standard security measures.

MTConnect data communicated from an Agent to a Client is represented using XML, the eXtensible Markup Language, which provides for a human-readable, ASCII-based representation of hierarchical data. The specific XML constructs permitted in MTConnectare codified using the XML Schema standard, but you don't need to be familiar with XML Schema to use MTConnect. Of course, data pulled from machines can be stored in a non-XML format as well—for example, it can be stored in a relational database or imported into an Excel spreadsheet. But all data communicated between an MTConnect device and its client applications must be represented in XML.

Besides the Agent and the Client, there are two more basic concepts you must understand to get started with MTConnect:

[2]When the name of a protocol or technology appears in this typeface, it means you can look in the Glossary (section **??**) for a concise description of it.

[3]The simulator is based on the open source controller EMC2 (`http://www.linuxcnc.org`).

1. Device schema: how a device, such as a controller attached to a machine tool with various axes and fixtures, describes what data are available to collect.
2. Data representation: the format in which result data is delivered to the requester.

MTConnect 1.0 defines one command, `probe`, that allows a client to discover the device schema of the machine represented by a particular Agent; and two commands, `sample` and `curren`, to actually collect data.

2.2 The Probe Command and the Device Schema

The MTConnect `probe` command is used to interrogate what data an Agent is able to report on behalf of the machine(s) it represents. (The most common case is for an Agent to represent exactly one machine, but it is possible for a single Agent to represent multiple machines.)

Each machine represented by an Agent is called a *device* in MTConnect. Therefore, the response to a probe is a `tree` of components, whose "root" or topmost node, Devices, has *children* (nodes directly connected to it and one level lower in the hierarchy) representing each individual machine (Device). In other words, you can think of the Devices node as a "container" that groups together a logical collection of devices—for example, a cell on a shop floor—whereas each Device node would represent a particular piece of hardware.

2.2.1 Issuing the Probe Command

To issue any MTConnect command to an Agent, you construct a URL by taking the base URL of the agent (in our example, the Test Agent is running at `http://agent.mtconnect.org/LinuxCNC` and appending a slash followed by the command name (in this case, `probe`). In later sections we'll see how to pass arguments to the command, but the `probe` command takes no arguments.

So, point your Web browser at `http://agent.mtconnect.org/LinuxCNC/probe` and you will see the device schema of a simulated vertical CNC mill, which is programmed to repeatedly cut a spiral pattern using the simple GCode program shown in listing **??**. You should see something very similar to listing 4; in the listing we have broken up the lines to improve readability, and we have numbered the lines for reference.

2.2.2 What Can Be Reported?

What data can this test Agent report? The beginning of the Devices "container" contents is indicated by the XML tag `<Devices>` (figure 4, line 9) and the end of the container contents by `</Devices>` (line 85). This particular Agent represents only a single Device (the simulated LinuxCNC vertical mill controller), so the Devices node encloses only a single Device node, which starts at line 10 and ends at line 84.

The children of each Device (i.e., the entities contained "within" it) are its subassemblies or components; the MTConnectspec states that *every* Device must have at least a Power component, which at a minimum reports whether the device is powered on or off. You can see at lines 77–82 that there is indeed a Power component.

You can also see that there are four axes that can deliver measurements. One is a spindle (lines 23–32), and there are also linear X, Y and Z aces (lines 33–42, 43–52, 53–62 respectively). Notice that each Linear axis has a different name ("X", "Y" and "Z" respectively); this shows that a given component can have multiple subcomponents of the same type (Linear axis) as long as they all have different names.

2.2.3 DataItems: What Actual Data Will Be Delivered?

What specific data—in what units, etc.—will be reported for each component? Take a look again at lines 33–41, which describe the Linear X axis measurements available. Inside the Linear node, we find a list of DataItems. Each item in this list corresponds to one type of measurement that is available for this component (the Linear X axis).

Lines 35–37 and 37–39 show the two types of measurements that this axis can report. Both measurements are of type POSITION (contrast with the Spindle type axis, whose measurement type is SPINDLE_SPEED, as you can see in line 25). One measurement corresponds to a COMMANDED position and the other to an ACTUAL position (the subType attribute of each DataItem). Both measurements are delivered in units of MILLIMETER, and both are in the category SAMPLE. The category tells you, in effect, what "kind" of data this is: a numerical sample value, an event (such as an alarm or a power state such as ON vs. OFF), and so on. For example, you can see that the Power subsystem (lines 77–82) can deliver a "measurement" whose type is POWER_STATUS and whose category is EVENT (rather than SAMPLE). The reason for this is that SAMPLEs are used to denote actual numerical data sample values, whereas EVENTs are used to denote properties that can only take on one of a fixed number of values, such as ON or OFF for the POWER_STATUS.

What determines whether a component reports SAMPLES or EVENTS?The complete MTConnect specification spells out exactly what different categories, types, and subtypes are allowed to appear in the response to a `probe`. The MTConnect Specification document describes these in plain English, but the authoritative (and most precise) description is the XML Schema file available from the MTConnect Web site, `http://www.mtconnect.org`.

You've no doubt noticed that each component has other attributes we haven't mentioned—for example, fields such as *id* or *nativeUnits*. You can read about all of these in detail in the MTConnect specification; in general we'll introduce them only as needed to get you writing applications quickly. However, it is worth noticing some attributes describing the Device represented by this agent (see line 10), because they tell you something about the machine:

- `uuid` is an identifier that uniquely identifies this component. A common method for generating these is to reverse the Internet domain name of the organization (e.g. mfg.mycorp.com becomes com.mycorp.mfg) and then adding the component's serial number or other unique identifer (e.g. com.mycorp.mfg.2205178).

