MTConnect® Standard
Part 5.0 – Interfaces

Version 1.4.0

Prepared for: MTConnect Institute

Prepared on: March 31, 2018

MTConnect® Specification and Materials

Table of Content

Table of Figures

1 Purpose of This Document

This document, *Part 5.0 – Interfaces* of the MTConnect® Standard, defines a structured data model used to organize information required to coordinate inter-operations between pieces of equipment.

This data model is based on an *Interaction Model* that defines the exchange of information between pieces of equipment and is organized in the MTConnect Standard as the XML element `Interfaces`.

Interfaces is modeled as an extension to the `MTConnectDevices` and `MTConnectStreams` XML documents. `Interfaces` leverages similar rules and terminology as those used to describe a component in the `MTConnectDevices` XML document. `Interfaces` also uses similar methods for reporting data to those used in the `MTConnectStreams` XML document.

As defined in *Part 2.0 – Devices Information Model,* `Interfaces` is modeled as a *Top Level* component in the `MTConnectDevices` document (see *Figure 3* below). Each individual `Interface` XML element is modeled as a *Lower Level* component of `Interfaces`. The data associated with each *Interface* is modeled within each *Lower Level* component.

Note: See *Part 2.0 – Device Information Model* and *Part 3.0 - Streams Information Model* of the MTConnect Standard for information on how `Interfaces` is structured in the XML documents which are returned from an *MTConnect Agent* in response to a `Probe`, `Sample`, or `Current` request.

2 Terminology and Conventions

Refer to *Section 5* of *Part 1.0 - Overview and Functionality* for a dictionary of terms, reserved language, and document conventions used in the MTConnect® Standard.

3 Interfaces Overview

In many manufacturing processes, multiple pieces of equipment must work together to perform a task. The traditional method for coordinating the activities between individual pieces of equipment is to connect them together using a series of signal wires to communicate equipment states and demands for action. These interactions are usually accomplished by using simple binary ON/OFF signals.

In the MTConnect® Standard, *Interfaces* provides a means to replace this traditional method for interconnecting pieces of equipment with a structured *Interaction Model* that provides a rich set of information used to coordinate the actions between pieces of equipment. Implementers may utilize the information provided by this data model to (1) realize the interaction between pieces of equipment and (2) to extend the functionality of the equipment to improve the overall performance of the manufacturing process.

The *Interaction Model* used to implement *Interfaces* provides a lightweight and efficient protocol, simplifies failure recovery scenarios, and defines a structure for implementing a Plug-And-Play relationship between pieces of equipment. By standardizing the information exchange using this higher level semantic information model, an implementer may more readily replace a piece of equipment in a manufacturing system with any other piece of equipment capable of providing similar *Interaction Model* functions.

Two primary functions are required to implement the *Interaction Model* for *Interfaces* and manage the flow of information between pieces of equipment. Each piece of equipment needs to have:

- An *MTConnect Agent* which provides:
  - The data required to implement the *Interaction Model*.
  - Any other data from a piece of equipment needed to implement the *Interface* – operating states of the equipment, position information, execution modes, process information, etc.
- A client software application that enables the piece of equipment to acquire and interpret information from another piece of equipment.

3.1 *Interfaces* Architecture

MTConnect Standard is based on a communications method that provides no direct way for one piece of equipment to change the state of, or cause an action to occur by, another piece of equipment. The *Interaction Model* used to implement *Interfaces* is based on a *Publish/Subscribe* type of communications as described in *Part 1 – Overview and Functionality* and utilizes a *Request* and *Response* information exchange mechanism. For *Interfaces*, pieces of equipment must perform both the publish (*MTConnect Agent*) and subscribe (client) functions.

Note: The current definition of *Interfaces* addresses the interaction between two pieces of equipment. Future releases of the MTConnect Standard may address the interaction between multiple (more than two) pieces of equipment.

The diagram below provides a high-level overview of a typical system architecture used to implement *Interfaces*.

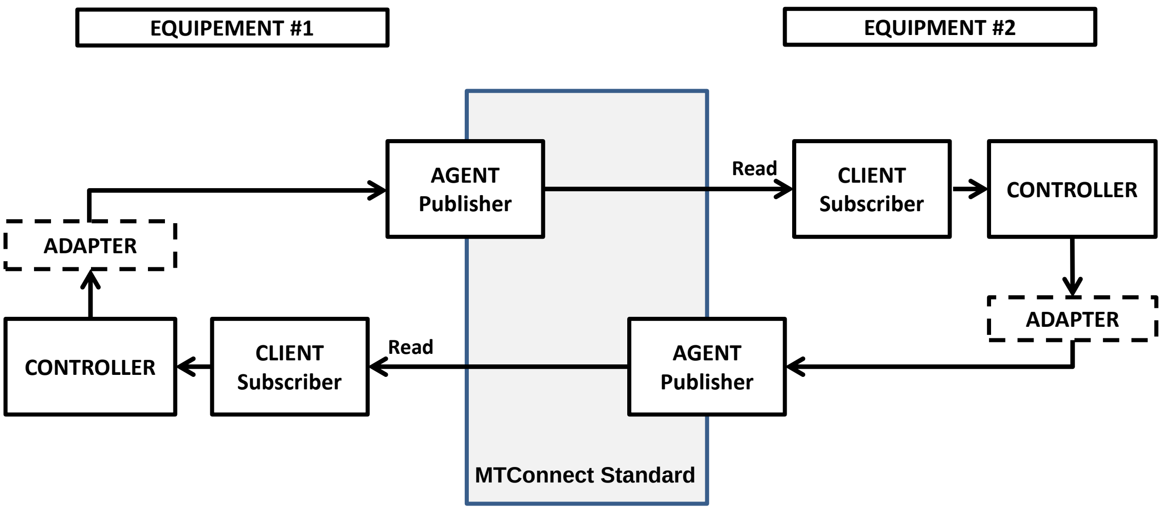

Figure 1: Data Flow Architecture for *Interfaces*

> Note: The data flow architecture illustrated in *Figure 1* above was historically referred to in the MTConnect Standard as a read-read concept.

In the implementation of the *Interaction Model* for *Interfaces*, two pieces of equipment can exchange information in the following manner. One piece of equipment indicates a *Request* for service by publishing a type of *Request* using a data item provided through an *MTConnect Agent* as defined in *Section 4* below. The client associated with a second piece of equipment, which is subscribing to data from the first machine, detects and interprets that *Request*. If the second machine chooses to take an action to fulfill this *Request*, it can indicate its acceptance by publishing a *Response* using a data item provided through its *MTConnect Agent*. The client on the first piece of equipment will continue to monitor information from the second piece of equipment until it detects an indication that the *Response* to the *Request* has been completed or has failed.

An example of this type of interaction between pieces of equipment can be represented by a machine tool that wants material to be loaded by a robot. In this example, the machine tool is the *Requester* and the robot is the *Responder*. On the other hand, if the robot wants the machine tool to open a door, the robot becomes the *Requester* and the machine tool the *Responder*.