- `name` is a human-readable name describing the component, LinuxCNC in our example.
- `sampleRate` is the maximum rate at which this device samples, in samples per second. We'll return to this when we use the Sample command.

As you can see, `probe` doesn't actually return data samples values; it only returns information describing the *structure* of the device and component hierarchy. Next we'll use this information to determine what data to request.

2.2.4 Paths and XPath

The hierarchical arrangement you saw in the output of `probe` leads us to the very important concept of the *path* to a node in the hierarchy. You can think of a path as a description of how to traverse the data, starting from the root node (the node labeled *m:MTConnectDevices* in line 2 of figure 4; don't worry too much about the name) and ending up at the node or nodes with the measurements you are actually interested in.

For example, to "navigate" to the node describing the Z-axis Actual position in figure 4, you might describe it like this:

1. Begin at the root node, *m:MTConnectDevices* (line 2).
2. From there, look for the Devices child (line 9).
3. From there, look for the first Device child (line 10).
4. From there, find the Components associated with the Device (line 15).
5. From there, locate the Axes subcomponent (line 16).
6. From there, find the Components list that makes up the Axes (line 22).
7. Within that list, find the Linear component whose name is Z (line 53).
8. Finally, inside that component, you can see what data items it will return.

Using the same "directory" notation you've seen for navigating file systems, we could write down the *path* to the Z axis data as:

```
   /m:MTConnectDevices/Devices/Device/Components/¬
Axes/Linear[@name="Z"]/DataItem[@subType="ACTUAL"]
```

(where we've used the symbol ¬ to indicate a line break that implies no additional spaces, i.e. the above string should be all run together on one line). In the above expression, the square brackets indicate additional *restrictions* in following the path; for example, the excerpt `Axes/Linear[@name="Z"]` can be read as "From the Axes component, find the child called Linear *whose name is 'Z'.*" In the case of Devices and Device, we didn't need these extra qualifiers, since there is only one Devices node and it has only one Device node as a child.

The notation above is legal syntax for `XPath`, a standard for describing how to "navigate" an XML document (such as the Probe output) to find what you want. However, it's also somewhat cumbersome. We can make it much more readable and compact by taking advantage of the fact that XPath doesn't require us to navigate "step by step". In particular, above we used the notation `A/B` to indicate "from A, navigate to the child of A that is called B", but XPath also lets us use the notation `A//B` to indicate "from A, navigate to the *descendant* of A that is called B." As in real life, a descendant could be a child, a grandchild (child of a child), etc. So it should be easy for you to convince yourself that the following XPath expression will also take us to the Z-axis Actual measurement:

```
//Devices//Device//Axes//Linear[@name="Z"]
```

Not nearly as bad! Notice that `Devices//Device` is just as valid as `Devices/Device`: since Device is a child of Devices, by definition it's also a descendant of Devices. Note also that we can omit the root node: since the path starts with `//Devices`, it implicitly means "start at the root of the tree, and locate the first descendant called Devices." In fact, for our simple CNC controller, even the following path would be valid:

```
//Linear[@name="Z"]
```

This says: "From the root of the tree, find the first descendant that is a Linear node whose name is 'Z'." This works because we know that in our simple machine, only one possible measurement can match that criterion. On the other hand, if we just gave the path as `//Linear`, it would match all three of the Linear axes (named "X", "Y" and "Z" respectively).

Next we'll use this path notation to actually get data samples.

2.3 Sample and Current: Data Items and Events

An *event* is the act of capturing a data item; in addition to the raw data value, the event includes metainformation such as the sequence number of the event (showing the order in which the measurements were taken or alarms noted) and the time at which the event occurred (when a measurement was taken or an alarm noted, to whatever time resolution is supported by the device).

The reason for Events is that MTConnect tries to avoid assumptions about the relationship between the rate at which a particular controller produces data and the rate at which a particular client application wants to consume it. As a simple example, consider spindle load. A particular Agent (embedded in a controller, say) may be able to produce samples of this value every 10 milliseconds, and indeed a sophisticated monitoring client might want to collect such fine-grained samples. But what about a coarser-grained monitoring client that only collects a couple of "snapshots" per second of this data? How would the Agent "know" at what granularity it should be capturing samples?

Using Events eliminates this apparent problem. You can think of the Agent as "filling a bucket" with data samples (events) on its own, and different applications then "draw samples" from the bucket at different frequencies as needed. Periodically, as the bucket gets full, the Agent discards the oldest samples—whether anyone has seen them or not.

2.3.1 The Current Command: Show Me a Snapshot

The command `current` returns the *most recent* set of measurement samples from all reporting components. It takes an optional argument, but we will first explain its behavior when this optional argument is omitted.

You already saw that a command is transmitted to an Agent by constructing an HTTP URL consisting of the Agent's base address and the name of the command. So, point your Web browser at `http://agent.mtconnect.org/LinuxCNC/current`, and you should see output similar to that shown in figure 4 (the timestamps and some of the values may be different, and we made the formatting easier to read in the printed listing, but the structure should be the same).

Let's see how the output of `current` in figure 4 corresponds to that of `probe`. Recall that when we looked at the output of the `probe` command (figure 4), we looked at the Axes entity (line 16) which has a child Spindle (line 23) whose name is "S", and that Spindle is capable of reporting its `ACTUAL SPINDLE_SPEED` (line 25) in units of `REVOLUTION/MINUTE` (line 28). While you're there, notice the *id* of this DataItem is 10 (line 27).

Now look at line 36 of the output from the `current` command (listing 4), which begins a `ComponentStream` for the device identified by the path `/Device[@name="LinuxCNC"]//Axes[@name="Axes"]` and the specific component `S` within that path. The correspondence makes it clear that the measurements contained in this `ComponentStream` element correspond to that same spindle.