3.2 *Request* and *Response* Information Exchange

The concept of a *Request* and *Response* information exchange is not unique to MTConnect *Interfaces*. This style of communication is used in many different types of environments and technologies.

An early version of a *Request* and *Response* information exchange was used by early sailors. When it was necessary to communicate between two ships before radio communications were available, or when secrecy was required, a sailor on each ship could communicate with the other using flags as a signaling device to request information or actions. The responding ship could acknowledge those requests for action and identify when the requested actions were completed.

The same basic *Request* and *Response* concept is implemented by MTConnect *Interfaces* using the `EVENT` data items defined in *Section 4*.

The `DataItem` elements defined by the *Interaction Model* each have a `Request` and `Response` subtype. These subtypes identify if the data item represents a *Request* or a *Response*. Using these data items, a piece of equipment changes the state of its *Request* or *Response* to indicate information that can be read by the other piece of equipment. To aid in understanding how the *Interaction Model* functions, one can view this *Interaction Model* as a simple state machine.

The interaction between two pieces of equipment can be described as follows. When the *Requester* wants an activity to be performed, it transitions its *Request* state from a `READY` state to an `ACTIVE` state. In turn, when the client on the *Responder* reads this information and interprets the *Request*, the *Responder* announces that it is performing the requested task by changing its response state to `ACTIVE`. When the action is finished, the *Responder* changes its response state to `COMPLETE`. This pattern of *Request* and *Response* provides the basis for the coordination of actions between pieces of equipment. These actions are implemented using `EVENT` category data items. (See *Section 4* for details on the `Event` type data items defined for *Interfaces*.)

Note: The implementation details of how the *Responder* piece of equipment reacts to the *Request* and then completes the requested task are up to the implementer.

The diagram below provides an example of the *Request* and *Response* state machine:

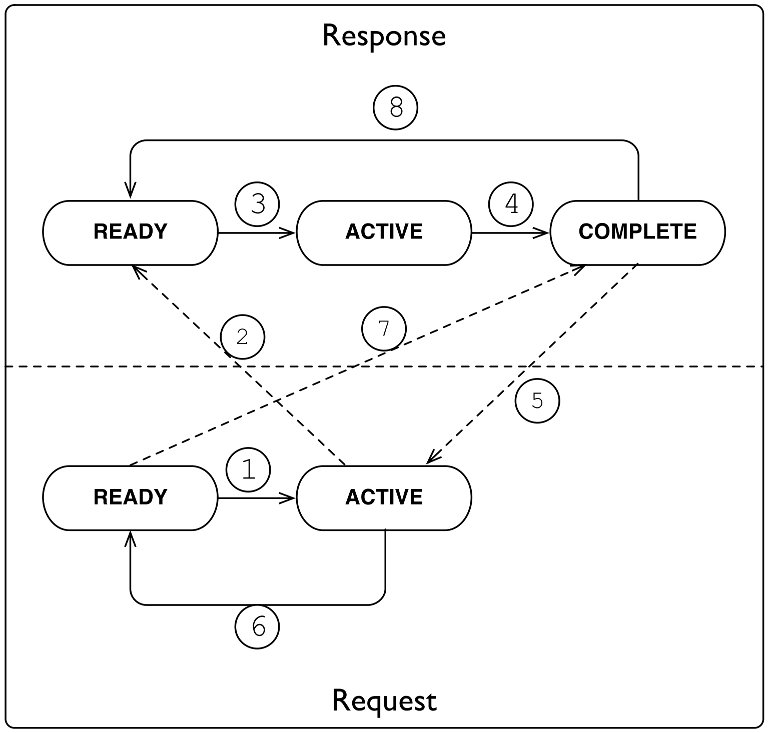

Figure 2: *Request* and *Response* Overview

The initial condition of both the *Request* and *Response* states on both pieces of equipment is `READY`. The dotted lines indicate the on-going communications that occur to monitor the progress of the interactions between the pieces of equipment.

The interaction between the pieces of equipment as illustrated in *Figure 2* progresses through the following sequence:

Step	Description
1	The *Request* transitions from `READY` to `ACTIVE` signaling that a service is needed.
2	The *Response* detects the transition of the *Request*.
3	The *Response* transitions from `READY` to `ACTIVE` indicating that it is performing the action.
4	Once the action has been performed, the *Response* transitions to `COMPLETE`.
5	The *Request* detects the action is `COMPLETE`.
6	The *Request* transitions back to `READY` acknowledging that the service has been performed.
7	The *Response* detects the *Request* has returned to `READY`.
8	In recognition of this acknowledgement, the *Response* transitions back to `READY`.

After the final action has been completed, both pieces of equipment are back in the `READY` state indicating that they are able to perform another action.

4 *Interfaces* for *Devices* and *Streams Information Models*

The *Interaction Model* for implementing *Interfaces* is defined in the MTConnect® Standard as an extension to the `MTConnectDevices` and `MTConnectStreams` XML documents.

A piece of equipment **MAY** support multiple different *Interfaces*. Each piece of equipment supporting *Interfaces* **MUST** organize the information associated with each *Interface* in a *Top Level* component called `Interfaces`. Each individual *Interface* is modeled as a *Lower Level* component called `Interface`. `Interface` is an abstract type XML element and will be replaced in the XML documents by specific `Interface` types defined below. The data associated with each *Interface* is modeled as data items within each of these *Lower Level* `Interface` components.

The following XML tree illustrates where `Interfaces` is modeled in the *Device Information Model* for a piece of equipment.