What are the values of those measurements? You can see a `Samples` element (line 39) containing a single sample (line 40), a `SpindleSpeed` measurement whose value is 3400.0. The measurement was taken at 11:18 and 16.372 seconds on May 23, 2008 (the `timestamp` attribute of the `SpindleSpeed` sample).

What are the units of the measurement? The fact that the SpindleSpeed measurement also has an *itemId* of 10, which corresponds to the *id* of 10 in line 27 of the `probe` output, is a further way of telling you that this SpindleSpeed sample is a measurement of the type described by the DataItem structure in that `probe`. That DataItem tells us that the units are revolutions per minute (figure 4, line 28). In other words, we looked at the *itemId* of the measurement value from `current`, and matched it up with the *id* of the DataItem from `probe` to determine, in effect, how to interpret the value.

There's another way (besides matching up the *itemId* and *id* fields) to do match these up as well. Notice the `path` attribute of the `Samples` element (figure 4, line 37). You can easily verify that the path given there, namely `/Device[@name='LinuxCNC']//Axes[@name='Axes']`, is an XPath expression. If we apply this path to the Probe output (figure 4), just as we did in section 2.2.4, it would return the entire

`Axes` node, which spans lines 11–42 in figure 4.

And as we'll see, the Client Libraries for MTConnect automate much of this matching-up and document-traversal for you.

2.3.2 Test Your Understanding...

In a similar way, you should be able to look at figures 4 and 4 and convince yourself of the following:

1. The Z-axis is a Linear axis that supplies Position measurements in units of millimeters; it is capable of providing both a commanded and actual position measurement. (Probe, lines 53–62)
2. The most recent actual value of the Z-axis is −0.1000000015mm, although the commanded value was −0.1mm. (Current, lines 28–33)
3. The Power subsystem (Probe lines 77–82) is only able to report Power status. Power status is an Event, rather than a Sample, because it is not a measured value but one of a finite number of possible states.
4. At the time this sample was taken, the current Power "measurement" shows that the power state is ON (Current lines 48–52).
5. The Controller subsystem (Probe lines 65–76) is able to report various items including Line, Controller Mode, Program and Execution. However, the current sample (Current lines 54–68) only reports values for Controller Mode (Automatic), Execution (Executing), and Program (a GCode file called `spiral.ngc`; it doesn't report a value for Line. This could be because this particular controller doesn't report that value. It illustrates the important concept that while an entity (e.g. Controller) may be *logically capable* of reporting on a particular type of measurement, that doesn't necessarily guarantee that the measurement *will always* be reported given some specific machine.

As you can see from the above discussion, the `current` command gives you the *data* associated with one or more measurements, and each data item can be one of a number of different types of data—a Sample (which itself can be a Position or a SpindleSpeed in this example) or an Event (such as the Execution, Controller Mode and Program "measurements" associated with the Controller in our example). The complete specification of different types of events and samples is codified in the MTConnect XML Schema.

You also saw that the `probe` command gives important `metadata` ("data about the data") associated with measurements. For example, it is the output of the Probe command that tells us that `SpindleSpeed` is reported in RPM whereas the `Z` axis position is reported in millimeters.

2.3.3 The Current Command with a Path Argument

In the previous example, we didn't pass any arguments to the `current` command, so by default it reported current values for all measurements reportable by this Agent. However, you can also call `current` with an argument that restricts its attention only to certain measurements. To do this, we simply construct an XPath expression (similar to what we saw in section 2.2.4) that matches only the parts of the "tree" we care about.

For example, suppose we care only about the feedrates of the various axes, and not the controller program state. If you pass an argument called *path* to the `current` command, the samples returned will be restricted only to those matching the given path. Referring again to the output of Probe (figure 4), recall that the path `//Axes` will match only the Axes element of the schema.

To issue a command with arguments to an Agent, we build up the command URL as usual, and then append a question mark after the command name, followed by the argument name and value:

```
http://agent.mtconnect.org/LinuxCNC/current?path=//Axes
```

Type this into a Web browser and you'll see that the result you get is much more concise than what you got when `current` was issued with no arguments. Indeed, what you get is that subset of the full snapshot that matches the path `//Axes`.

Similarly, the path `//Axes//Spindle` will match only the Spindle axis and not the other (linear) axes; you can read this path as: "Find all the Axes components, wherever they occur; then, find any Spindle components that are descendants of those Axes components."

What if we wanted only the linear Z axis? You might think to try `//Axes//Linear`, but all three (X, Y, Z) axes match that path. In this case, we can constrain the match using the *attributes* of an element, as we showed in the first example in section 2.2.4 and write the path expression `//Axes//Linear[@name="Z"]` to get what we want. You can read this as: "Find all the Axes components, wherever they occur; then, inside each one of them, find any Linear components *whose* name *attribute has the value 'Z'.*" Recall that we don't need to use this trick to select the Spindle, because there is only one Spindle; but it would be perfectly fine to use the path `//Axes//Spindle[@name="S"]` in that query, leading to the URL

```
   http://agent.mtconnect.org/LinuxCNC/¬
current?path=//Axes//Spindle[@name="S"]
```

What if the path we give doesn't match anything? For example, what if we said

`//Axes//Spindle[@name="NoSuchSpindle"]` or
`//Axes//NeitherLinearNorSpindle`

If you try it, you'll see that you still get back a well-formed XML response document—but it will contain no measurements or data samples. In particular, the document will still have a Streams element that contains a DeviceStreams list, but that list will be empty. This illustrates that it's not an error for an XPath not to match anything.

Try It You're strongly encouraged to try these behaviors yourself by typing these URL's right into your browser. There's no better way to get familiar with the XPath syntax than doing it yourself.

XPath syntax is extremely rich, and if you know how to use it, you can construct quite sophisticated queries. We will stick to simple examples, but one interesting syntax construct is the use of the vertical bar or "pipe symbol", `|`, to mean "one or the other." You construct each path expression completely separately, then join them with `|`. Thus in the above example, if we wanted the values of the X and Y axes only, we could use:

```
    http://agent.mtconnect.org/LinuxCNC/¬
current?path=//Axes//Linear[@name="X"]|//Axes//Linear[@name="Y"]
```

2.3.4 The Sample Command

The command `sample` is a generalization of `current`. Rather than returning only the most recent measurement sample from one or more components, it returns a set of samples.

It takes up to four arguments. The `start` argument tells the Agent what the first (earliest) measurement to be returned should be. Notice in the example output from the Current command (figure 4) that each sample item has a sequence number associated with it (look for the *sequence* attribute associated with each *m:Position* measurement, for example at lines 15, 19, 29, etc.) Every time an Agent records a new event from a controller, it increments the sequence number associated with that controller, so that every single measurement taken gets its own sequence number. The sequence numbers are a simple way to number the events so you can tell which ones you've seen before.

Thus, the `start` argument to the `sample` command is the desired starting sequence number of data items. The Client must supply this since there may be multiple Clients communicating with a single Agent, and the Agent doesn't necessarily keep track of which sequence numbers have been seen by which clients. However, looking again at the output of the Current command, you'll see that the Header element has the attribute `nextSequence="52056912"` (line 8). This is the Agent's way of telling the client what sequence number that client should request next, i.e. it's the lowest sequence number that client hasn't seen yet. As you'll see in the next chapter, the Client Utility Library keeps track of this for you automatically.

Why don't I have to keep track of the sequence number for the Current command?By definition, the Current command returns the "most recent" sample value for one or more measurements—so you don't need to supply a sequence number.

The optional `count` argument specifies a maximum number of samples to return, and defaults to 100 if no value is given. Fewer samples may be returned if not enough are available (for example, if `count` is 100, but fewer than 100 samples have been reported since the desired starting sequence number). In fact, if there are *zero* samples available matching the given path and starting sequence number (perhaps the machine is offline or idle), zero events may be returned.

The last argument, `freq`, is for scenarios in which a Client wishes to continuously receive a streaming data feed of specific data items. Use of this argument is optional and we won't cover it further in this overview, since it requires some programming techniques not covered in this tutorial.

As before, we encode these arguments into the URL by appending a question mark, and we separate multiple arguments with ampersands (&). For example, to issue the Sample command with arguments *count*= 100, *sequence*= 23456 and *path*="//Axes//Linear", we would construct the URL

```
    http://agent.mtconnect.org/LinuxCNC/sample?count=¬
  100&sequence=23456&path="//Axes//Linear"
```

In the unlikely event you need to embed an ampersand in the URL as part of an argument value, you can use the escape sequence `&amp;` (you need both the & and the trailing semicolon) to do so.)

2.3.5 Recap: Things to Remember

"Error" responses are rare: Keep in mind that as long as a command to the Agent is well-formed—i.e., the Agent URL is correct, values have been supplied for required arguments, and any values for optional arguments are legal—then the response will *always* be a valid XML document conforming to the MTConnect XML Schema. Of course, in some cases, the document may not have any "interesting" content: as we saw above, if you specify an XPath that doesn't correspond to any component that can deliver measurements, or if you request samples but there are no new samples to report, you may get back a document with zero samples. But it is still a legal response and must still conform to the MTConnect XML Schema. Of course, if the URL encoding the command to the Agent is ill-formed, you may get an HTTP error.

Is Probe always necessary? A Client can certainly issue `sample` or `current` commands without ever having issued a `probe` first. For example, a particular Client may be designed to work only with a specific device, and the device's schema may be "hardwired" into the Client code. For example, we used knowledge from the Probe command above to determine in what units a particular measurement was reported. But if the engineer writing the application "knows" what units and measurements are reported by the machine, and he's writing an application that will only gather data from that specific machine, he may choose to "hardcode" this knowledge into the application itself rather than going through the extra step of doing a Probe. Of course, if the controller behavior changes for some reason, this would be reflected in the output of the Probe command; so it's more portable and somewhat safer to use Probe rather than hardwiring this knowledge into the application.

In contrast, a more generic Client, such as a monitoring/analysis tool that can be used with a variety of devices, might first do a `probe` to find out what devices are available to be monitored, possibly presenting the human operator with a menu or selection list to narrow down what she wishes to see, and would also use the Probe output to get information about the units in which each measurement is reported, in order to perform unit conversions.

As we'll see, the Client Libraries described in the next section *do* expect you to do a Probe first, because they will then remember for you such information as measurement units and information about the machine itself.

3 Client Library, Utilities and Extensions (CLUE) for Microsoft .Net

In the previous section we described how to type URL's into a Web browser to issue commands to the Agent directly. This section describes how to write client applications for MTConnect that will do this programmatically, using the MTConnect Client Library, Utilities and Extensions (CLUE) for Microsoft .NET Framework 2.0 or later.[4]

As you already know, *you don't need any SDK to write MTConnect applications*, as long as your programming language or development system includes libraries for using XML, XPath and HTTP. The CLUE libraries and the API's they provide, even though not part of the official MTConnect speification, may help streamline your code. CLUE uses .NET's underlying HTTP and XML/XPath libraries to automate and streamline some of the procedures you learned about in the previous section.