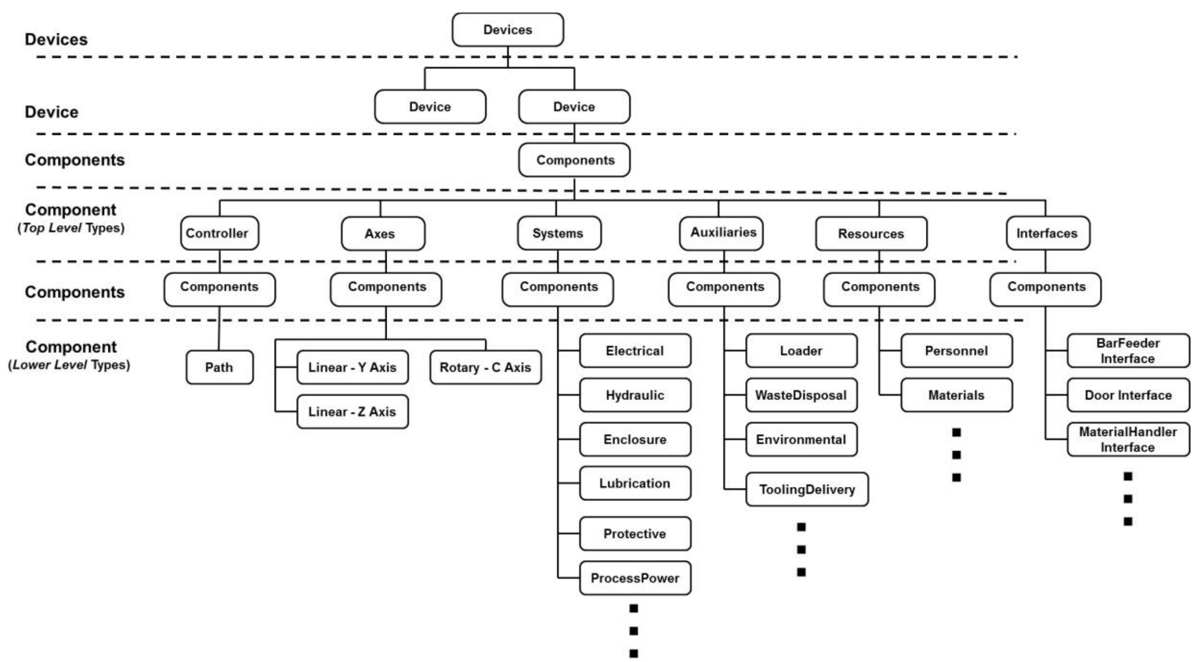

Figure 3: `Interfaces` as a *Structural Element*

4.1 Interfaces

`Interfaces` is an XML *Structural Element* in the `MTConnectDevices` XML document. `Interfaces` is a container type XML element. `Interfaces` is used to group information describing *Lower Level* `Interface` XML elements, which each provide information for an individual *Interface*.

If the `Interfaces` container appears in the XML document, it **MUST** contain one or more `Interface` type XML elements.

4.2 Interface

`Interface` is the next level of *Structural Element* in the `MTConnectDevices` XML document. As an abstract type XML element, `Interface` will be replaced in the XML documents by specific `Interface` types defined below.

Each `Interface` is also a container type element. As a container, the `Interface` XML element is used to organize information required to implement the *Interaction Model* for an *Interface*. It also provides structure for describing the *Lower Level Structural Elements* associated with the `Interface`. Each `Interface` contains *Data Entities* available from the piece of equipment that may be needed to coordinate activities with associated pieces of equipment.

The information provided by a piece of equipment for each *Interface* is returned in a `ComponentStream` container of an `MTConnectStreams` document in the same manner as all other types of components.

4.2.1 XML Schema Structure for `Interface`

The following XML schema represents the structure of an `Interface` XML element.

The schema for an `Interface` element is the same as defined for `Component` elements described in *Section 4.4* in *Part 2.0 – Devices Information Model* of the MTConnect Standard. The figure below shows the attributes defined for `Interface` and the elements that may be associated with `Interface`.

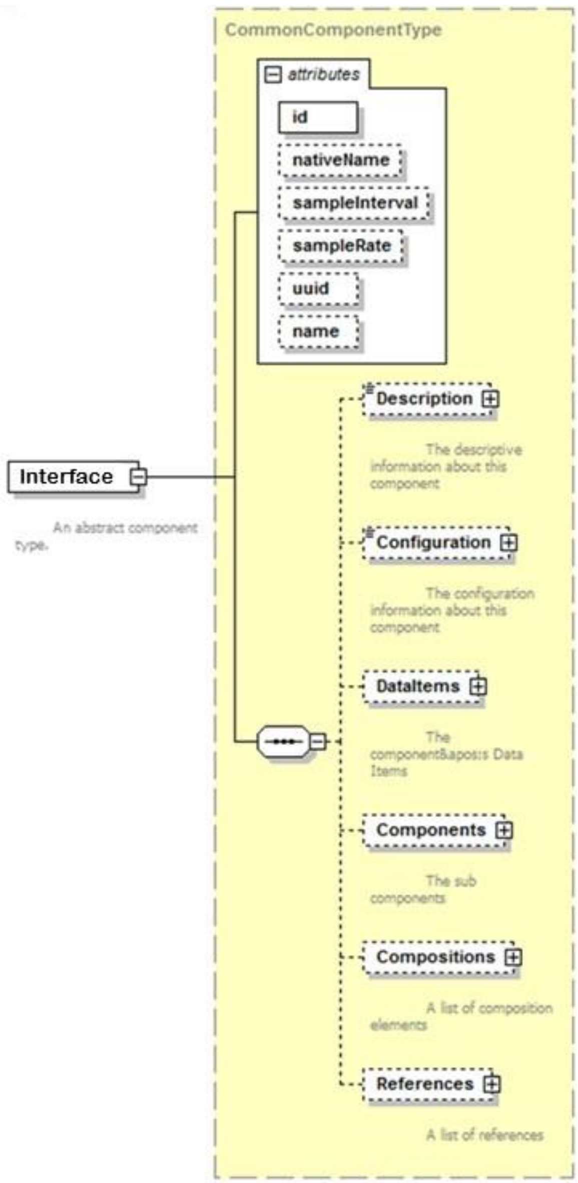

Figure 4: `Interface` Schema

Refer to *Part 2.0 – Devices Information Model*, *Section 4.4* for complete descriptions of the attributes and elements that are illustrated above for `Interface`.

4.2.2 `Interface` Types

As an abstract type XML element, `Interface` is replaced in the `MTConnectDevices` document with a XML element representing a specific type of *Interface*. An initial list of `Interface` types is defined below.

Interface	Description
`BarFeederInterface`	`BarFeederInterface` provides the set of information used to coordinate the operations between a Bar Feeder and another piece of equipment. Bar Feeder is a piece of equipment that pushes bar stock (i.e., long pieces of material of various shapes) into an associated piece of equipment – most typically a lathe or turning center.
`MaterialHandlerInterface`	`MaterialHandlerInterface` provides the set of information used to coordinate the operations between a piece of equipment and another associated piece of equipment used to automatically handle various types of materials or services associated with the original piece of equipment. A material handler is a piece of equipment capable of providing any one, or more, of a variety of support services for another piece of equipment or a process: Loading/unloading material or tooling Part inspection Testing Cleaning Etc. A robot is a common example of a material handler.
`DoorInterface`	`DoorInterface` provides the set of information used to coordinate the operations between two pieces of equipment, one of which controls the operation of a door. The piece of equipment that is controlling the door **MUST** provide the data item `Door_State` as part of the set of information provided.
`ChuckInterface`	`ChuckInterface` provides the set of information used to coordinate the operations between two pieces of equipment, one of which controls the operation of a chuck. The piece of equipment that is controlling the chuck **MUST** provide the data item `Chuck_State` as part of the set of information provided.

Note: Additional `Interface` types may be defined in future releases of the MTConnect Standard.

In order to implement the *Interaction Model* for *Interfaces*, each piece of equipment associated with an *Interface* **MUST** provide an `Interface` XML element for that type of *Interface*. A piece of equipment **MAY** support any number of unique *Interfaces*.

4.2.3 Data for `Interface`

Each *Interface* **MUST** provide (1) the data associated with the specific *Interface* to implement the *Interaction Model* and (2) any additional data that may be needed by another piece of equipment to understand the operating states and conditions of the first piece of equipment as it applies to the *Interface.*

Details on data items specific to the *Interaction Model* for each type of *Interface* are provided in *Section 4.2.4*.