In fact, once you've made it all the way to the end of this tutorial, you may find it helpful to peruse the source code of CLUE to understand just how the CLUE API calls are converted to HTTP, XML and XPath calls.

A unique feature of the Microsoft .NET framework is that libraries compiled for it can be called from applications in any language supported by .NET. Currently these include C# (C Sharp), J# (similar to Java), Visual Basic.NET, and many, many others. Our examples will use C#, but translation to other .NET-supported languages is straightforward.

3.1 Overview of CLUE for .NET

We will use C# for most examples, occasionally showing the equivalent VB.Net code. There are three object abstractions provided by the CUL (and therefore three classes provided) in the namespace MTConnect:

1. MTConnect.MTCClient: represents a connection to an MTConnect Agent. Remembers the Agent's host and port number and the "last sequence number" sample seen from the Agent (section 2.3.4). Provides methods for sending commands such as Probe, Sample, etc. An application that communicates with multiple Agents would create multiple instances of this object.

2. MTCResponse: a class that encapsulates a successful response from the Agent. Given a response message, this class provides methods that let you easily extract specific measurements using just the names of the components (e.g. "Linear, X" to get an axis position measurement), or if you wish, get the raw XML.

3. MTCEvent: given a measurement extracted from an MTCResponse, provides functions to perform unit conversions by cross-referencing the Probe schema. Converts attribute values to native classes

[4]Future versions of CLUE may include support for other development environments and languages such as Java, Python and Ruby.

(e.g. converts the `timestamp` attribute to a .Net `DateTime` object on which time calculations can be performed).

A typical application using the .NET CLUE Libraries has the following structure:

1. Create an instance of an `MTCAgent` object to represent the connection to a particular Agent.
2. Issue a Probe command, which returns an `MTCProbeResponse` object. Methods on this object tell you about the device(s) connected to the agent, including the devices' manufacturers, serial numbers, and any other MTConnect-provided information. It also provides CLUE the information that will be necessary later to associate the correct measurement units with returned samples.
3. While measurements are still needed:
   (a) Optionally create an XPath expression corresponding to the desired measurements (or use no expression, to get all reportable measurements);
   (b) Call the `Sample` or `Current` method on the Agent object—both of these return an `MTCSampleResponse` object;
   (c) Call `FindByComponentAndName` on the `MTCSampleResponse` object to extract desired measurements, which are returned as `MTCEvent` objects that include the measurement value, units, lane, and other information.

3.2 Example 1: Hello World for MTConnect

This first example repeatedly fetches measurement values for a particular component type and name, e.g. a `Linear` type axis named `Z`. To keep it simple, in this version we will use the Current command with no arguments (section 2.3.1).

The reason you must specify both the Component type and Name for a desired measurement is that the MTConnect specification guarantees that any given *combination* of Component and Name must be unique within a device, but either Component or Name by itself might not be. So there might be more than one Component of type Linear (e.g., there could be X, Y and Z linear axes), and more than one Component whose name is `Z` (e.g., a machine-specific property named Z), but there can be at most one component whose type is Linear *and* whose name is `Z`. Later we will see another way to constrain what measurement is desired, by passing a Path argument to `Current`.

Here is some example output from that program, where each new set of measurements is delivered after each press of the Return key:

3.2.1 Things To Notice

CLUE tries to automate some common "housekeeping" tasks even in a simple program like this example.

Next Sequence. Along with each set of measurements, the program displays the “next sequence”. Recall the “bucket” model whereby the Agent “fills the bucket” with measurements and a client application removes them. The “next sequence” number is the number of the next sample the client has not yet seen. Later when we use the Sample command, we’ll show how to use this to make sure we don’t miss any samples. The `MTCAgent` object automatically parses this value from the output of every Sample or Current command (e.g., see line 3 of figure 4) and remembers it for you.

Units. The units of each measurement, if known, are reported. The first time you do a Probe, the `MTCAgent` object automatically remembers the unit types associated with each measurement (e.g., see lines 13, 18, 25, etc. in figure 4)

3.2.2 Using Fewer Constraints

Change lines 23–24 of Example 4 as follows:

```
23   String name = String.Empty;
24   String comp = "Linear";
```

Then rebuild and re-run the program. Now, each time you press Return you should see a total of 6 measurements per sample: a commanded and an actual position for each of the X, Y and Z Linear axes.

What happened? Because we did not specify a particular component name (we used `String.Empty`) in line 23, we got samples for *all* components whose component type is `Linear`. In fact, this would correspond directly to the XPath expression `//ComponentStream[@component="Linear"]`, and you can browse the source code of CLUE (file `Client.cs`) to verify that this is exactly what’s being done.

Why use String.Empty?`String.empty` is the “natural” way of declaring an empty string in C#; you could also pass the empty string literal `""`, but `String.empty` is more efficient since it doesn’t create a new object. Also, CLUE also allows you to pass the value `null` in this case, though in general `null` and the empty string are *not* interchangeable.

If you set `comp` to the empty string but pass a value (such as `"X"`) for `name`, you’ll get measurement samples for all components regardless of their type whose name matches `"X"`. (For example, if the same Agent is reporting on multiple distinct sets of linear axes, more than one of which has an axis called `X`, this combination of arguments would give back samples for all of the X axes.)