An implementer may choose any other data available from a piece of equipment to describe the operating states and other information needed to support an *Interface.*

4.2.3.1 `References` for `Interface`

Some of the data items needed to support a specific *Interface* may already be defined elsewhere in the XML document for a piece of equipment. However, the implementer may not be able to directly associate this data with the *Interface* since the MTConnect Standard does not permit multiple occurrences of a piece of data to be configured in a XML document. `References` provides a mechanism for associating information defined elsewhere in the *Information Model* for a piece of equipment with a specific *Interface*.

`References` is an XML container that organizes pointers to information defined elsewhere in the XML document for a piece of equipment. `References` **MAY** contain one or more `Reference` XML elements.

`Reference` is an XML element that provides an individual pointer to information that is associated with another *Structural Element* or *Data Entity* defined elsewhere in the XML document that is also required for an *Interface*.

`References` is an economical syntax for providing interface specific information without directly duplicating the occurrence of the data. It provides an efficient, near-time, information flow between pieces of equipment.

For more information on the definition for `References` and `Reference`, see *Section 4.7* and *4.8* of *Part 2.0 - Devices Information Model*.

4.2.4 Data Items for `Interface`

Each `Interface` XML element contains data items which are used to communicate information required to execute the *Interface*. When these data items are read by another piece of equipment, that piece of equipment can then determine the actions that it may take based upon that data.

Some data items **MAY** be directly associated with the `Interface` element and others will be organized in a *Lower Level* `References` XML element.

It is up to an implementer to determine which additional data items are required for a particular *Interface*.

The data items that have been specifically defined to support the implementation of an *Interface* are provided below.

4.2.4.1 **INTERFACE_STATE for Interface**

INTERFACE_STATE is a data item specifically defined for *Interfaces*. It defines the operational state of the *Interface*. This is an indicator identifying whether the *Interface* is functioning or not.

An INTERFACE_STATE data item **MUST** be defined for every Interface XML element.

INTERFACE_STATE is reported in the MTConnectStreams XML document as InterfaceState. InterfaceState reports one of two states – ENABLED or DISABLED, which are provided in the CDATA for InterfaceState.

The table below shows both the INTERFACE_STATE data item as defined in the MTConnectDevices document and the corresponding *Element Name* that **MUST** be reported in the MTConnectStreams document.

EVENT **Data Item Type**	**Event** ***Element Name***	**Description and** **Valid Data Values**
INTERFACE_STATE	InterfaceState	The current functional or operational state of an Interface type element indicating whether the *Interface* is active or not currently functioning. Valid Data Values: - ENABLED: The *Interface* is currently operational and performing as expected. - DISABLED: The *Interface* is currently not operational. When the INTERFACE_STATE is DISABLED, the state of all data items that are specific for the *Interaction Model* associated with that *Interface* **MUST** be set to NOT_READY.

4.2.4.2 Specific Data Items for the *Interaction Model* for `Interface`

A special set of data items have been defined to be used in conjunction with `Interface` type elements. When modeled in the `MTConnectDevices` document, these data items are all *Data Entities* in the `EVENT` category (See *Part 3.0 – Streams Information Model* for details on how the corresponding data items are reported in the `MTConnectStreams` document). They provide information from a piece of equipment to *Request* a service to be performed by another associated piece of equipment; and for the associated piece of equipment to indicate its progress in performing its *Response* to the *Request* for service.

Many of the data items describing the services associated with an *Interface* are paired to describe two distinct actions – one to *Request* an action to be performed and a second to reverse the action or to return to an original state. For example, a `DoorInterface` will have two actions `OPEN_DOOR` and `CLOSE_DOOR`. An example of an implementation of this would be a robot that indicates to a machine that it would like to have a door opened so that the robot could extract a part from the machine and then asks the machine to close that door once the part has been removed.

When these data items are used to describe a service associated with an *Interface*, they **MUST** have one of the following two `subType` elements: `REQUEST` or `RESPONSE`. These `subType` elements **MUST** be specified to define whether the piece of equipment is functioning as the *Requester* or *Responder* for the service to be performed. The *Requester* **MUST** specify the `REQUEST` `subType` for the data item and the *Responder* **MUST** specify a corresponding `RESPONSE` `subType` for the data item to enable the coordination between the two pieces of equipment.

These data items and their associated `subType` provide the basic structure for implementing the *Interaction Model* for an *Interface*.

The table below provides a list of the data items that have been defined to identify the services to be performed for or by a piece of equipment associated with an *Interface*.

The table also provides the corresponding transformed *Element Name* for each data item that **MAY** be returned by an *MTConnect Agent* as an `Event type` XML *Data Entity* in the `MTConnectStreams` XML document. The *Controlled Vocabulary* for each of these data items are defined below in *Section 4.2.4.3*.

EVENT Data Item **Type**	Event *Element Name*	Description
`MATERIAL_FEED`	`MaterialFeed`	Service to advance material or feed product to a piece of equipment from a continuous or bulk source.
`MATERIAL_CHANGE`	`MaterialChange`	Service to change the type of material or product being loaded or fed to a piece of equipment.
`MATERIAL_RETRACT`	`MaterialRetract`	Service to remove or retract material or product.
`PART_CHANGE`	`PartChange`	Service to change the part or product associated with a piece of equipment to a different part or product.
`MATERIAL_LOAD`	`MaterialLoad`	Service to load a piece of material or product.
`MATERIAL_UNLOAD`	`MaterialUnload`	Service to unload a piece of material or product.
`OPEN_DOOR`	`OpenDoor`	Service to open a door.
`CLOSE_DOOR`	`CloseDoor`	Service to close a door.
`OPEN_CHUCK`	`OpenChuck`	Service to open a chuck.
`CLOSE_CHUCK`	`CloseChuck`	Service to close a chuck

4.2.4.3 `Event` States for *Interfaces*

For each of the data items above, the *Valid Data Values* for the CDATA that is returned for these data items in the MTConnectStreams document is defined by a *Controlled Vocabulary*. This *Controlled Vocabulary* represents the state information to be communicated by a piece of equipment for the data items defined in the table above.

The *Request* portion of the *Interaction Model* for *Interfaces* has four states as defined in the table below:

Request State	Description
NOT_READY	The *Requester* is not ready to make a *Request*.
READY	The *Requester* is prepared to make a *Request*, but no *Request* for service is required. The *Requester* will transition to ACTIVE when it needs a service to be performed.
ACTIVE	The *Requester* has initiated a *Request* for a service and the service has not yet been completed by the *Responder*.
FAIL	CONDITION 1: When the *Requester* has detected a failure condition, it indicates to the *Responder* to either not initiate an action or stop its action before it completes by changing its state to FAIL. CONDITION 2: If the *Responder* changes its state to FAIL, the *Requester* **MUST** change its state to FAIL. ACTIONS: After detecting a failure, the *Requester* **SHOULD NOT** change its state to any other value until the *Responder* has acknowledged the FAIL state by changing its state to FAIL. Once the FAIL state has been acknowledged by the *Responder*, the *Requester* may attempt to clear its FAIL state. As part of the attempt to clear the FAIL state, the *Requester* **MUST** reset any partial actions that were initiated and attempt to return to a condition where it is again ready to perform a service. If the recovery is successful, the *Requester* changes its *Request* state from FAIL to READY. If for some reason the *Requester* is not again prepared to perform a service, it transitions its state from FAIL to NOT_READY.

The following diagram shows a graphical representation of the possible state transitions for a *Request*:

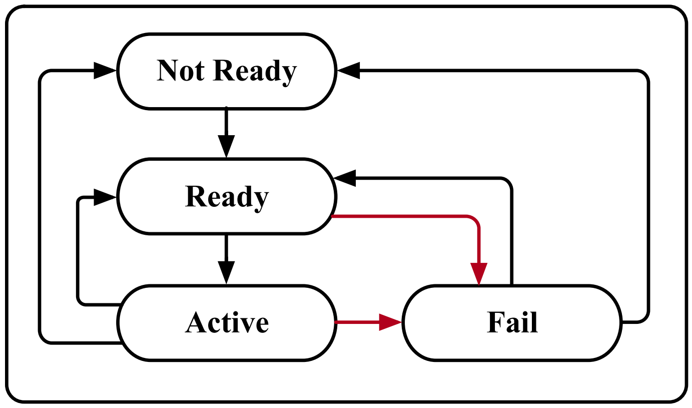

Figure 5: *Request* State Diagram

The *Response* portion of the *Interaction Model* for *Interfaces* has five states as defined in the table below:

Response State	Description
`NOT_READY`	The *Responder* is not ready to perform a service.
`READY`	The *Responder* is prepared to react to a *Request*, but no *Request* for service has been detected. The *Responder* **MUST** transition to **`ACTIVE`** to inform the *Requester* that it has detected and accepted the *Request* and is in the process of performing the requested service. If the *Responder* is not ready to perform a *Request*, it **MUST** transition to a `NOT_READY` state.
`ACTIVE`	The *Responder* has detected and accepted a *Request* for a service and is in the process of performing the service, but the service has not yet been completed. In normal operation, the *Responder* **MUST NOT** change its state to `ACTIVE` unless the *Requester* state is `ACTIVE`.
`FAIL`	CONDITION 1: The *Responder* has failed while executing the actions required to perform a service and the service has not yet been completed or the *Responder* has detected that the *Requestor* has unexpectedly changed state. CONDITION 2: If the *Requester* changes its state to `FAIL`, the *Responder* **MUST** change its state to `FAIL`. ACTIONS: After entering a `FAIL` state, the *Responder* **SHOULD NOT** change its state to any other value until the *Requester* has acknowledged the `FAIL` state by changing its state to `FAIL`. Once the `FAIL` state has been acknowledged by the *Requester*, the *Responder* may attempt to clear its `FAIL` state. As part of the attempt to clear the `FAIL` state, the *Responder* **MUST** reset any partial actions that were initiated and attempt to return to a condition where it is again ready to perform a service. If the recovery is successful, the *Responder* changes its *Response* state from `FAIL` to `READY`. If for some reason the *Responder* is not again prepared to perform a service, it transitions its state from `FAIL` to `NOT_READY`.
`COMPLETE`	The *Responder* has completed the actions required to perform the service. The *Responder* **MUST** remain in the `COMPLETE` state until the *Requester* acknowledges that the service is complete by changing its state to `READY`. At that point, the *Responder* **MUST** change its state to either `READY` if it is again prepared to perform a service or `NOT_READY` if it is not prepared to perform a service.

The state values described in the above tables **MUST** be provided in the CDATA for each of the *Interface* specific data items provided in the `MTConnectStreams` document.

The following diagram shows a graphical representation of the possible state transitions for a *Response*:

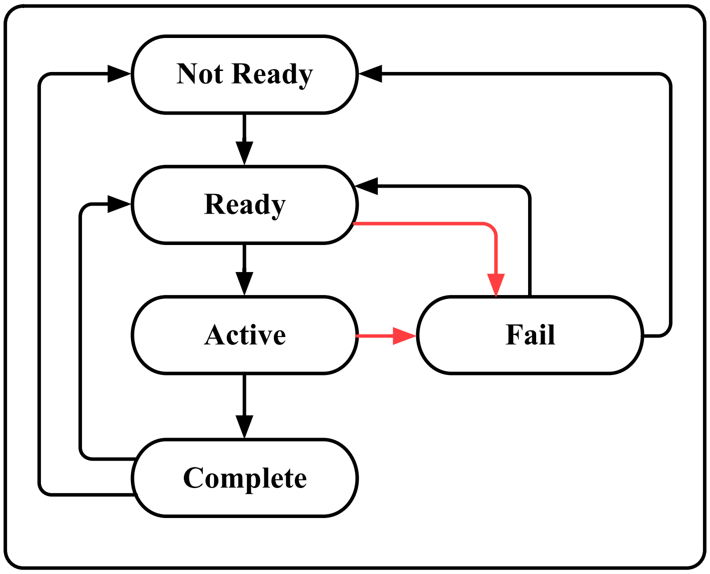

Figure 6: *Response* State Diagram

5 Operation and Error Recovery

The *Request/Response* state model implemented for *Interfaces* may also be represented by a graphical model. The following scenario demonstrates the state transitions that occur during a successful *Request* for service and the resulting *Response* to fulfill that service *Request*.

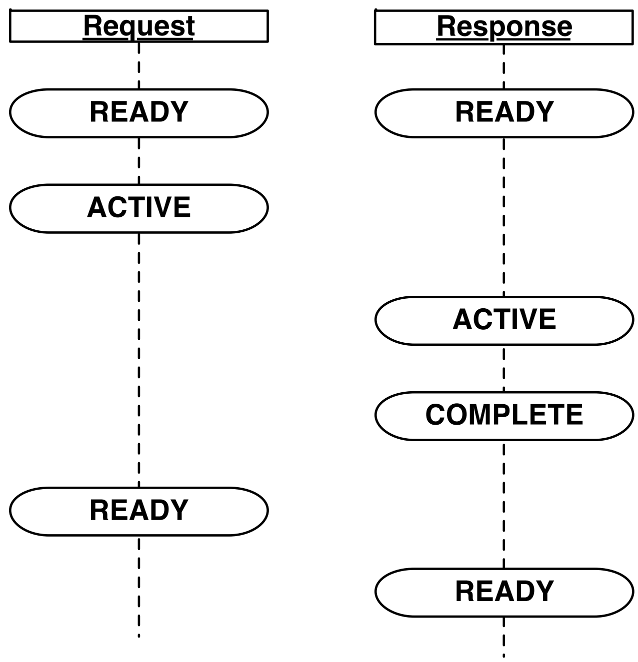

Figure 7: Success Scenario

5.1 *Request/Response* Failure Handling and Recovery

A significant feature of the *Request/Response Interaction Model* is the ability for either piece of equipment to detect a failure associated with either the *Request* or *Response* actions. When either a failure or unexpected action occurs, the *Request* and the *Response* portion of the *Interaction Model* can announce a `FAIL` state upon detecting a problem. The following are graphical models describing multiple scenarios where either the *Request*er or *Responder* detects and reacts to a failure. In these examples, either the *Requester* or *Responder* announces the detection of a failure by setting either the *Request* or the *Response* state to `FAIL`.