3.2.3 Other Ways To Use `FindByComponentAndName`

An alternative version of `FindByComponentAndName` allows you to specify additional criteria to match on certain attributes. For example, if you know you only want to look at actual positions or velocities and not commanded ones, i.e. position or velocity measurements whose *subType* attribute matches `ACTUAL`, you could pass the XPath expression `//[@subType="ACTUAL"]` to `FindByComponentAndName`. So, for example, the following call would extract only commanded position measurements for *any* Linear axis:

```
        evts = resp.FindByComponentAndName("Linear", ¬
     String.empty, "//[@subType='COMMANDED']");
```

Note the use of single quotes inside the double quotes to delimit the constraint string. We could also have used the standard C# syntax for including quotes in a string: `"//@subType=\"COMMANDED\"`.

Finally, if you want to just specify an XPath expression directly to capture all of the constraints, you can use `FindByXPath` to do just that. You can pass it a string or a precompiled XPathExpression object (this is an advanced feature of the .NET support for XPath, so we'll just illustrate using a string.) The following call to `FindByXpath` also returns only commanded position measurements for Linear axes (compare it to the example above):

```
        evts = resp.FindByXPath("//Linear//[@subType='COMMANDED']");
```

3.2.4 Discussion

In our simple example, we didn't pass any arguments to Current. As you recall from section 2.3.1, this causes Current to return a "complete snapshot" of all reportable measurements, even though our simple app is then only selecting a small subset of them to examine.

Because our example machine is pretty simple, this is probably OK. But if we were dealing with a more complex production machine that can report hundreds of measurements at a time, it's inefficient to ask for everything just to look at one or two items. In that case, we'd be better off passing a Path argument to Current, to narrow down which measurements will be returned. An overloaded version of the `Current` method of `MTCAgent` allows us to pass an XPath expression directly to constrain "up front" which measurements are reported. To illustrate this technique, example 4 shows an alternative version of our simple client in which we only request the Power subsystem's measurements when we do the Current command; then when we call `FindByComponentAndName` we just pass empty strings for both `comp` and `name`, since we know there's just one set of measurements in the reply anyway.

3.3 Example 2: Working With Samples

In this section we show example code for working with larger volumes of samples at once. Rather than just using the Current command to get subsequent "snapshots" of the machine state, we'll use the Sample command to periodically download a batch of samples and export it for use by other applications.

In this section we show example code to capture some data according to user specifications and then write that data into a CSV (Comma-Separated Values) file. Many popular tools, including databases such as Microsoft Access or MySQL and spreadsheets such as Microsoft Excel, can import CSV files directly.

3.4 Example 4: Alarm Monitoring

TBD

4 Code Listings

```
\begin{verbatim}

 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <m:MTConnectDevices
 3    xsi:schemaLocation="urn:mtconnect.com:MTConnectDevices:0.9
 4                        http://www.mtconnect.org/schemas/MTConnectDevices.xsd"
 5    xmlns:m="urn:mtconnect.com:MTConnectDevices:0.9"
 6    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 7   <Header sender="localhost" creationTime="2008-05-23T11:08:55-07:00"
 8           bufferSize="100000" version="0.9" instanceId="1209943337"/>
 9   <Devices>
10     <Device sampleRate="100.0" name="LinuxCNC" iso841Class="6" uuid="linux-01">
11       <Description serialNumber="01" manufacturer="NIST"/>
12       <DataItems>
13         <DataItem type="ALARM" name="alarm" category="EVENT" id="2"/>
14       </DataItems>
15       <Components>
16         <Axes name="Axes">
17           <DataItems>
18             <DataItem type="PATH_FEEDRATE" subType="OVERRIDE"
19                       name="path_feedrate" units="PERCENT"
20                       category="SAMPLE" id="3" nativeUnits="PERCENT"/>
21           </DataItems>
22           <Components>
23             <Spindle name="S">
24               <DataItems>
25                 <DataItem type="SPINDLE_SPEED" subType="ACTUAL"
26                           name="Sspeed" units="REVOLUTION/MINUTE"
27                           category="SAMPLE" id="10"
28                           nativeUnits="REVOLUTION/MINUTE">
29                   <Source>spindle_speed</Source>
30                 </DataItem>
31               </DataItems>
32             </Spindle>
33             <Linear name="X">
34               <DataItems>
35                 <DataItem type="POSITION" subType="ACTUAL" name="Xact"
36                           units="MILLIMETER" category="SAMPLE" id="4"
37                           nativeUnits="MILLIMETER"/>
38                 <DataItem type="POSITION"
39                           subType="COMMANDED" name="Xcom" units="MILLIMETER"
40                           category="SAMPLE" id="5" nativeUnits="MILLIMETER"/>
41               </DataItems>
42             </Linear>
43             <Linear name="Y">
44               <DataItems>
45                 <DataItem type="POSITION" subType="ACTUAL" name="Yact"
46                           units="MILLIMETER" category="SAMPLE" id="6"
47                           nativeUnits="MILLIMETER"/>
48                 <DataItem type="POSITION"
49                           subType="COMMANDED" name="Ycom" units="MILLIMETER"
50                           category="SAMPLE" id="7" nativeUnits="MILLIMETER"/>
51               </DataItems>
52             </Linear>
53             <Linear name="Z">
54               <DataItems>
55                 <DataItem type="POSITION" subType="ACTUAL" name="Zact"
56                           units="MILLIMETER" category="SAMPLE" id="8"
57                           nativeUnits="MILLIMETER"/>
58                 <DataItem type="POSITION"
59                           subType="COMMANDED" name="Zcom" units="MILLIMETER"
60                           category="SAMPLE" id="9" nativeUnits="MILLIMETER"/>
61               </DataItems>
62             </Linear>
63           </Components>
64         </Axes>
65         <Controller name="Controller">
66           <DataItems>
67             <DataItem type="LINE" subType="ACTUAL" name="line"
68                       category="EVENT" id="11"/>
69             <DataItem type="CONTROLLER_MODE"
70                       name="mode" category="EVENT" id="12"/>
71             <DataItem type="PROGRAM" name="program" category="EVENT"
72                       id="13"/>
73             <DataItem type="EXECUTION" name="execution"
74                       category="EVENT" id="14"/>
75           </DataItems>
76         </Controller>
77         <Power name="power">
78           <DataItems>
79             <DataItem type="POWER_STATUS" name="power" category="EVENT"
80                       id="1"/>
81           </DataItems>
82         </Power>
83       </Components>
84     </Device>
85   </Devices>
86 </m:MTConnectDevices>

\end{verbatim}
```