Once a failure is detected, the *Interaction Model* provides information from each piece of equipment as they attempt to recover from a failure, reset all of their functions associated with the *Interface* to their original state, and return to normal operation.

The following are scenarios that describe how pieces of equipment may react to different types of failures and how they indicate when they are again ready to request a service or respond to a request for service after recovering from those failures:

Scenario #1 – *Responder* Fails Immediately

In this scenario, a failure is detected by the *Responder* immediately after a *Request* for service has been initiated by the *Requester*.

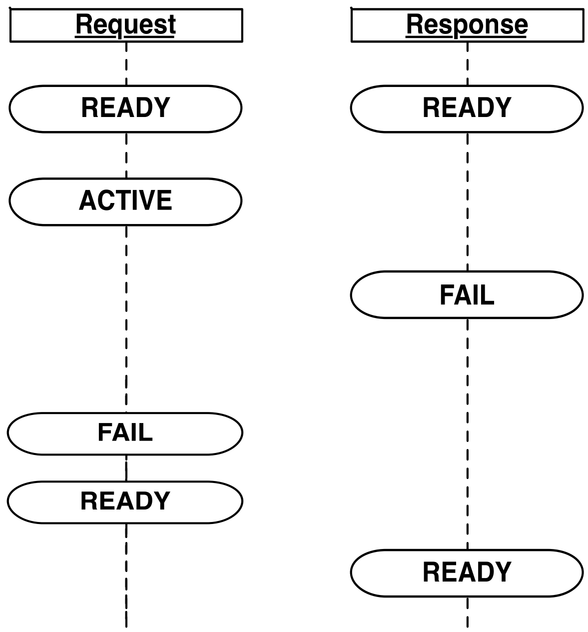

Figure 8: *Responder* – Immediate Failure

In this case, the *Request* transitions to `ACTIVE` and the *Responder* immediately detects a failure before it can transition the *Response* state to `ACTIVE`. When this occurs, the *Responder* transitions the *Response* state to `FAIL`.

After detecting that the *Responder* has transitioned its state to `FAIL`, the *Requester* **MUST** change its state to `FAIL`.

The *Requester*, as part of clearing a failure, resets any partial actions that were initiated and attempts to return to a condition where it is again ready to request a service. If the recovery is successful, the *Requester* changes its state from `FAIL` to `READY`. If for some reason the *Requester* cannot return to a condition where it is again ready to request a service, it transitions its state from `FAIL` to `NOT_READY`.

The *Responder*, as part of clearing a failure, resets any partial actions that were initiated and attempts to return to a condition where it is again ready to perform a service. If the recovery is successful, the *Responder* changes its *Response* state from `FAIL` to `READY`. If for some reason the *Responder* is not again prepared to perform a service, it transitions its state from `FAIL` to `NOT_READY`.

<u>Scenario #2 – *Responder* Fails While Providing a Service</u>

This is the most common failure scenario. In this case, the *Responder* will begin the actions required to provide a service. During these actions, the *Responder* detects a failure and transitions its *Response* state to `FAIL`.

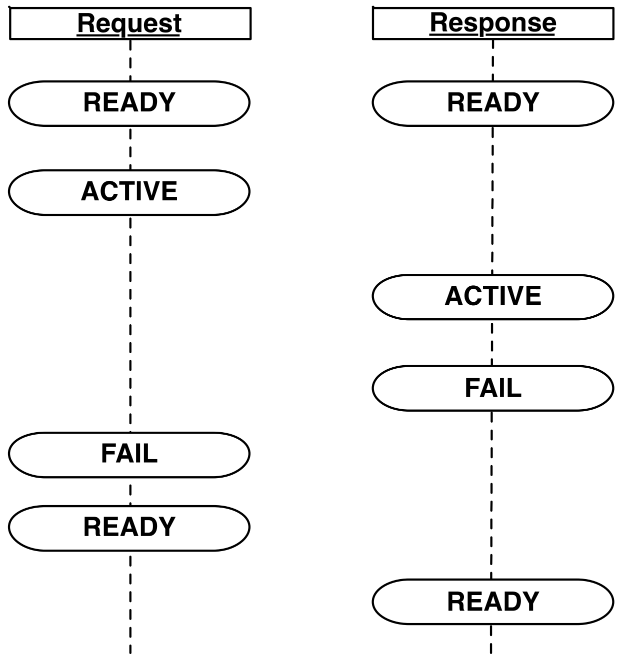

Figure 9: *Responder* Fails While Providing a Service

When a *Requester* detects a failure of a *Responder*, it transitions it state from `ACTIVE` to `FAIL`.

The *Requester* resets any partial actions that were initiated and attempts to return to a condition where it is again ready to request a service. If the recovery is successful, the *Requester* changes its state from `FAIL` to `READY` if the failure has been cleared and it is again prepared to request another service. If for some reason the *Requester* cannot return to a condition where it is again ready to request a service, it transitions its state from `FAIL` to `NOT_READY`.

The *Responder*, as part of clearing a failure, resets any partial actions that were initiated and attempts to return to a condition where it is again ready to perform a service. If the recovery is successful, the *Responder* changes its *Response* state from `FAIL` to `READY` if it is again prepared to perform a service. If for some reason the *Responder* is not again prepared to perform a service, it transitions its state from `FAIL` to `NOT_READY`.

<u>Scenario #3 – *Requester* Failure During a Service *Request*</u>

In this scenario, the *Responder* will begin the actions required to provide a service. During these actions, the *Requester* detects a failure and transitions its *Request* state to `FAIL`.

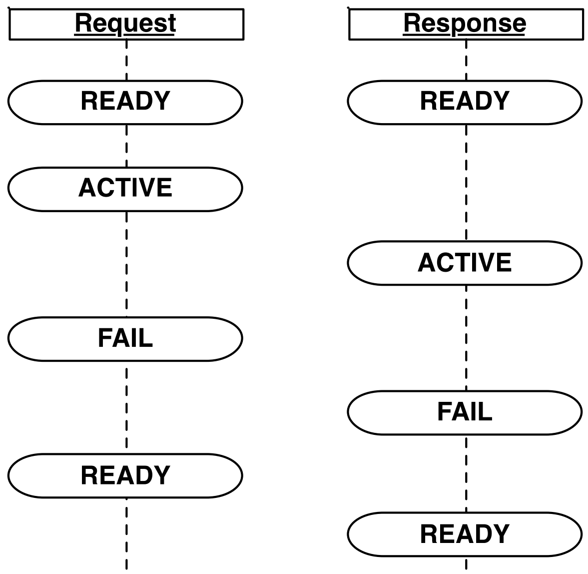

Figure 10: Requester Fails During a Service Request

When the *Responder* detects that the *Requester* has transitioned its *Request* state to `FAIL`, the *Responder* also transitions its *Response* state to `FAIL`.