Listing 1: The raw XML output of issuing the Probe command to the Test Agent by fetching from the

```
 1 <?xml version="1.0" encoding="UTF-8"?>
 2 <m:MTConnectStreams xmlns:m="urn:mtconnect.com:MTConnectStreams:0.9"
 3                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 4                  xsi:schemaLocation="urn:mtconnect.com:MTConnectStreams:0.9
 5                  http://www.mtconnect.org/schemas/MTConnectStreams.xsd">
 6   <Header sender="localhost" creationTime="2008-05-23T11:18:36-07:00"
 7           bufferSize="100000" version="0.9" instanceId="1209943337"
 8           nextSequence="52056912"/>
 9   <Streams>
10     <DeviceStream name="LinuxCNC" uuid="linux-01">
11       <ComponentStream
12          path="/Device[@name=’LinuxCNC’]//Axes[@name=’Axes’]" name="Y"
13          component="Linear">
14         <Samples>
15           <m:Position itemId="6" subType="ACTUAL" sequence="52056909"
16                       timestamp="2008-05-23T11:18:36.418">0.9901518226
17           </m:Position>
18           <m:Position itemId="7" subType="COMMANDED"
19                       sequence="52056910"
20                       timestamp="2008-05-23T11:18:36.418">0.9949269086
21           </m:Position>
22         </Samples>
23       </ComponentStream>
24       <ComponentStream
25          path="/Device[@name=’LinuxCNC’]//Axes[@name=’Axes’]" name="Z"
26          component="Linear">
27         <Samples>
28           <m:Position itemId="9" subType="COMMANDED"
29                       sequence="52056362"
30                       timestamp="2008-05-23T11:18:20.986">-0.1</m:Position>
31           <m:Position itemId="8" subType="ACTUAL" sequence="52056361"
32                       timestamp="2008-05-23T11:18:20.986">-0.1000000015
33           </m:Position>
34         </Samples>
35       </ComponentStream>
36       <ComponentStream
37          path="/Device[@name=’LinuxCNC’]//Axes[@name=’Axes’]" name="S"
38          component="Spindle">
39         <Samples>
40           <m:SpindleSpeed itemId="10" subType="ACTUAL"
41                           sequence="52056246"
42                           timestamp="2008-05-23T11:18:16.372">3400.0
43           </m:SpindleSpeed>
44         </Samples>
45       </ComponentStream>
46       <ComponentStream path="/Device[@name=’LinuxCNC’]" name="power"
47                        component="Power">
48         <Events>
49           <m:PowerStatus itemId="1" sequence="4"
50                          timestamp="2008-05-04T16:22:18.371">ON
51           </m:PowerStatus>
52         </Events>
53       </ComponentStream>
54       <ComponentStream path="/Device[@name=’LinuxCNC’]"
55                        name="Controller" component="Controller">
56         <Events>
57           <m:Execution itemId="14" sequence="52056316"
58                        timestamp="2008-05-23T11:18:18.402">EXECUTING
59           </m:Execution>
60           <m:ControllerMode itemId="12" sequence="3"
61                             timestamp="2008-05-04T16:22:18.371">AUTOMATIC
62           </m:ControllerMode>
63           <m:Program itemId="13" sequence="6"
64                      timestamp="2008-05-04T16:22:18.371">
65             /home/mtconnect/emc2/nc_files/examples/spiral.ngc
66           </m:Program>
67         </Events>
68       </ComponentStream>
69       <ComponentStream
70          path="/Device[@name=’LinuxCNC’]//Axes[@name=’Axes’]" name="X"
71          component="Linear">
72         <Samples>
73           <m:Position itemId="4" subType="ACTUAL" sequence="52056911"
74                       timestamp="2008-05-23T11:18:36.418">-1.6620382071
75           </m:Position>
76           <m:Position itemId="5" subType="COMMANDED"
77                       sequence="52056908"
78                       timestamp="2008-05-23T11:18:36.418">-1.6591126305
79           </m:Position>
80         </Samples>
81       </ComponentStream>
82     </DeviceStream>
83   </Streams>
84 </m:MTConnectStreams>
```