The *Requester*, as part of clearing a failure, resets any partial actions that were initiated and attempts to return to a condition where it is again ready to request a service. If the recovery is successful, the *Requester* changes its state from `FAIL` to `READY`. If for some reason the *Requester* cannot return to a condition where it is again ready to request a service, it transitions its state from `FAIL` to `NOT_READY`.

The *Responder*, as part of clearing a failure, resets any partial actions that were initiated and attempts to return to a condition where it is again ready to perform a service. If the recovery is successful, the *Responder* changes its *Response* state from `FAIL` to `READY`. If for some reason the *Responder* is not again prepared to perform a service, it transitions its state from `FAIL` to `NOT_READY`.

Scenario #4 – *Requester* Changes to an Unexpected State While *Responder* is Providing a Service

In some cases, a *Requester* may transition to an unexpected state after it has initiated a *Request* for service.

As demonstrated below, the *Requester* has initiated a *Request* for service and its *Request* state has been changed to ACTIVE. The *Responder* begins the actions required to provide the service. During these actions, the *Requester* transitions its *Request* state back to READY before the *Responder* can complete its actions. This **SHOULD** be regarded as a failure of the *Requester*.

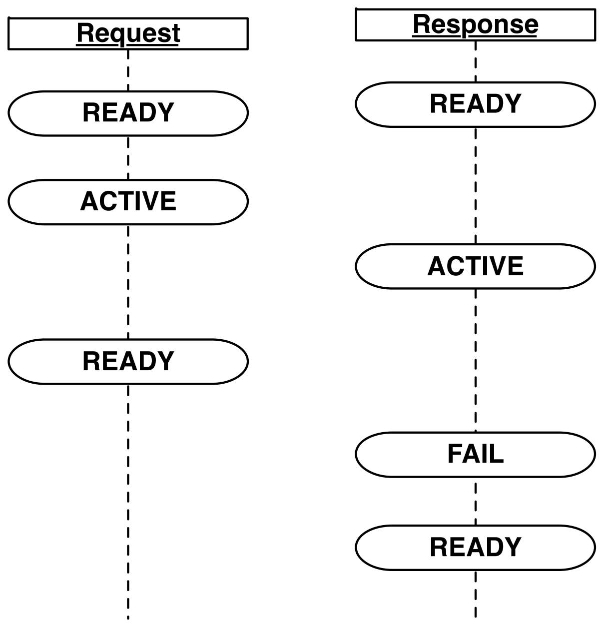

Figure 11: *Requester* Makes Unexpected State Change

In this case, the *Responder* reacts to this change of state of the *Requester* in the same way as though the *Requester* had transitioned its *Request* state to FAIL (i.e., the same as in Scenario #3 above).

At this point, the *Responder* then transitions its *Response* state to FAIL.

The *Responder* resets any partial actions that were initiated and attempts to return to its original condition where it is again ready to perform a service. If the recovery is successful, the *Responder* changes its *Response* state from FAIL to READY. If for some reason the *Responder* is not again prepared to perform a service, it transitions its state from FAIL to NOT_READY.

Note: The same scenario exists if the *Requester* transitions its *Request* state to NOT_READY. However, in this case, the *Requester* then transitions its *Request* state to READY after it resets all of its functions back to a condition where it is again prepared to make a *Request* for service.

Scenario #5 – *Responder* Changes to an Unexpected State While Providing a Service

Similar to Scenario #5, a *Responder* may transition to an unexpected state while providing a service.

As demonstrated below, the *Responder* is performing the actions to provide a service and the *Response* state is `ACTIVE`. During these actions, the *Responder* transitions its state to `NOT_READY` before completing its actions. This should be regarded as a failure of the *Responder*.

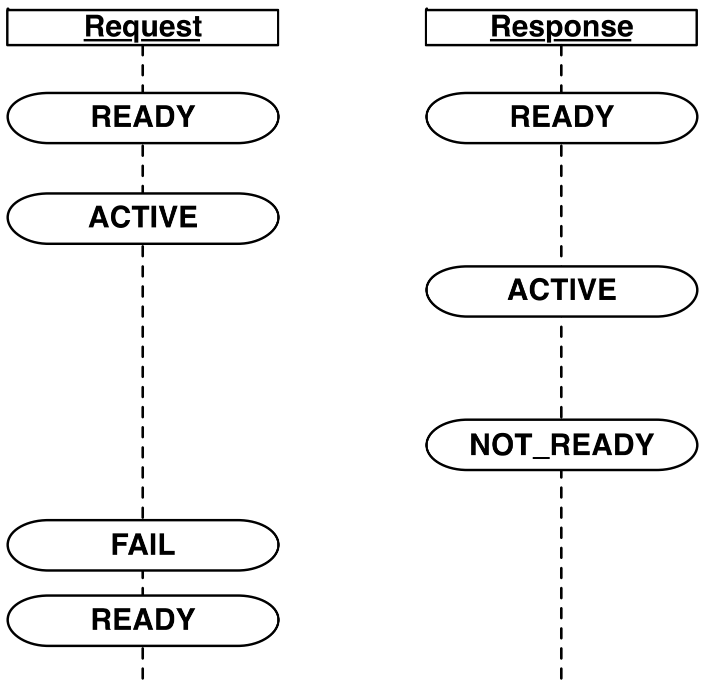

Figure 12: *Responder* Makes Unexpected State Change

Upon detecting an unexpected state change of the *Responder*, the *Requester* transitions its state to `FAIL`.

The *Requester* resets any partial actions that were initiated and attempts to return to a condition where it is again ready to request a service. If the recovery is successful, the *Requester* changes its state from `FAIL` to `READY`. If for some reason the *Requester* cannot return to a condition where it is again ready to request a service, it transitions its state from `FAIL` to `NOT_READY`.

Since the *Responder* has failed to an invalid state, the condition of the *Responder* is unknown. Where possible, the *Responder* should try to reset to an initial state.

The *Responder*, as part of clearing the cause for the change to the unexpected state, should attempt to reset any partial actions that were initiated and then return to a condition where it is again ready to perform a service. If the recovery is successful, the *Responder* changes its *Response* state from the unexpected state to `READY`. If for some reason the *Responder* is not again prepared to perform a service, it maintains its state as `NOT_READY`.

Scenario #6 – *Responder* or *Requester* Become UNAVAILABLE or Experience a Loss of Communications

In this scenario, a failure occurs in the communications connection between the *Responder* and *Requester*. This failure may result from the InterfaceState from either piece of equipment returning a value of UNAVAILABLE or one of the pieces of equipment does not provide a heartbeat within the desired amount of time (See *Part 1.0 - Overview and Functionality* for details on heartbeat).

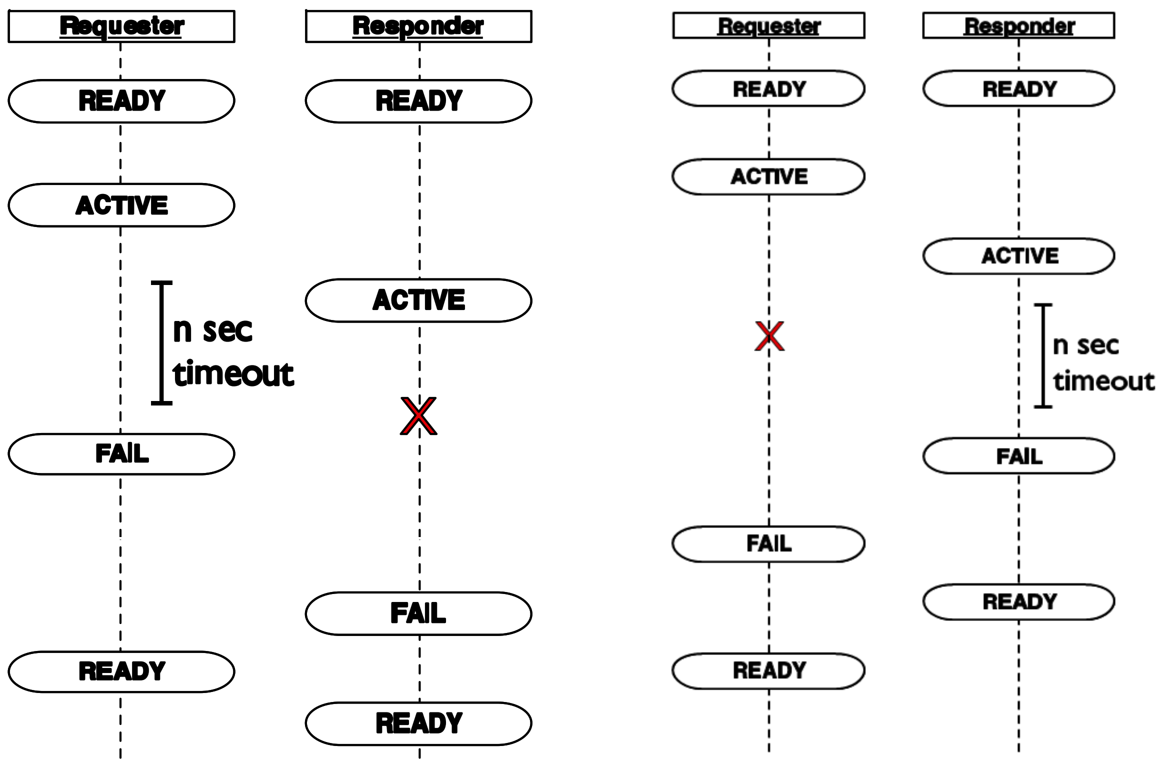

Figure 13: *Requester/Responder* Communication Failures

When one of these situations occurs, each piece of equipment assumes that there has been a failure of the other piece of equipment.

When normal communications are re-established, neither piece of equipment should assume that the *Request/Response* state of the other piece of equipment remains valid. Both pieces of equipment should set their state to FAIL.

The *Requester*, as part of clearing its FAIL state, resets any partial actions that were initiated and attempts to return to a condition where it is again ready to request a service. If the recovery is successful, the *Requester* changes its state from FAIL to READY. If for some reason the *Requester* cannot return to a condition where it is again ready to request a service, it transitions its state from FAIL to NOT_READY.

The *Responder*, as part of clearing its `FAIL` state, resets any partial actions that were initiated and attempts to return to a condition where it is again ready to perform a service. If the recovery is successful, the *Responder* changes its *Response* state from `FAIL` to `READY`. If for some reason the *Responder* is not again prepared to perform a service, it transitions its state from `FAIL` to `NOT_READY`.

Appendices

A. Bibliography

1. Engineering Industries Association. *EIA Standard* - *EIA-274-D*, Interchangeable Variable, Block Data Format for Positioning, Contouring, and Contouring/Positioning Numerically Controlled Machines. Washington, D.C. 1979.

2. ISO TC 184/SC4/WG3 N1089. *ISO/DIS 10303-238*: Industrial automation systems and integration Product data representation and exchange Part 238: Application Protocols: Application interpreted model for computerized numerical controllers. Geneva, Switzerland, 2004.

3. International Organization for Standardization. *ISO 14649*: Industrial automation systems and integration – Physical device control – Data model for computerized numerical controllers – Part 10: General process data. Geneva, Switzerland, 2004.

4. International Organization for Standardization. *ISO 14649*: Industrial automation systems and integration – Physical device control – Data model for computerized numerical controllers – Part 11: Process data for milling. Geneva, Switzerland, 2000.

5. International Organization for Standardization. *ISO 6983/1* – Numerical Control of machines – Program format and definition of address words – Part 1: Data format for positioning, line and contouring control systems. Geneva, Switzerland, 1982.

6. Electronic Industries Association. *ANSI/EIA-494-B-1992*, 32 Bit Binary CL (BCL) and 7 Bit ASCII CL (ACL) Exchange Input Format for Numerically Controlled Machines. Washington, D.C. 1992.

7. National Aerospace Standard. *Uniform Cutting Tests* - NAS Series: Metal Cutting Equipment Specifications. Washington, D.C. 1969.

8. International Organization for Standardization. *ISO 10303-11*: 1994, Industrial automation systems and integration Product data representation and exchange Part 11: Description methods: The EXPRESS language reference manual. Geneva, Switzerland, 1994.

9. International Organization for Standardization. *ISO 10303-21*: 1996, Industrial automation systems and integration -- Product data representation and exchange -- Part 21: Implementation methods: Clear text encoding of the exchange structure. Geneva, Switzerland, 1996.

10. H.L. Horton, F.D. Jones, and E. Oberg. *Machinery's handbook*. Industrial Press, Inc. New York, 1984.

11. International Organization for Standardization. *ISO 841-2001: Industrial automation systems and integration* - *Numerical control of machines* - *Coordinate systems and motion nomenclature.* Geneva, Switzerland, 2001.

12. ASME B5.57: *Methods for Performance Evaluation of Computer Numerically Controlled Lathes and Turning Centers,* 1998

13. ASME/ANSI B5.54: *Methods for Performance Evaluation of Computer Numerically Controlled Machining Centers. 2005.*

14. OPC Foundation. *OPC Unified Architecture Specification, Part 1: Concepts Version 1.00. July 28, 2006.*

15. IEEE STD 1451.0-2007*, Standard for a Smart Transducer Interface for Sensors and Actuators – Common Functions, Communication Protocols, and Transducer Electronic Data Sheet (TEDS) Formats, IEEE Instrumentation and Measurement Society, TC-9, The Institute of Electrical and Electronics Engineers, Inc., New York, N.Y. 10016, SH99684, October 5, 2007.*

16. IEEE STD 1451.4-1994*, Standard for a Smart Transducer Interface for Sensors and Actuators – Mixed-Mode Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats, IEEE Instrumentation and Measurement Society, TC-9, The Institute of Electrical and Electronics Engineers, Inc., New York, N.Y. 10016, SH95225, December 15, 2004.*