Listing 2: The raw XML output of issuing the Current command to the Test Agent without any arguments. You can see this by pointing a Web browser at `http://agent.mtconnect.org/LinuxCNC/probe`. The lines are numbered for reference only; the line numbers aren’t part of the actual XML output.

```
using System;
using MTConnect;


namespace SimpleBrowser
{
    class Program
    {
        public static void Main(string[] args)
        {
            String agentUrl = "http://agent.mtconnect.org/"; // the Test Agent
            MTCSampleResponse resp;   // to hold responses from the Agent
            String input;

            // create Agent object used to communicate with the Agent
            MTCAgent ag = new MTCAgent(agentUrl);

            // first do a Probe and print some interesting info about the device
            MTCProbeResponse prb = ag.Probe();

            Console.WriteLine("MTConnect devices attached to this Agent:");
            foreach (MTCDevice d in prb.devices)
            {
                Console.WriteLine("{0} {1} s/n#{2}, UUID={3}, sample rate={4:F}",
                    d.manufacturer, d.name, d.serialNumber,
                    d.uuid, d.sampleRate);
            }
            // The component name and type we are interested in:
            String name = "X";
            String comp = "Linear";
            // Continuously get next sample until user exits.
            Console.WriteLine("Hit return for next sample or x to exit");
            ag.nextSequence = 3;
            while (true)
            {
                resp = ag.Current();    // get a set of samples
                Console.WriteLine("Getting next sample {0}", ag.nextSequence);
                MTCEvent[] e = resp.FindByComponentAndName(comp, name);
                foreach (MTCEvent ev in e)
                {
                    Console.WriteLine("{0}({1}) {2:D8} {3:-9}{4:9} {5:hh:mm:ss.ff} {6:9} {7}",
                        ev.componentType,
                        ev.componentName,
                        ev.sequence,
                        ev.valueType.ToString(),
                        (ev.valueSubtype == MTCEvent.ValueSubtype.UNKNOWN ? "" :
                         "[" + ev.valueSubtype.ToString() + "]"),
                        ev.timestamp.ToLocalTime(),
                        ev.value,
                        ev.units.ToString());
                    if (ev.hasParseErrors)
                        Console.WriteLine("  (Warning: " + ev.parseErrors);
                }
                input = Console.ReadLine();
                if (input.StartsWith("x")) { break; }
            }
        }
    }
}

```

Listing 3: Simple C# program that allows user to specify a component type and name, and repeatedly calls the Current command to get a new snapshot of samples and then extracts the measurement value(s) from the desired component out of each set of samples.

```
Sample output to be inserted here
```

Listing 4: Example output from the simple program in figure 4.

```
using System;
using System.Collections.Generic;
using System.Text;
using MTConnect;
using System.Xml;
using System.Xml.XPath;


namespace SimpleReporter
{
    class Program
    {
        ///  Coming soon.
    }
}

```

Listing 5: Given user-specified sample criteria, collect a number of samples of and export them to a Comma-Separated Values (CSV) file, for possible import into a database or spreadsheet.

```
namespace MTConnect
{
    public class MTCAgent
    {
        public enum debugLevel { debugNone, debugErrors, debugWarnings,
               debugInfo };
        public debugLevel debugging;
        public String instanceId;  // instanceID of this Agent
        public String version;  // MTConnect protocol version of the response
        public Int32 nextSequence;
               // next sequence number, updated on each Sample or Current
        public XPathNavigator cachedProbe;
               // navigator on root of first Probe() response
        // constructor: takes a base URL at which to contact the MTCAgent
        public MTCAgent(String url);
        // or use this method to discover the Agent(s)
        public MTCAgent[] FindByLDAP();   // using default LDAP server
        public MTCAgent[] FindByLDAP(String uri); // using specific LDAP svr
        public void SquashExceptions(bool squashExceptions);
        // Probe, Sample and Current
        public MTCProbeResponse Probe();
        public MTCSampleResponse Sample(String path, Int32 maxSamples,
                                  Int32 start);
        public MTCSampleResponse Current();
        public MTCSampleResponse Current(String path);

    }
    public struct MTCDevice
    {
        public String name;
        public double sampleRate;
        public String uuid;
        public String iso841class;
        public String serialNumber;
        public String manufacturer;
    };
    ///  You shouldn't normally use this base class (MTCResponse) --
    ///    use the subclasses MTCProbeResponse and MTCSampleResponse
    public class MTCResponse
    {
        public XPathNavigator xmlNav;
        public XPathDocument responseDoc;
        public MTCAgent agent;  // agent from which we got this response
        public MTCResponse(String loadPath, MTCAgent agt);
        public override String ToString();
    }
    /// MTCSampleResponse encapsulates the functionality for getting
    /// and parsing the XML document
    /// returned by a Current or Sample request to an Agent.
    /// You normally shouldn't have to call the constructor directly---
    /// instead get an instance of this object by calling Current or Sample.
    public class MTCSampleResponse : MTCResponse
    {
        public MTCSampleResponse(String loadPath, MTCAgent agt);
        public MTCEvent[] FindByComponentAndName(String component,String name);
        public MTCEvent[] FindByComponentAndName(String component,
                          String name, String xPathPredicate);
    }
    /// MTCProbeResponse encapsulates the functinality for interpreting
    /// the response to a Probe request.
    /// You normally shouldn't have to call the constructor directly---
    /// instead get an instance of this object by calling Probe.
    public class MTCProbeResponse : MTCResponse
    {
        public MTCDevice[] devices;
               // the devices represented in the probe data
        public MTCProbeResponse(String loadPath, MTCAgent agt);
    }
}
```

Listing 6: The MTConnect namespace, classes and methods provided by CLUE 1.0 for Microsoft .NET Framework. The class descriptions are given using C# syntax but translation to other object-oriented .NET languages is straightforward.

```
   1 g20 g64
   2 s3400 m3
   3 g0z1
   4 g0x0y0z1
   5 g0 x1.724638 y-1.012731
   6 g1z-.1 f24
   7 g1 x1.724638 y-1.012731
   8 x1.613302 y-1.178668
   9 x1.486083 y-1.332506
  10 x1.344282 y-1.472733
 ...[more x,y positions to define a spiral]...
1002 x0.008776 y0.004794
1003 x0.007368 y0.003115
1004 x0.005732 y0.001773
1005 x0.003920 y0.000795
1006 x0.001990 y0.000200
1007 g0z1
1008 m2
```

Listing 7: An elided version of the GCode program continuously running on the MTConnect Test Agent (line numbers added). It repeatedly cuts a spiral pattern.